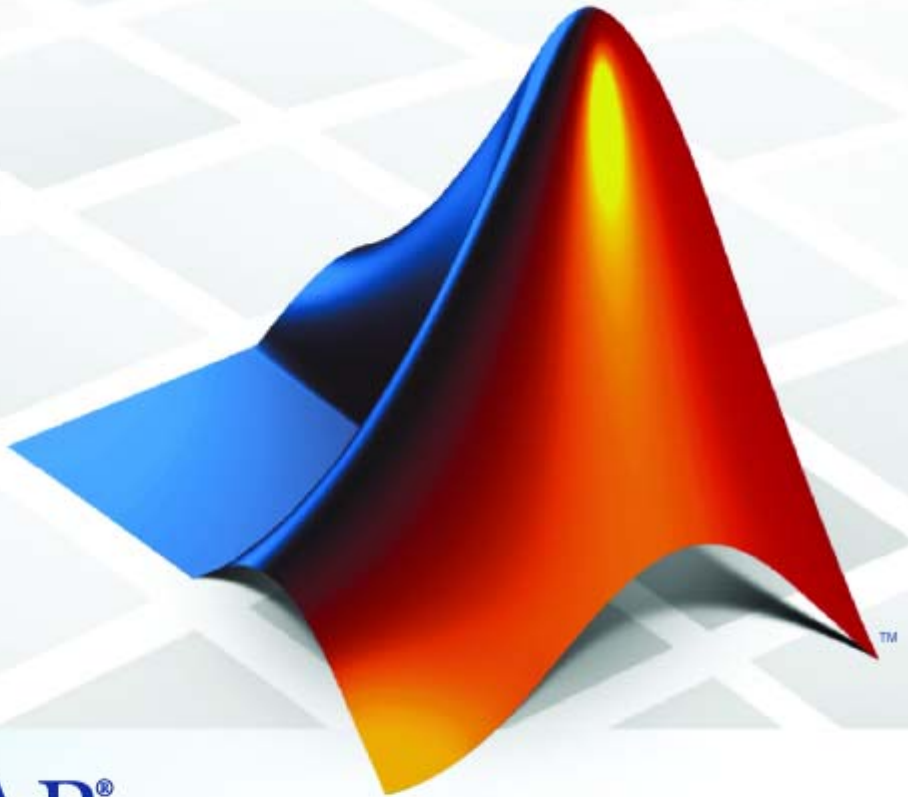


EDA Simulator Link™ 3

User's Guide

For Use with Cadence® Incisive®



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

EDA Simulator Link™ User's Guide

© COPYRIGHT 2006–2009 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2006 Online only
March 2007 Online only
September 2007 Online only
March 2008 Online only
October 2008 Online only
March 2009 Online only
September 2009 Online only

New for Version 1 (Release 2006+)
Updated for Version 2.0 (Release 2007a)
Updated for Version 2.1 (Release 2007b)
Updated for Version 2.2 (Release 2008a)
Updated for Version 2.3 (Release 2008b)
Updated for Version 2.4 (Release 2009a)
Updated for Version 3.0 (Release 2009b)

Getting Started

1

Product Overview	1-2
Integration with Other Products	1-2
Hardware Description Language (HDL) Support	1-4
Linking with MATLAB and the HDL Simulator	1-4
Linking with Simulink and the HDL Simulator	1-6
Communicating with MATLAB or Simulink and the HDL Simulator	1-7
Requirements	1-9
What You Need to Know	1-9
Required Products	1-9
Setting Up Your Environment for the EDA Simulator	
Link Software	1-12
Installing the Link Software	1-12
Installing Related Application Software	1-12
Using the EDA Simulator Link Libraries	1-13
Starting the HDL Simulator	1-18
Starting the HDL Simulator from MATLAB	1-18
Starting the Cadence Incisive HDL Simulator from a Shell	1-20
Learning More About the EDA Simulator Link	
Software	1-21
Documentation Overview	1-21
Online Help	1-22
Demos and Tutorials	1-23

Simulating an HDL Component in a MATLAB Test Bench Environment

2

Using MATLAB as a Test Bench	2-2
Overview to MATLAB Test Bench Functions	2-2
Workflow for Simulating an HDL Component with a MATLAB Test Bench Function	2-4
Code HDL Modules for Verification Using MATLAB ..	2-7
Overview to Coding HDL Modules for Use with MATLAB	2-7
Choosing an HDL Module Name	2-7
Specifying Port Direction Modes in HDL Components (MATLAB as Test Bench)	2-8
Specifying Port Data Types in HDL Components (MATLAB as Test Bench)	2-8
Compiling and Elaborating the HDL Design	2-10
Sample VHDL Entity Definition	2-10
Compiling and Debugging the HDL Model	2-10
Code an EDA Simulator Link MATLAB Test Bench Function	2-12
Process for Coding MATLAB EDA Simulator Link Functions	2-12
Syntax of a Test Bench Function	2-13
Sample MATLAB Test Bench Function	2-13
Place Test Bench Function on MATLAB Search Path ..	2-21
Use MATLAB which Function to Find Test Bench	2-21
Add Test Bench Function to MATLAB Search Path	2-21
Start hddaedmon to Provide Connection to HDL Simulator	2-22
Launch HDL Simulator for Use with MATLAB	2-24
Launching the HDL Simulator	2-24
Loading an HDL Design for Verification	2-24

Invoke matlabtb to Bind MATLAB Test Bench Function Calls	2-26
Invoking matlabtb	2-26
Binding the HDL Module Component to the MATLAB Test Bench Function	2-28
Schedule Options for a Test Bench Session	2-30
About Scheduling Options for Test Bench Sessions	2-30
Scheduling Test Bench Session Using matlabtb Arguments	2-30
Scheduling Test Bench M-Functions Using the tnext Parameter	2-32
Run MATLAB Test Bench Simulation	2-34
Process for Running MATLAB Test Bench Cosimulation ..	2-34
Checking the MATLAB Server's Link Status	2-34
Running a Test Bench Cosimulation	2-35
Applying Stimuli with the HDL Simulator force Command	2-37
Restarting a Test Bench Simulation	2-38
Stop Test Bench Simulation	2-39

Replacing an HDL Component with a MATLAB Component Function

3

Workflow for Creating a MATLAB Component Function for Use with the HDL Simulator	3-2
Overview to Using a MATLAB Function as a Component	3-5
Code HDL Modules for Visualization Using MATLAB ..	3-7
Code an EDA Simulator Link MATLAB Component Function	3-8

Overview to Coding an EDA Simulator Link Component Function	3-8
Syntax of a Component Function	3-9
Place Component Function on MATLAB Search Path	3-10
Start hdd daemon to Provide Connection to HDL Simulator	3-11
Start HDL Simulator for Use with MATLAB	3-12
Invoke matlabcp to Bind MATLAB Component Function Calls	3-13
About Binding Component Function Calls	3-13
Example of Binding Component Function Calls	3-13
Schedule Options for a Component Session	3-14
About Scheduling Options for Component Sessions	3-14
Scheduling Component Session Using matlabcp Arguments	3-15
Scheduling Test Bench M-Functions Using the tnext Parameter	3-16
Run MATLAB Component Simulation	3-17

Simulating an HDL Component in a Simulink Test Bench Environment

4

Workflow for Simulating an HDL Component in a Simulink Test Bench Environment	4-2
Overview to Using Simulink as a Test Bench	4-5
Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link ...	4-5

Introduction to the EDA Simulator Link HDL Cosimulation Block	4-8
Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator	4-10
Creating Your Simulink Model	4-10
Running and Testing a Hardware Model in Simulink	4-10
Adding a Value Change Dump (VCD) File (Optional)	4-10
Code an HDL Component for Use with Simulink Test Bench Applications	4-11
Overview to Coding HDL Components for Use with Simulink	4-11
Specifying Port Direction Modes in the HDL Component ..	4-11
Specifying Port Data Types in the HDL Component	4-11
Launch HDL Simulator for Test Bench Cosimulation with Simulink	4-12
Starting the HDL Simulator from MATLAB	4-12
Loading an Instance of an HDL Module for Cosimulation ..	4-12
Add the HDL Cosimulation Block to the Simulink Test Bench Model	4-13
Define the HDL Cosimulation Block Interface (Simulink as Test Bench)	4-14
Accessing the HDL Cosimulation Block Interface	4-14
Mapping HDL Signals to Block Ports	4-15
Specifying the Signal Data Types	4-27
Configuring the Simulink and HDL Simulator Timing Relationship	4-27
Configuring the Communication Link in the HDL Cosimulation Block	4-34
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box	4-36
Programmatically Controlling the Block Parameters	4-37
Start the HDL Simulation	4-40
Run a Test Bench Cosimulation Session (Simulink as Test Bench)	4-41

Setting Simulink Software Configuration Parameters	4-41
Determining an Available Socket Port Number	4-43
Checking the Connection Status	4-43
Running and Testing a Cosimulation Model	4-43
Avoiding Race Conditions in HDL Simulation When Cosimulating With the EDA Simulator Link HDL Cosimulation Block	4-44

Replacing an HDL Component with a Simulink Algorithm

5

Workflow for Using Simulink as HDL Component	5-2
Overview to Component Simulation with Simulink . . .	5-4
How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link	5-4
Creating the HDL Module	5-5
Introduction to the HDL Cosimulation Block (Simulink as Component)	5-6
Code an HDL Component for Use with Simulink Applications	5-7
Create Simulink Model for Component Cosimulation with the HDL Simulator	5-8
Launch HDL Simulator for Component Cosimulation with Simulink	5-9
Add the HDL Cosimulation Block to the Simulink Component Model	5-10
Define the HDL Cosimulation Block Interface	5-11
Start the Simulation in Simulink	5-12

Run a Component Cosimulation Session	5-13
--	------

Recording Simulink Signal State Transitions for Post-Processing

6

Adding a Value Change Dump (VCD) File	6-2
---	-----

Defining EDA Simulator Link M-Functions and Function Parameters

7

M-Function Syntax and Function Argument Definitions	7-2
Oscfilter Function Example	7-5
Gaining Access to and Applying Port Information	7-7

Additional Deployment Options

8

Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software	8-2
Using the Configuration and Diagnostic Script for UNIX/Linux	8-2
Performing Cross-Network Cosimulation	8-6
Why Perform Cross-Network Cosimulation?	8-6

Preparing for Cross-Network Cosimulation (MATLAB or Simulink)	8-6
Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB	8-8
Performing Cross-Network Cosimulation with the HDL Simulator and Simulink	8-10
Establishing EDA Simulator Link Machine	
Configuration Requirements	8-12
Valid Configurations For Using the EDA Simulator Link Software with MATLAB Applications	8-12
Valid Configurations For Using the EDA Simulator Link Software with Simulink Software	8-13
Specifying TCP/IP Socket Communication	8-15
Communication Modes and Socket Ports	8-15
Choosing TCP/IP Socket Ports	8-16
Specifying TCP/IP Values	8-18
TCP/IP Services	8-19
Improving Simulation Speed	8-20
Obtaining Baseline Performance Numbers	8-20
Analyzing Simulation Performance	8-20
Cosimulating Frame-Based Signals with Simulink	8-22

Advanced Operational Topics

9

Avoiding Race Conditions in HDL Simulators	9-2
Overview to Avoiding Race Conditions	9-2
Potential Race Conditions in Simulink Link Sessions	9-2
Potential Race Conditions in MATLAB Link Sessions	9-3
Further Reading	9-4
Performing Data Type Conversions	9-5
Converting HDL Data to Send to MATLAB	9-5
Array Indexing Differences Between MATLAB and HDL	9-7
Converting Data for Manipulation	9-9

Converting Data for Return to the HDL Simulator	9-10
Understanding How Simulink Software Drives	
Cosimulation Signals	9-14
Understanding the Representation of Simulation	
Time	9-15
Overview to the Representation of Simulation Time	9-15
Defining the Simulink and HDL Simulator Timing	
Relationship	9-16
Setting the Timing Mode with EDA Simulator Link	9-16
Relative Timing Mode	9-18
Absolute Timing Mode	9-20
Timing Mode Usage Considerations	9-21
Setting HDL Cosimulation Port Sample Times	9-23
Handling Multirate Signals	9-24
Understanding Block Simulation Latency	9-25
Overview to Block Simulation Latency	9-25
Interfacing with Continuous Time Signals	9-27

Functions — Alphabetical List

10 |

Blocks — Alphabetical List

11 |

Index

Getting Started

- “Product Overview” on page 1-2
- “Requirements” on page 1-9
- “Setting Up Your Environment for the EDA Simulator Link Software” on page 1-12
- “Starting the HDL Simulator” on page 1-18
- “Learning More About the EDA Simulator Link Software” on page 1-21

Product Overview

In this section...
“Integration with Other Products” on page 1-2
“Hardware Description Language (HDL) Support” on page 1-4
“Linking with MATLAB and the HDL Simulator” on page 1-4
“Linking with Simulink and the HDL Simulator” on page 1-6
“Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7

Integration with Other Products

The EDA Simulator Link™ cosimulation interface integrates MathWorks™ tools into the Electronic Design Automation (EDA) workflow. This integration enhances capabilities for field programmable gate array (FPGA) and application-specific integrated circuit (ASIC) development. The software provides a bidirectional link between the Cadence hardware description language (HDL) simulator, Incisive™, and The MathWorks™ MATLAB® and Simulink® products. Such linking makes direct hardware design verification and cosimulation possible. The integration of these tools allows users to apply each product to the tasks it does best:

- Cadence Incisive® — Hardware modeling in HDL and simulation
- MATLAB — Numerical computing, algorithm development, and visualization
- Simulink — Simulation of system-level designs and complex models

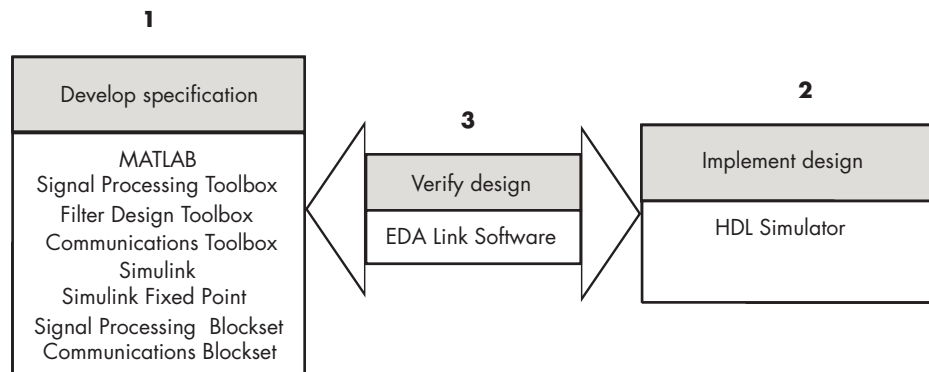
Note Cadence Incisive software may also be referred to as *the HDL simulator* throughout this document.

The EDA Simulator Link software consists of MATLAB functions that establish communication links between the HDL simulator and MATLAB and a library of Simulink blocks that you may use to include HDL simulator designs in Simulink models for cosimulation.

EDA Simulator Link software streamlines FPGA and ASIC development by integrating tools available for these processes:

- 1** Developing specifications for hardware design reference models
- 2** Implementing a hardware design in HDL based on a reference model
- 3** Verifying the design against the reference design

The following figure shows how the HDL simulator and MathWorks products fit into this hardware design scenario.



As the figure shows, EDA Simulator Link software connects tools that traditionally have been used discretely to perform specific steps in the design process. By connecting these tools, the link simplifies verification by allowing you to cosimulate the implementation and original specification directly. This cosimulation results in significant time savings and the elimination of errors inherent to manual comparison and inspection.

In addition to the preceding design scenario, EDA Simulator Link software enables you to work with tools in the following ways:

- Use MATLAB or Simulink to create test signals and software test benches for HDL code
- Use MATLAB or Simulink to provide a behavioral model for an HDL simulation
- Use MATLAB analysis and visualization capabilities for real-time insight into an HDL implementation
- Use Simulink to translate legacy HDL descriptions into system-level views

Note You can cosimulate a module using SystemVerilog, SystemC or both with MATLAB or Simulink using the EDA Simulator Link software. Write simple wrappers around the SystemC and make sure that the SystemVerilog cosimulation connections are to ports or signals of data types supported by the link cosimulation interface.

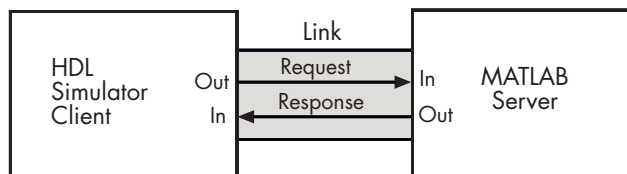
Hardware Description Language (HDL) Support

All EDA Simulator Link MATLAB functions and the HDL Cosimulation block offer the same language-transparent feature set for both Verilog and VHDL models.

EDA Simulator Link software also supports mixed-language HDL models (models with both Verilog and VHDL components), allowing you to cosimulate VHDL and Verilog signals simultaneously. Both MATLAB and Simulink software can access components in different languages at any level.

Linking with MATLAB and the HDL Simulator

When linked with MATLAB, the HDL simulator functions as the client, as the following figure shows.

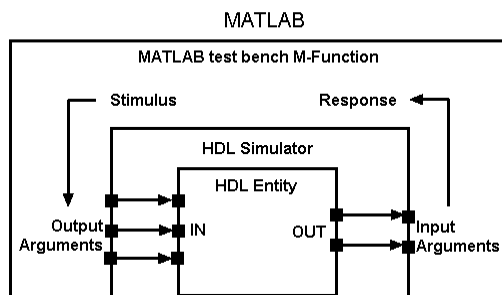


In this scenario, a MATLAB server function waits for service requests that it receives from an HDL simulator session. After receiving a request, the server establishes a communication link and invokes a specified MATLAB function that computes data for, verifies, or visualizes the HDL module (coded in VHDL or Verilog) that is under simulation in the HDL simulator.

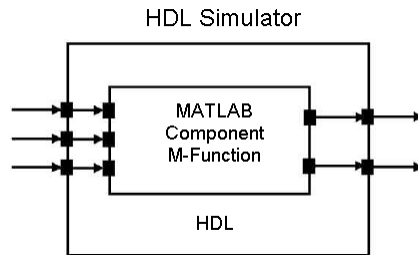
After the server is running, you can start and configure the HDL simulator or use with MATLAB with the supplied EDA Simulator Link function `nclaunch`. Required and optional parameters allow you to specify the following:

- Tcl commands that execute as part of startup
- A specific Cadence Incisive executable to start
- The name of an HDL startup file to store the complete startup script for future use or reference

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



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

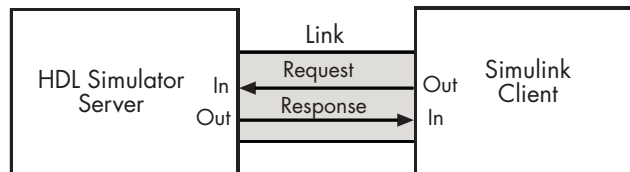


When you begin a specific test bench or component session, you specify parameters that identify the following information:

- The mode and, if appropriate, TCP/IP data necessary for connecting to a MATLAB server
- The MATLAB function that is associated with and executes on behalf of the HDL instance
- Timing specifications and other control data that specifies when the module's MATLAB function is to be called

Linking with Simulink and the HDL Simulator

When linked with Simulink, the HDL 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 an HDL simulator Wave window to monitor simulation timing diagrams.

Using the Block Parameters dialog box for an HDL Cosimulation block, you can configure the following:

- Block input and output ports that correspond to signals (including internal signals) of an HDL module. You can specify sample times and fixed-point data types for individual block output ports if desired.
- Type of communication and communication settings used for exchanging data between the simulation tools.
- Rising-edge or falling-edge clocks to apply to your module. The period of each clock is individually specifiable.
- Tcl commands to run before and after the simulation.

EDA Simulator Link software equips the HDL simulator with a set of customized functions. Using the function `hdlsimulink`, you execute the HDL simulator with an instance of an HDL module for cosimulation with Simulink. After the module is loaded, you can start the cosimulation session from Simulink.

EDA Simulator Link software also includes a block for generating value change dump (VCD) files. You can use VCD files generated with this block to perform the following tasks:

- View Simulink simulation waveforms in your HDL simulation environment
- Compare results of multiple simulation runs, using the same or different simulation environments
- Use as input to post-simulation analysis tools

Communicating with MATLAB or Simulink and the HDL Simulator

The mode of communication that you use for a link between the HDL simulator and MATLAB or Simulink depends on whether your simulation application runs in a local, single-system configuration or in a network configuration. If the HDL simulator and The MathWorks products can run locally on the same system and your application requires only one communication channel, you have the option of choosing between shared memory and TCP/IP socket communication. Shared memory communication provides optimal performance and is the default mode of communication.

TCP/IP socket mode is more versatile. You can use it for single-system and network configurations. This option offers the greatest scalability.

For more on TCP/IP socket communication, see “Specifying TCP/IP Socket Communication” on page 8-15.

Requirements

In this section...
“What You Need to Know” on page 1-9
“Required Products” on page 1-9

What You Need to Know

The documentation provided with the EDA Simulator Link software assumes users have a moderate level of prerequisite knowledge in the following subject areas:

- Hardware design and system integration
- VHDL and/or Verilog
- Cadence Incisive simulators
- MATLAB

Experience with Simulink and Simulink Fixed Point software is required for applying the Simulink component of the product.

Depending on your application, experience with the following MATLAB toolboxes and Simulink blocksets might also be useful:

- Signal Processing Toolbox™
- Filter Design Toolbox™
- Communications Toolbox™
- Signal Processing Blockset™
- Communications Blockset™
- Video and Image Processing Blockset™

Required Products

EDA Simulator Link software requires the following:

Platform	Visit the EDA Simulator Link requirements page on The MathWorks Web site for specific platforms supported with the current release.
Application software	Requires Cadence Incisive HDL Simulator, Incisive Design Team Simulator, or Incisive Enterprise Specman Simulator. Visit the EDA Simulator Link product requirements page on The MathWorks Web site for specific versions supported with the current release.
Application software required for cosimulation with Simulink	MATLAB Simulink Simulink Fixed Point Fixed-Point Toolbox
Optional application software	Communications Blockset Signal Processing Blockset Filter Design Toolbox Signal Processing Toolbox Video and Image Processing Blockset
	<hr/> Note Many EDA Simulator Link demos require one or more of the optional products listed. <hr/>
Platform-specific software	The EDA Simulator Link shared libraries (<code>liblfihdls*.so</code> , <code>liblfihdlc*.so</code>) are built using the <code>gcc</code> included in the Cadence Incisive simulator platform distribution. If you are linking your own applications into the HDL simulator, the recommendation is that you also build against this <code>gcc</code> . See the HDL simulator documentation for

more details about how to build and link
your own applications.

Setting Up Your Environment for the EDA Simulator Link Software

In this section...

“Installing the Link Software” on page 1-12

“Installing Related Application Software” on page 1-12

“Using the EDA Simulator Link Libraries” on page 1-13

Installing the Link Software

For details on how to install the EDA Simulator Link software, see the MATLAB installation instructions.

You can start using the link software right away by issuing the link HDL simulator launch command, `nclaunch`, from MATLAB, and providing the EDA Simulator Link library information and other required parameters (see “Using the EDA Simulator Link Libraries” on page 1-13). No special setup is required.

If you would like some assistance in running through the setup, you can diagnose your setup (correct omissions and errors) and also customize your setup for future invocations of `nclaunch` by following the process in “Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software” on page 8-2.

Installing Related Application Software

Based on your configuration decisions and the software required for your EDA Simulator Link application, identify software you need to install and where you need to install it. For example, if you need to run multiple instances of the link MATLAB server on different machines, you need to install MATLAB and any applicable toolbox software on multiple systems. Each instance of MATLAB can run only one instance of the server.

For details on how to install the HDL simulator, see the installation instructions for that product. For information on installing and activating

MathWorks products, see the MATLAB installation and activation instructions.

Using the EDA Simulator Link Libraries

In general, you want to use the same compiler for all libraries linked into the same executable. The link software provides many versions of the same library compilers that are available with the HDL simulators (usually some version of GCC). Using the same libraries ensures compatibility with other C++ libraries that may get linked into the HDL simulator, including SystemC libraries.

If you have any of these conditions, choose the version of the EDA Simulator Link library that matches the compiler used for that code:

- Link other third-party applications into your HDL simulator.
- Compile and link in SystemC code as part of your design or testbench.
- Write custom C/C++ applications and link them into your HDL simulator.

If you do not link any other code into your HDL simulator, you can use any version of the supplied libraries. The MATLAB `nclaunch` command chooses a default version of this library.

Note EDA Simulator Link software supports running in 32-bit mode on a 64-bit Solaris machine, but it does not support running on a 32-bit Solaris platform.

For examples on specifying EDA Simulator Link libraries when cosimulating across a network, see “Performing Cross-Network Cosimulation” on page 8-6.

Library Names

The EDA Simulator Link HDL libraries use the following naming format:

```
edalink/extensions/version/arch/lib{version_short_name}{client_server_tag}_{compiler_tag}
.{libext}
```

where

version	incisive
arch	linux32, linux64, solaris32, solaris64
version_short_name	lfihdl
client_server_tag	c or s (MATLAB or Simulink)
compiler_tag	gcc323, gcc346, gcc41, tmwsprow
libext	so

Not all combinations are supported. See “Default Libraries” on page 1-14 for valid combinations.

For more on MATLAB build compilers, see MATLAB Build Compilers — Release 2009a.

Default Libraries

EDA Simulator Link scripts fully support the use of default libraries.

The following table lists all the libraries shipped with the link software. The default libraries for each platform are in bold text.

Platform	MATLAB Library	Simulink Library
Linux32, Linux64	<ul style="list-style-type: none"> • liblfihdlc_gcc323.so (default for IUS 6.1 or unsupported releases) • liblfihdlc_gcc346.so • liblfihdlc_gcc41.so (default for IS 6.2, 8.1) • liblfihdlc_tmwgcc.so 	<ul style="list-style-type: none"> • liblfihdls_gcc323.so (default for IUS 6.1 or unsupported releases) • liblfihdls_gcc346.so • liblfihdls_gcc41.so (default for IUS 6.2, 8.1) • liblfihdls1_tmwgcc.so
Solaris32, Solaris64	<ul style="list-style-type: none"> • liblfihdlc_gcc323.so (default for IUS 6.1 or unsupported releases) • liblfihdlc_gcc41.so (default for IS 6.2, 8.1) 	<ul style="list-style-type: none"> • liblfihdls_gcc323.so (default for IUS 6.1 or unsupported releases) • liblfihdls_gcc41.so (default for IS 6.2, 8.1)

Platform	MATLAB Library	Simulink Library
	<ul style="list-style-type: none"> liblfihdlc_tmwspro.so 	<ul style="list-style-type: none"> liblfihdls_tmwspro.so

Using an Alternative Library

You can use a different HDL-side library by specifying it explicitly using the `libfile` parameter to the `nc1launch` MATLAB command. You should choose the version of the library that matches the compiler and system libraries you are using for any other C/C++ libraries linked into the HDL simulator. Depending on the version of your HDL simulator, you may need to explicitly set additional paths in the `LD_LIBRARY_PATH` environment variable.

For example, if you want to use a nondefault library:

- 1 Copy the system libraries from the MATLAB installation (found in `matlabroot/sys/os/platform`) to the machine with the HDL simulator (where `matlabroot` is your MATLAB installation and `platform` is one of the above architecture, for example, `linux32`).
- 2 Modify the `LD_LIBRARY_PATH` environment variable to add the path to the system libraries that were copied in step 1.

Example: EDA Simulator Link Alternate Library Using nclaunch. In this example, you are using the 32-bit Solaris version of IUS 5.83p2 on the same 64-bit Solaris machine which is running MATLAB. Because you have your own C++ application, and you are linking into ncsim which you used SunPro 11 to compile, you are using the EDA Simulator Link version compiled with SunPro 11, instead of using the default library version compiled with GCC 3.2.3.

In MATLAB:

```
>> currPath = getenv('PATH');
>> currLdPath = getenv('LD_LIBRARY_PATH');
>> setenv('PATH', ['/tools/IUS-583p2/bin:' currPath]);
>> nclaunch('tclstart', { 'exec ncvhdl inverter.vhd', ...
                        'exec ncelab -access +rwc inverter', ...
                        'hdlsimulink -gui inverter' }, ...
            'libfile', 'liblfihdls_tmwspro');
```

The PATH is changed to ensure we get the correct version of the HDL simulator tools. Note that the nclaunch MATLAB command will automatically detect the use of the 32-bit version of the HDL simulator and use the solaris32 library folder in the EDA Simulator Link installation; there is no need to specify the libdir parameter in this case.

The library resolution can be verified using ldd from within the ncsim console GUI.

```
ncsim> exec ldd /path/to/liblfihdls_tmwspro.so
libxnet.so.1 => /lib/libxnet.so.1
librt.so.1 => /lib/librt.so.1
libm.so.2 => /lib/libm.so.2
libc.so.1 => /lib/libc.so.1
libstlport.so.1 => /tools/SUNWspro_studio11_20070319/opt/SUNWspro/lib/stlport4/
libstlport.so.1
libCrun.so.1 => /usr/lib/libCrun.so.1
libaio.so.1 => /lib/libaio.so.1
libmd5.so.1 => /lib/libmd5.so.1
libm.so.1 => /lib/libm.so.1
libsocket.so.1 => /lib/libsocket.so.1
libnsl.so.1 => /lib/libnsl.so.1
```

```

libmp.so.2 => /lib/libmp.so.2
libscf.so.1 => /lib/libscf.so.1
libdoor.so.1 => /lib/libdoor.so.1
libuutil.so.1 => /lib/libuutil.so.1
/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1
/platform/SUNW,Sun-Blade-1000/lib/libmd5_psr.so.1

```

Example: EDA Simulator Link Alternate Library Using System Shell.

This example shows how to load a Cadence Incisive simulator session by explicitly specifying the EDA Simulator Link library (default or not). By explicitly using a system shell, you can execute this example on the same machine as MATLAB, on a different machine, and even on a machine with a different operating system.

In this example, you are running the 64-bit Linux version of Cadence Incisive 5.83p2; it does not matter what machine MATLAB is running on. Instead of using the default library version compiled with GCC 3.2.3 in the Cadence Incisive distribution, you are using the version compiled with GCC 3.4.6 in the Cadence Incisive distribution.

In a csh-compatible system shell:

```

csh> setenv PATH /tools/ius-583p2/lxx/tools/bin/64bit:${PATH}
csh> setenv LD_LIBRARY_PATH /tools/ius-583p2/lxx/tools/systemc/gcc/3.4.6-x86_64
    /install/lib64:${LD_LIBRARY_PATH}
csh> ncvhdl inverter.vhd
csh> ncelab -access +rwc inverter
csh> ncsim -tcl -loadvpi /tools/matlab-7b/toolbox/edalink/extensions/incisive/linux64
    /liblfihdlc_gcc346:matlabclient inverter.vhd

```

The PATH is changed to ensure we get the correct version of the Cadence Incisive tools. Although ncsim will automatically find any GCC libs in its installations, the LD_LIBRARY_PATH is changed to show how you might do this with a custom installation of GCC.

You can check the proper library resolution using ldd as in the previous example.

Starting the HDL Simulator

In this section...
“Starting the HDL Simulator from MATLAB” on page 1-18
“Starting the Cadence Incisive HDL Simulator from a Shell” on page 1-20

Starting the HDL Simulator from MATLAB

Start the Cadence Incisive or NC Simulator simulator directly from MATLAB or Simulink by calling the MATLAB function `nclaunch`. This function starts and configures the HDL simulator for use with the EDA Simulator Link software. By default, the function starts the first version of the simulator executable (`ncsim.exe`) that it finds on the system path (defined by the `path` variable), using a temporary file that is overwritten each time the HDL simulator starts.

Note If you want to start a different version of the simulator executable than the first one found on the system path, use the `setenv` and `getenv` MATLAB functions to set and get the environment of any sub-shells spawned by `UNIX()`, `DOS()`, or `system()`.

To start the HDL simulator from MATLAB, enter `nclaunch` at the MATLAB command prompt:

```
>> nclaunch('PropertyName', 'PropertyValue'...)
```

You can customize the startup file and communication mode to be used between MATLAB or Simulink and the HDL simulator by specifying the call to `nclaunch` with property name/property value pairs. Refer to `nclaunch` reference documentation for specific information regarding the property name/property value pairs.

Note The `nclaunch` command requires the use of property name/property value pairs. You get an error if you try to use the function without them.

See “Examples of Starting the Cadence Incisive Simulator from MATLAB” on page 1-19 for examples of using `nclaunch` with various property/name value pairs and other parameters.

When you specify a communication mode using `nclaunch`, the function applies the specified communication mode to all MATLAB or Simulink/HDL simulator sessions.

Examples of Starting the Cadence Incisive Simulator from MATLAB

The following example changes the folder location to `VHDLproj` and then calls the function `nclaunch`. Because the command line omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies Tcl commands that load and initialize the HDL simulator for test bench instance `modsimrand`.

```
cd VHDLproj
nclaunch('tclstart',...
        'hdlsimmatlab modsimrand; matlabtb modsimrand 10 ns -socket 4449')
```

The following example changes the folder location to `VHDLproj` and then calls the function `nclaunch`. Because the function call omits the `'hdlsimdir'` and `'startupfile'` properties, `nclaunch` creates a temporary file. The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in library `work` for cosimulation between `nclaunch` and Simulink. The `'socketsimulink'` property specifies TCP/IP socket communication on the same computer, using socket port 4449.

```
cd VHDLproj
nclaunch('tclstart', 'hdlsimulink work.parse', 'socketsimulink', '4449')
```

Another option is to bring `ncsim` up in the terminal instead of launching the Simvision GUI, thereby allowing you to interact with the simulation. This next example lists the steps necessary for you to do this:

- 1 Start `hdlldaemon` in MATLAB.
- 2 Start an `xterm` from MATLAB in the background (key point).

- 3** Run `ncsim` in the `xterm` shell having it call back to the `hdlserver` to run your `matlabcp` function as usual.
- 4** Have the `matlabcp` function touch a file to signal completion while an M script polls for completion.

The M script can then change test parameters and run more tests.

Starting the Cadence Incisive HDL Simulator from a Shell

To start the HDL simulator from a shell and include the EDA Simulator Link libraries, you need to first run the configuration script. See “Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software” on page 8-2.

After you have the configuration files, you can start the HDL simulator from the shell by typing:

```
% ncsim -f matlabconfigfile modelname
```

`matlabconfigfile` should be the name of the MATLAB configuration file you created with `syscheckin`. If you are connecting to Simulink, this should be the name of the Simulink configuration file. For example:

```
% ncsim -gui -f simulinkconfigfile modelname
```

Either way, you must also specify the path to the configuration file if it does not reside in the same folder as `ncsim.exe`.

You can also specify any other existing configuration files you may also be using with this call.

Learning More About the EDA Simulator Link Software

In this section...
“Documentation Overview” on page 1-21
“Online Help” on page 1-22
“Demos and Tutorials” on page 1-23

Documentation Overview

The following documentation is available with this product.

Chapter 1, “Getting Started”	Explains what the product is, the steps for installing and setting it up, how you might apply it to the hardware design process, and how to gain access to product documentation and online help. Directs you to product demos and tutorials.
Chapter 2, “Simulating an HDL Component in a MATLAB Test Bench Environment”	Explains how to code HDL models and MATLAB test bench functions for EDA Simulator Link MATLAB applications. Provides details on how the link interface maps HDL data types to MATLAB data types and vice versa. Explains how to start and control HDL simulator and MATLAB test bench sessions.
Chapter 3, “Replacing an HDL Component with a MATLAB Component Function”	Discusses the same topics as the chapter for test bench cosimulation using MATLAB software but instead using MATLAB to visualize an HDL module component.
Chapter 4, “Simulating an HDL Component in a Simulink Test Bench Environment”	Explains how to use the HDL simulator and Simulink for cosimulation modeling where Simulink acts as the test bench
Chapter 5, “Replacing an HDL Component with a Simulink Algorithm”	Explains how to use the HDL simulator and Simulink for cosimulation modeling where Simulink replaces an HDL component

Chapter 6, “Recording Simulink Signal State Transitions for Post-Processing”	Provides instruction for adding a Value Change Dump (VCD) file block to your Simulink model for signal state change capture.
Chapter 8, “Additional Deployment Options”	Contains several procedures for additional cosimulation arrangements: for example, performing cross-network cosimulation.
Chapter 9, “Advanced Operational Topics”	Contains a variety of topics that provide a deeper understanding of how cosimulation works.
Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”	Describes how to use the input and output arguments to an M-function created for use with the EDA Simulator Link commands <code>matlabtb</code> and <code>matlabcp</code> .
Chapter 10, “Functions — Alphabetical List”	Describes EDA Simulator Link functions for use with MATLAB.
Chapter 11, “Blocks — Alphabetical List”	Describes EDA Simulator Link blocks for use with Simulink.

Online Help

The following online help is available:


- Online help in the MATLAB Help browser. Click the EDA Simulator Link product link in the browser’s Contents or access using the MATLAB `doc` command at the MATLAB command prompt:

```
doc hdldaemon
```

- M-help for EDA Simulator Link MATLAB functions. This help is accessible with the MATLAB `help` command. For example, enter the command

```
help hdldaemon
```

at the MATLAB command prompt.

- Function help in the MATLAB command window by clicking the  icon.
- Block reference pages accessible through the Simulink interface.

Demos and Tutorials

The EDA Simulator Link software provides demos and tutorials to help you get started.

The demos give you a quick view of the product's capabilities and examples of how you might apply the product. You can run them with limited product exposure. You can find the EDA Simulator Link demos with the online documentation. To access demos, type at the MATLAB command prompt:

```
>> demos
```

Select **Links and Targets > EDA Simulator Link** from the navigational pane.

Simulating an HDL Component in a MATLAB Test Bench Environment

- “Using MATLAB as a Test Bench” on page 2-2
- “Code HDL Modules for Verification Using MATLAB ” on page 2-7
- “Code an EDA Simulator Link MATLAB Test Bench Function” on page 2-12
- “Place Test Bench Function on MATLAB Search Path” on page 2-21
- “Start hlddaemon to Provide Connection to HDL Simulator” on page 2-22
- “Launch HDL Simulator for Use with MATLAB” on page 2-24
- “Invoke matlabb to Bind MATLAB Test Bench Function Calls” on page 2-26
- “Schedule Options for a Test Bench Session” on page 2-30
- “Run MATLAB Test Bench Simulation” on page 2-34
- “Stop Test Bench Simulation” on page 2-39

Using MATLAB as a Test Bench

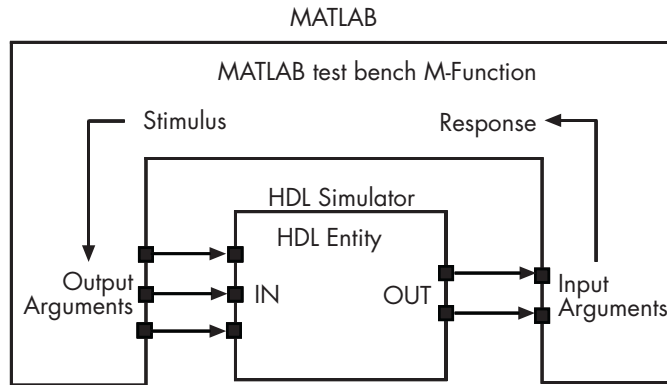
In this section...
“Overview to MATLAB Test Bench Functions” on page 2-2
“Workflow for Simulating an HDL Component with a MATLAB Test Bench Function” on page 2-4

Overview to MATLAB Test Bench Functions

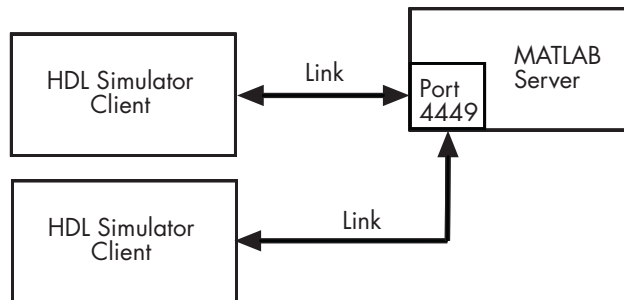
The EDA Simulator Link software provides a means for verifying HDL modules within the MATLAB environment. You do so by coding an HDL model and a MATLAB M-function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB test bench 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 M-function wraps around and communicates with the HDL simulator during a test bench simulation session.



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



The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to ensure the server can track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function `hdldaemon`, waits for connection requests from instances of the HDL simulator running on the same or different computers. When

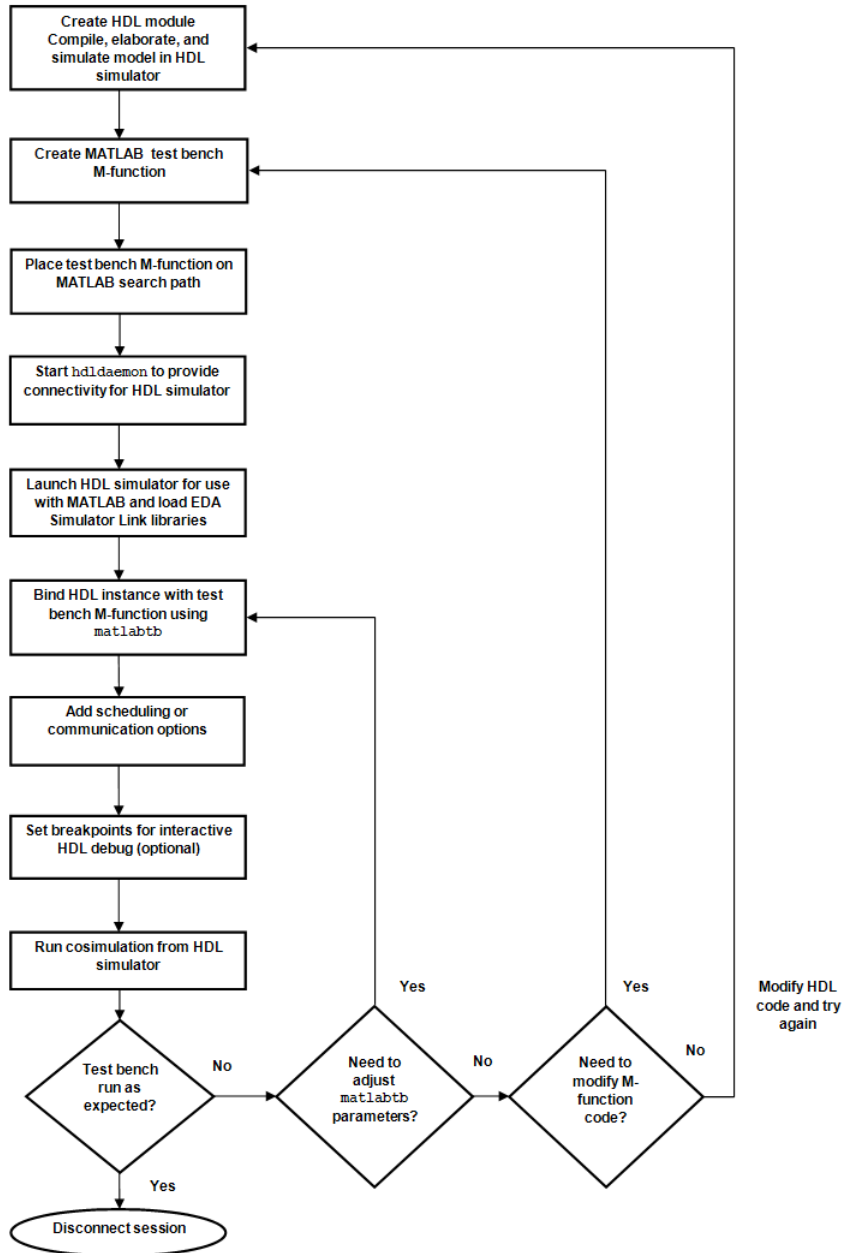
the server receives a request, it executes the specified MATLAB M-function you have coded to perform tasks on behalf of a module in your HDL design. Parameters that you specify when you start the server indicate whether the server establishes shared memory or TCP/IP socket communication links.

Refer to “Establishing EDA Simulator Link Machine Configuration Requirements” on page 8-12 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions and component functions (see Chapter 3, “Replacing an HDL Component with a MATLAB Component Function”) are virtually identical. Most of this chapter focuses on test bench functions, but in general all operations can be performed on and with both functions.

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

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



The workflow is as follows:

- 1** Create HDL module. Compile, elaborate, and simulate model in HDL simulator.
- 2** Create MATLAB test bench M-function.
- 3** Place test bench function on MATLAB search path.
- 4** Start hhdldaemon to provide connection to HDL simulator.
- 5** Launch HDL simulator for use with MATLAB and load EDA Simulator Link libraries.
- 6** Bind HDL instance with test bench function.
- 7** Add scheduling options.
- 8** Set breakpoints for interactive HDL debug (optional).
- 9** Run cosimulation from HDL simulator.
- 10** End cosimulation session.

Code HDL Modules for Verification Using MATLAB

In this section...

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

“Choosing an HDL Module Name” on page 2-7

“Specifying Port Direction Modes in HDL Components (MATLAB as Test Bench)” on page 2-8

“Specifying Port Data Types in HDL Components (MATLAB as Test Bench)” on page 2-8

“Compiling and Elaborating the HDL Design” on page 2-10

“Sample VHDL Entity Definition” on page 2-10

“Compiling and Debugging the HDL Model” on page 2-10

Overview to Coding HDL Modules for Use with MATLAB

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

Choosing an HDL Module Name

Although not required, when naming the HDL module, consider choosing a name that also can be used as a MATLAB M-function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, EDA Simulator Link software assumes that an HDL module and its simulation function share the same name. See “Invoke matlabtb to Bind MATLAB Test Bench Function Calls” on page 2-26.

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

Specifying Port Direction Modes in HDL Components (MATLAB as Test Bench)

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

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

Specifying Port Data Types in HDL Components (MATLAB as Test Bench)

This section describes how to specify data types compatible with MATLAB for ports in your HDL modules. For details on how the EDA Simulator Link interface converts data types for the MATLAB environment, see “Performing Data Type Conversions” on page 9-5.

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

Port Data Types for VHDL Entities

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

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

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

Note The EDA Simulator Link software does not support VHDL extended identifiers for the following components:

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

However, the software does support basic identifiers for VHDL.

Port Data Types for Verilog Modules

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

- reg
- integer
- wire

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

Compiling and Elaborating the HDL Design

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

Sample VHDL Entity Definition

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

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

Both input ports are defined as vectors consisting of five standard logic values. The output port `adj` is also defined as a standard logic vector, but consists of only two values. The output ports `dvalid` and `odata` are defined as scalar standard logic ports. For information on how the EDA Simulator Link interface converts data of standard logic scalar and array types for use in the MATLAB environment, see “Performing Data Type Conversions” on page 9-5.

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

Compiling and Debugging the HDL Model

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

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

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


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

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

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

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

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

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

Code an EDA Simulator Link MATLAB Test Bench Function

In this section...
“Process for Coding MATLAB EDA Simulator Link Functions” on page 2-12
“Syntax of a Test Bench Function” on page 2-13
“Sample MATLAB Test Bench Function” on page 2-13

Process for Coding MATLAB EDA Simulator Link Functions

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

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

- 1** Learn the syntax for a MATLAB EDA Simulator Link test bench function (see “Syntax of a Test Bench Function” on page 2-13).
- 2** Understand how EDA Simulator Link software converts data from the HDL simulator for use in the MATLAB environment (see “Performing Data Type Conversions” on page 9-5 in Chapter 9, “Advanced Operational Topics”).
- 3** Choose a name for the MATLAB function (see “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 2-28).
- 4** Define expected parameters in the function definition line (see “M-Function Syntax and Function Argument Definitions” on page 7-2).
- 5** Determine the types of port data being passed into the function (see Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).
- 6** Extract and, if appropriate for the simulation, apply information received in the `portinfo` structure (see “Gaining Access to and Applying Port

Information” on page 7-7 in the Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).

- 7 Convert data for manipulation in the MATLAB environment, as necessary (see “Converting HDL Data to Send to MATLAB” on page 9-5 in Chapter 9, “Advanced Operational Topics”).
- 8 Convert data that needs to be returned to the HDL simulator (see “Converting Data for Return to the HDL Simulator” on page 9-10 in Chapter 9, “Advanced Operational Topics”).

Syntax of a Test Bench Function

The syntax of a MATLAB test bench function is

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

See the Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters” for an explanation of each of the function arguments.

Sample MATLAB Test Bench Function

This section uses a sample MATLAB function to identify sections of a MATLAB test bench function required by the EDA Simulator Link software. You can see the full text of the code used in this sample in the section MATLAB Function Example: manchester_decoder.m on page 2-18.

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

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

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

1 Specify the MATLAB function name and required parameters.

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

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

See Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”.

The function declaration performs the following actions:

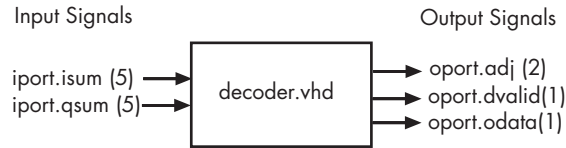
- Names the function. This declaration names the function `manchester_decoder`, which differs from the entity name `decoder`. Because the names differ, the function name must be specified explicitly later when the entity is initialized for verification with the `matlabtb` or `matlabtbeval` function. See “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 2-28.
- Defines required argument and return parameters. A MATLAB test bench function *must* return two parameters, `iport` and `tnext`, and pass three arguments, `oport`, `tnow`, and `portinfo`, and *must* appear in the order shown. See Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”.

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

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

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

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



For more information on the required MATLAB test bench function parameters, see Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”.

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

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

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

VHDL Example Port Definitions

Port	Direction	Type...	Converts to/Requires Conversion to...
isum	IN	STD_LOGIC_VECTOR(4 DOWNT0 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 DOWNT0 0)	A 5-bit column or row vector of characters where each bit maps to a standard logic character literal.

VHDL Example Port Definitions (Continued)

Port	Direction	Type...	Converts to/Requires Conversion to...
adj	OUT	STD_LOGIC_VECTOR(1 DOWNT0 0)	A 2-element column vector of characters. Each character matches a corresponding character literal that represents a logic state and maps to a single bit.
dvalid	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.
odata	OUT	STD_LOGIC	A character that matches the character literal representing the logic state.

For more information on interface data type conversions, see “Performing Data Type Conversions” on page 9-5 in Chapter 9, “Advanced Operational Topics”.

3 Set up any required timing parameters.

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

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

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

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

```
%% Compute one row and plot
isum = isum + 1;
adj(isum) = mv12dec(oport.adj');
data(isum) = mv12dec([oport.dvalid oport.odata]);
.
.
.
```

The two calls to `mv12dec` convert the binary data that the MATLAB function receives from the entity's output ports, `adj`, `dvalid`, and `odata` to unsigned decimal values that MATLAB can compute. The function converts the 2-bit transposed vector `oport.adj` to a decimal value in the range 0 to 4 and `oport.dvalid` and `oport.odata` to the decimal value 0 or 1.

Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters” provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the HDL simulator.

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

```
if isum == 17
    iport.isum = dec2mv1(isum,5);
    iport.qsum = dec2mv1(qsum,5);
else
    iport.isum = dec2mv1(isum,5);
end
```

The three calls to `dec2mv1` convert the decimal values computed by MATLAB to binary data that the MATLAB function can deposit to the entity's input ports, `isum` and `qsum`. In each case, the function converts a decimal value to 5-element bit vector with each bit representing a character that maps to a character literal representing a logic state.

“Converting Data for Return to the HDL Simulator” on page 9-10 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

MATLAB Function Example: manchester_decoder.m

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

% Copyright 2003-2009 The MathWorks, Inc.
% $Revision: 1.1.6.2 $ $Date: 2009/05/07 18:21:12 $

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
end

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

Place Test Bench Function on MATLAB Search Path

In this section...

“Use MATLAB which Function to Find Test Bench” on page 2-21

“Add Test Bench Function to MATLAB Search Path” on page 2-21

Use MATLAB which Function to Find Test Bench

The MATLAB M-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:

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

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

Add Test Bench Function to MATLAB Search Path

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

Start `hdldaemon` to Provide Connection to HDL Simulator

Start the MATLAB server as follows:

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

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

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

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

Note The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you initialize the HDL simulator for use with a MATLAB link session using the `matlabtb` or `matlabcp` function. In addition, if you specify TCP/IP socket mode, the socket port that you specify with `hdldaemon` and `matlabtb` or `matlabcp` must match. For more information on modes of communication, see “Specifying TCP/IP Socket Communication” on page 8-15.

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

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

Launch HDL Simulator for Use with MATLAB

In this section...
“Launching the HDL Simulator” on page 2-24
“Loading an HDL Design for Verification” on page 2-24

Launching the HDL Simulator

Starting the HDL Simulator from MATLAB

Start the HDL simulator directly from MATLAB by calling the MATLAB function `nclaunch`. See “Starting the HDL Simulator” on page 1-18 for instructions on using `nclaunch`.

Starting `ncsim` in the Terminal

If you would like to bring up `ncsim` in the terminal, instead of launching the Simvision GUI, perform the following steps:

- 1 Start `hdldaemon` in MATLAB.
- 2 Start an `xterm` from MATLAB in the background.
- 3 Run `ncsim` in the `xterm` shell, having it call back to the `hdlserver` to run your `matlabtb` function as usual.
- 4 Specify that the `matlabtb` function use the `touch` command on a file to signal completion while an M-script polls for completion.

The M-script can then change test parameters and run more tests.

Loading an HDL Design for Verification

After you start the HDL simulator from MATLAB with a call to `nclaunch`, load an instance of an HDL module for verification with the function `hdlsimmatlab`. At this point, it is assumed that you have coded and compiled your HDL model as explained in “Code HDL Modules for Verification Using MATLAB” on page 2-7. Issue the function `hdlsimmatlab` for each instance of an entity or module in your model that you want to cosimulate. For example:

```
hdlsimmatlab work.osc_top
```

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

Invoke `matlabtb` to Bind MATLAB Test Bench Function Calls

In this section...
“Invoking <code>matlabtb</code> ” on page 2-26
“Binding the HDL Module Component to the MATLAB Test Bench Function” on page 2-28

Invoking `matlabtb`

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

Be sure to follow the path specifications for MATLAB test bench sessions when invoking `matlabtb`, as explained in “Specifying HDL Signal/Port and Module Paths for MATLAB Test Bench Cosimulation” on page 2-26.

For instructions in issuing the `matlabtb` command, see “Running a Test Bench Cosimulation” on page 2-35.

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

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

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

In the following example:

```
matlabtb u_osc_filter -mfunc oscfilter
```

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

Path Specifications for MATLAB Link Sessions with Verilog Top Level.

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

The following examples show valid signal and module path specifications:

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

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

- top.sub/port_or_sig

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

- :sub:port_or_sig

:

:sub

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

Path Specifications for MATLAB Link Sessions with VHDL Top Level.

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

The following examples show valid signal and module path specifications:

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

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

- top.sub/port_or_sig

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

- :sub:port_or_sig

:

:sub

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

Binding the HDL Module Component to the MATLAB Test Bench Function

By default, the EDA Simulator Link software assumes that the name for a MATLAB M-function matches the name of the HDL module that the function verifies. When you create a test bench function that has a different name than the design under test, you must associate the design with the MATLAB test bench 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.

See the `-mfunc` argument of `matlabtb` for more information.

For a full list of `matlabtb` parameters, see the `matlabtb` 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 Function Call

In this example, you form an association between the `inverter_v1` component and the MATLAB component function `inverter_tb`. To do so, you invoke the function `matlabtb` with the `-mfunc` argument when you set up the simulation.

```
matlabtb inverter_v1 -mfunc inverter_tb
```

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

Schedule Options for a Test Bench Session

In this section...

“About Scheduling Options for Test Bench Sessions” on page 2-30

“Scheduling Test Bench Session Using `matlabtb` Arguments” on page 2-30

“Scheduling Test Bench M-Functions Using the `tnext` Parameter” on page 2-32

About Scheduling Options for Test Bench Sessions

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

- Using the arguments to the EDA Simulator Link function `matlabtb`
- Inside the MATLAB M-function using the `tnext` parameter

The two types of scheduling are not mutually exclusive. You can combine the `matlabtb` timing arguments and the `tnext` parameter of an M-function to schedule component session callbacks.

Note All scheduling options for test bench sessions (`matlabtb`) apply equally to component sessions (`matlabcp`).

Scheduling Test Bench Session Using `matlabtb` Arguments

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

You can schedule a MATLAB test bench function to execute using the `matlabtb` 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'->'1'.
 - Verilog: Rising edge is the transition from 0, x, or z to 1
- **Falling edge**—When a specified signal experiences a falling edge
 - VHDL: Falling edge is '1'->'0'. Values 'X', 'Z', 'H', and 'L' will be ignored.
 - Verilog: Falling edge is the transition from 1 to x, z or 0.
- **Signal state change**—When a specified signal changes state, based on a list using the -sensitivity argument to `matlabtb`.

Decide on a combination of options that best meet your test bench application requirements. For details on using the `matlabtb` arguments for scheduling, see the reference page for `matlabtb`.

Signals can be anywhere in the HDL hierarchy and you do not need to relate them to the HDL instance.

Examples of Scheduling Options with `matlabtb`

Use Discrete Times with -repeat Option. The following command instructs the HDL simulator to schedule the callback for 10 ns from the first call and repeat every 10 ns thereafter. The callbacks cease after 500 ns.

```
matlabtb lowpass_filter 10ns -repeat 10ns -cancel 500ns
```

Invoke `matlabtb` on Rising Edge. Consider the following `matlabtb` command:

```
ncsim> matlabtb test -rising /test/clock
-socket 4449
```

With this command, you link an instance of the HDL instance `test` with the function `test.m`. This function executes within the context of MATLAB based on specified timing parameters. In this case, the EDA Simulator Link

software calls the MATLAB function when the signal `/test/clock` experiences a rising edge.

Invoke Callback on Signal State Change. In this example, you sensitize the test bench to the signal `sine_out` and instruct the EDA Simulator Link software to call the test bench function each time the signal changes state by using the `-sensitivity` argument to `matlabtb`.

```
matlabtb osc_top -sensitivity /osc_top/sine_out
```

Scheduling Test Bench M-Functions Using the `tnext` Parameter

You can control the callback timing of a MATLAB test bench 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 M-function prototype, see Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”.

Example of Scheduling with `tnext`

In this example, each time the HDL simulator calls the test bench function (via EDA Simulator Link), `tnext` schedules the next callback to the MATLAB component 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
```

Run MATLAB Test Bench Simulation

In this section...

“Process for Running MATLAB Test Bench Cosimulation” on page 2-34

“Checking the MATLAB Server’s Link Status” on page 2-34

“Running a Test Bench Cosimulation” on page 2-35

“Applying Stimuli with the HDL Simulator force Command” on page 2-37

“Restarting a Test Bench Simulation” on page 2-38

Process for Running MATLAB Test Bench Cosimulation

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

- 1 Check the MATLAB server’s link status.
- 2 Run and monitor the test bench session.
- 3 Apply test bench stimuli.
- 4 Restart simulator during a test bench session (if necessary).

Checking the MATLAB Server’s Link Status

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

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

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

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


If the server is not running, the message reads

```
HDLDaemon is NOT running
```

See "Link Status" in the `hdldaemon` reference documentation for information on determining the mode of communication and the TCP/IP socket in use.

Running a Test Bench Cosimulation

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

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

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

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

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

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

- The mode and, if appropriate, TCP/IP data necessary for connecting to a MATLAB server (see `matlabtb` reference)
- The MATLAB function that is associated with and executes on behalf of the HDL instance (see “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 2-28)
- 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 2-30)

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 M-code.

6 Resume the simulation, as needed.

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

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

The following HDL simulator command resumes a simulation:

```
run -continue
```

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

Applying Stimuli with the HDL Simulator force Command

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

Another option is to issue force commands in the HDL simulator main window.

For example, consider the following sequence of force commands:

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

These commands drive the following signals:

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



For example,

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

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

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

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

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

All three approaches may lead to performance degradation.

Restarting a Test Bench Simulation

Because the HDL simulator issues the service requests during a MATLAB test bench session, you must restart a test bench 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` command.

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

Stop Test Bench Simulation

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

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

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

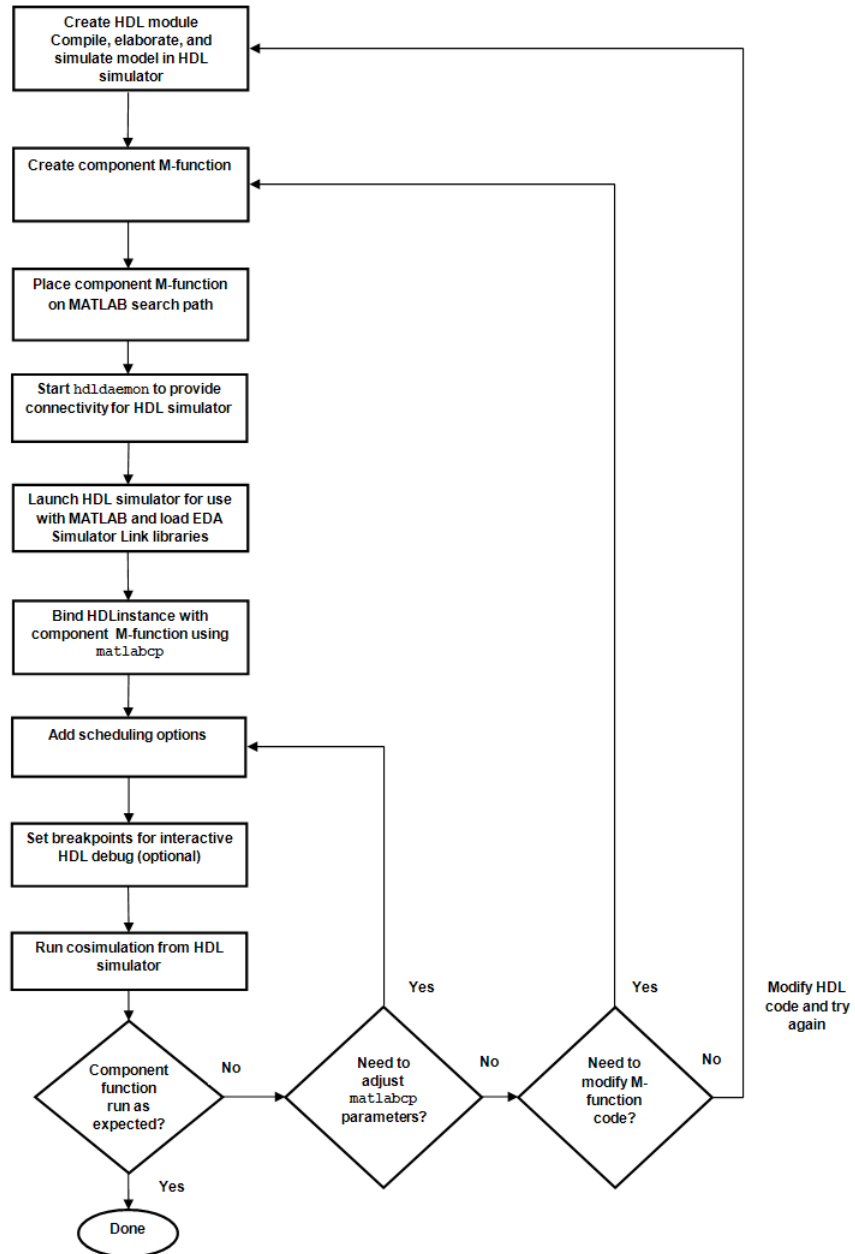
For more information on closing HDL simulator sessions, see the HDL simulator documentation.

Replacing an HDL Component with a MATLAB Component Function

- “Workflow for Creating a MATLAB Component Function for Use with the HDL Simulator” on page 3-2
- “Overview to Using a MATLAB Function as a Component” on page 3-5
- “Code HDL Modules for Visualization Using MATLAB” on page 3-7
- “Code an EDA Simulator Link MATLAB Component Function” on page 3-8
- “Place Component Function on MATLAB Search Path” on page 3-10
- “Start hdd daemon to Provide Connection to HDL Simulator” on page 3-11
- “Start HDL Simulator for Use with MATLAB” on page 3-12
- “Invoke matlabcp to Bind MATLAB Component Function Calls” on page 3-13
- “Schedule Options for a Component Session” on page 3-14
- “Run MATLAB Component Simulation” on page 3-17

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

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



The workflow is as follows:

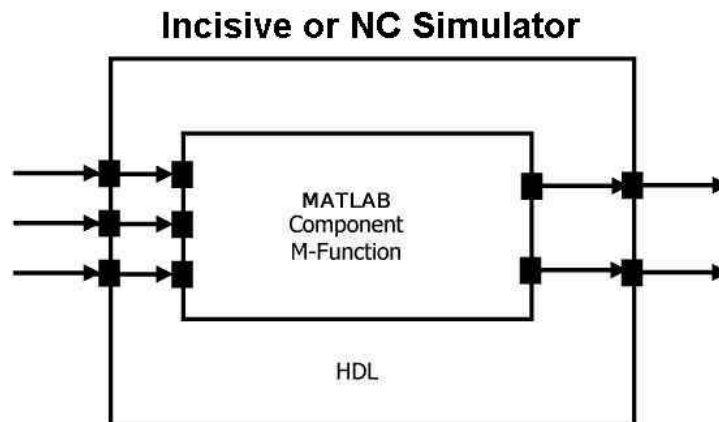
- 1** Create HDL module. Compile, elaborate, and simulate model in HDL simulator.
- 2** Create MATLAB component function.
- 3** Place component function on MATLAB search path.
- 4** Start hhdldaemon to provide connection to HDL simulator.
- 5** Launch HDL simulator for use with MATLAB and load EDA Simulator Link libraries.
- 6** Bind HDL instance with component function.
- 7** Add scheduling options.
- 8** Set breakpoints for interactive HDL debug (optional).
- 9** Run cosimulation from HDL simulator.

Overview to Using a MATLAB Function as a Component

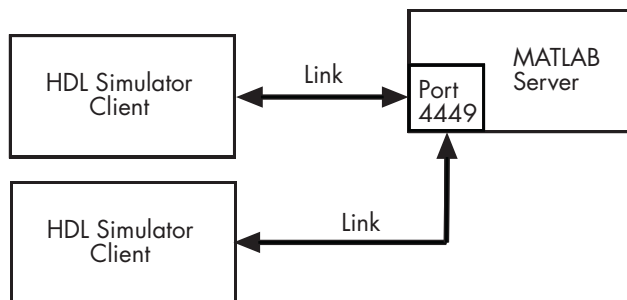
The EDA Simulator Link software provides a means for visualizing HDL components within the MATLAB environment. You do so by coding an HDL model and a MATLAB M-function that can share data with the HDL model. This chapter discusses the programming, interfacing, and scheduling conventions for MATLAB component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

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

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



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



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

Refer to “Establishing EDA Simulator Link Machine Configuration Requirements” on page 8-12 for valid machine configurations.

Note The programming, interfacing, and scheduling conventions for test bench functions (see) and component functions are virtually identical. Steps in the component workflow actually map to documentation for test bench functions; however, the same procedures apply to component functions.

Code HDL Modules for Visualization Using MATLAB

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

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

See “Code HDL Modules for Verification Using MATLAB ” on page 2-7 for full instructions on following this process.

Code an EDA Simulator Link MATLAB Component Function

In this section...
“Overview to Coding an EDA Simulator Link Component Function” on page 3-8
“Syntax of a Component Function” on page 3-9

Overview to Coding an EDA Simulator Link Component Function

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

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

- 1** Learn the syntax for a MATLAB EDA Simulator Link component function (see “Syntax of a Component Function” on page 3-9.).
- 2** Understand how EDA Simulator Link software converts data from the HDL simulator for use in the MATLAB environment (see “Performing Data Type Conversions” on page 9-5 in Chapter 9, “Advanced Operational Topics”).
- 3** Choose a name for the MATLAB component function (see “Invoke matlabcp to Bind MATLAB Component Function Calls” on page 3-13).
- 4** Define expected parameters in the component function definition line (see Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).
- 5** Determine the types of port data being passed into the function (see Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).
- 6** Extract and, if appropriate for the simulation, apply information received in the `portinfo` structure (see “Gaining Access to and Applying Port

Information” on page 7-7 in the Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).

- 7 Convert data for manipulation in the MATLAB environment, as necessary (see “Converting HDL Data to Send to MATLAB” on page 9-5 in Chapter 9, “Advanced Operational Topics”).
- 8 Convert data that needs to be returned to the HDL simulator (see “Converting Data for Return to the HDL Simulator” on page 9-10 in Chapter 9, “Advanced Operational Topics”).

Syntax of a Component Function

The syntax of a MATLAB component function is

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

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

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

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

See the Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters” for an explanation of each of the function arguments. For more information on using `tnext` and `tnow` for simulation scheduling with `matlabtb` (it is the same for `matlabcp`), see “Scheduling Test Bench M-Functions Using the `tnext` Parameter” on page 2-32.

Place Component Function on MATLAB Search Path

The MATLAB M-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). See “Place Test Bench Function on MATLAB Search Path” on page 2-21 for instructions on performing this process.

Start hlldaemon to Provide Connection to HDL Simulator

See “Start hlldaemon to Provide Connection to HDL Simulator” on page 2-22 for instructions on starting hlldaemon.

Start HDL Simulator for Use with MATLAB

See “Launch HDL Simulator for Use with MATLAB” on page 2-24 for instructions on starting the HDL simulator from MATLAB.

Refer to “Loading an HDL Design for Verification” on page 2-24 for instructions on loading an instance of an HDL module for visualization with the function `hdlsimmatlab`.

Invoke matlabcp to Bind MATLAB Component Function Calls

In this section...

“About Binding Component Function Calls” on page 3-13

“Example of Binding Component Function Calls” on page 3-13

About Binding Component Function Calls

When you create a component function that has a different name than the design under test, you must associate the design with the MATLAB component function using the `-mfunc` argument to `matlabcp`. The procedure for binding component function calls is exactly the same as the procedure for binding test bench function calls. See “Binding the HDL Module Component to the MATLAB Test Bench Function” on page 2-28 for instructions on binding a test bench function to an HDL design, and follow the same instructions for use with a component session.

See “Invoke matlabtb to Bind MATLAB Test Bench Function Calls” on page 2-26 for instructions on invoking `matlabtb`. Follow the same instructions for use with `matlabcp`.

You can find an example binding a component session in the next section.

For examples of invoking and binding test bench sessions, see “Example of Binding Test Bench Function Call” on page 2-29.

Example of Binding Component Function Calls

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

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

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

Schedule Options for a Component Session

In this section...
“About Scheduling Options for Component Sessions” on page 3-14
“Scheduling Component Session Using matlabcp Arguments” on page 3-15
“Scheduling Test Bench M-Functions Using the tnext Parameter” on page 3-16

About Scheduling Options for Component Sessions

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

- Using the scheduling arguments of the EDA Simulator Link function `matlabcp`
- Inside the MATLAB M-function using the `tnext` parameter

You schedule a component session (`matlabcp`) the same way you schedule a test bench session (`matlabtb`).

The two types of scheduling are not mutually exclusive. The `Oscfilter` demo shows a component function that uses both the `matlabcp` timing arguments and the `tnext` parameter of an M-function to schedule component session callbacks.

You can find examples of both types of scheduling component sessions in the next sections:

- “Scheduling Component Session Using `matlabcp` Arguments” on page 3-15
- “Scheduling Test Bench M-Functions Using the `tnext` Parameter” on page 3-16

For examples of scheduling test bench sessions, see “Examples of Scheduling Options with `matlabtb`” on page 2-31 and “Example of Scheduling with `tnext`” on page 2-33.

Scheduling Component Session Using matlabcp Arguments

See “Scheduling Test Bench Session Using matlabb Arguments” on page 2-30 for instructions on scheduling test bench function calls using matlabb and follow the same procedure for scheduling component sessions using matlabcp.

Examples of Scheduling Component Sessions Using matlabcp Arguments

Call Back Function on Rising Edge. Call back function on the rising edge of signal clk1:

```
matlabb top -rising clk1 -mfunc matlabb
```

Call Back Function on Falling Edge. Call back function on the falling edge of signal clk1:

```
matlabb top -falling clk1 -mfunc matlabb
```

Call back function on the falling edge of signal Signal_1:

```
matlabb top falling Signal_1 -mfunc matlabb
```

Schedule Callbacks Using Explicit Times. Schedule callbacks on the following explicit times:

```
matlabb top 15 16 17 Signal_1 -mfunc matlabb
```

Schedule callbacks on the following explicit times:

```
matlabb top 10e6 fs 20e3 ps 21 ns Signal_1 -mfunc matlabb
```

Schedule callbacks on the following explicit times and repeat those times every 4ns:

```
matlabb vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 4 ns -mfunc matlabb
```

Schedule Callbacks on Signal State Change. Schedule callbacks sensitive to signal outclk1:

```
matlabb vlogtestbench_top -sensitivity vlogtestbench_top.outclk1 -mfunc matlabb
```

Scheduling Test Bench M-Functions Using the `tnext` Parameter

See “Scheduling Test Bench M-Functions Using the `tnext` Parameter” on page 2-32 for instructions on scheduling test function calls using the `tnext` parameter of an M-function and follow the same procedure for scheduling component sessions using `tnext`.

Example of Scheduling Component Sessions Using `tnext`

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

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

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

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

See Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters” for more about `tnext` and `tnow`.

Run MATLAB Component Simulation

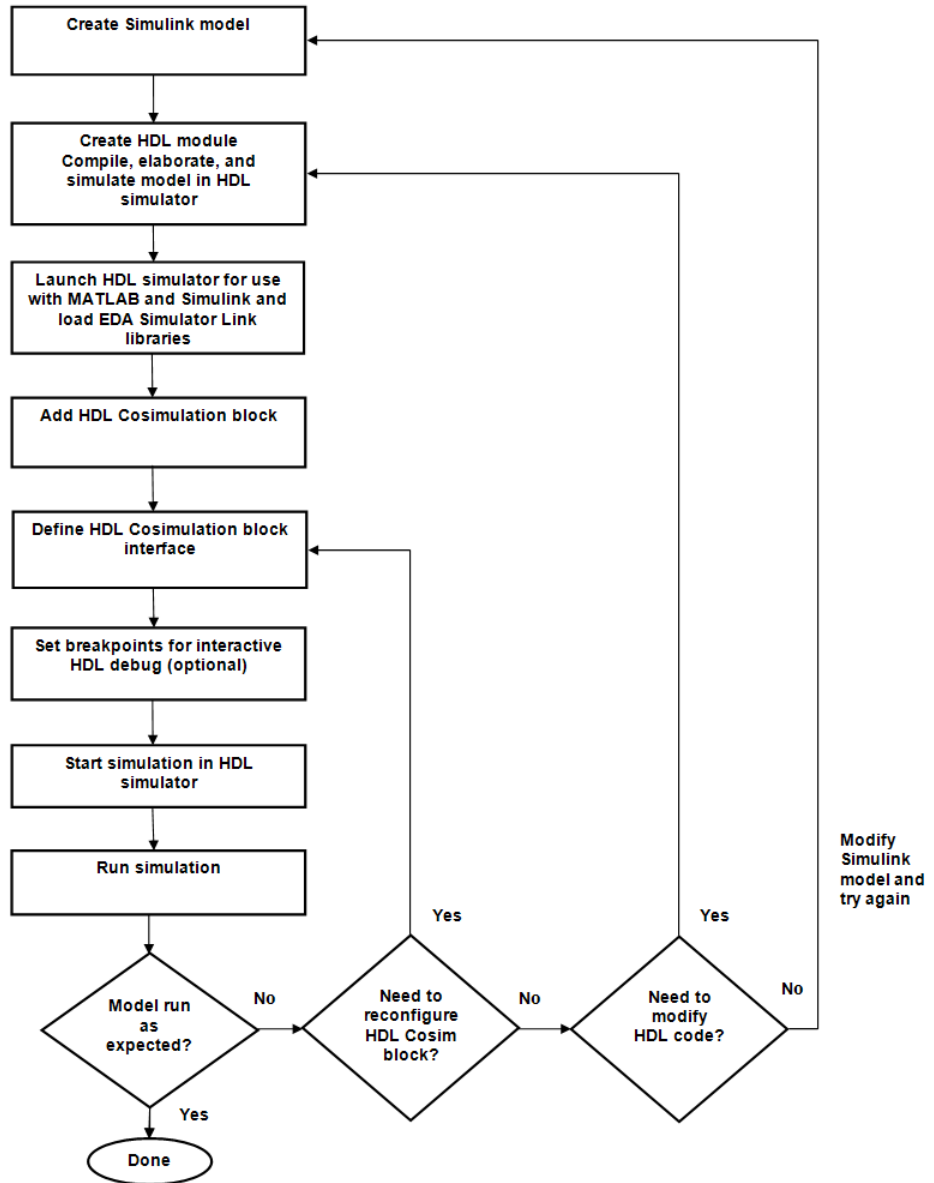
Running a MATLAB component session is the same as running one for a test bench simulation. See “Run MATLAB Test Bench Simulation” on page 2-34 for instructions on running a MATLAB test bench cosimulation session with the HDL simulator, and follow the same instructions for running a component session.

Simulating an HDL Component in a Simulink Test Bench Environment

- “Workflow for Simulating an HDL Component in a Simulink Test Bench Environment” on page 4-2
- “Overview to Using Simulink as a Test Bench” on page 4-5
- “Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator” on page 4-10
- “Code an HDL Component for Use with Simulink Test Bench Applications” on page 4-11
- “Launch HDL Simulator for Test Bench Cosimulation with Simulink” on page 4-12
- “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 4-13
- “Define the HDL Cosimulation Block Interface (Simulink as Test Bench)” on page 4-14
- “Start the HDL Simulation” on page 4-40
- “Run a Test Bench Cosimulation Session (Simulink as Test Bench)” on page 4-41

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

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



The workflow is as follows:

- 1** Create Simulink model.
- 2** Create HDL module. Compile, elaborate, and simulate model in HDL simulator. “Code HDL Modules for Verification Using MATLAB ” on page 2-7
- 3** Launch HDL simulator for use with MATLAB and load EDA Simulator Link libraries.
- 4** Add HDL Cosimulation block.
- 5** Define HDL Cosimulation block interfaces.
- 6** Set breakpoints for interactive HDL debug (optional).
- 7** Run cosimulation.

Overview to Using Simulink as a Test Bench

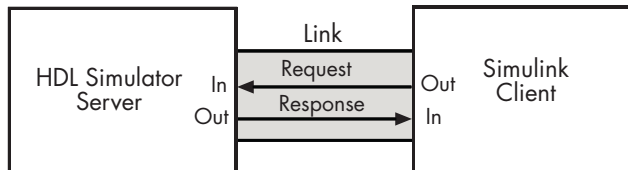
In this section...

“Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link” on page 4-5

“Introduction to the EDA Simulator Link HDL Cosimulation Block” on page 4-8

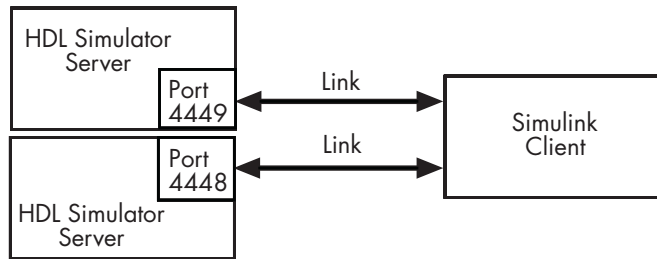
Understanding How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link

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



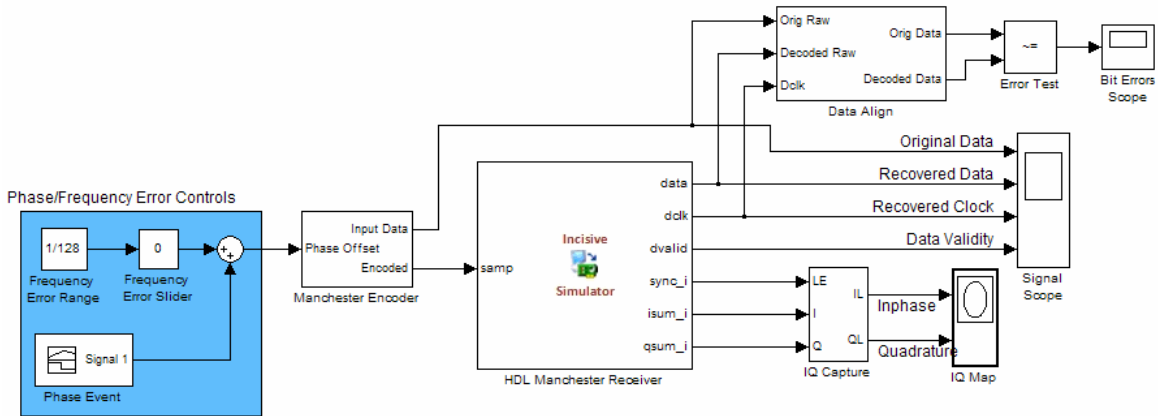
In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For example, you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

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



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

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



Running a Cosimulation

1. Click the command to the right to compile the HDL and launch the HDL simulator.
2. Perform any debug setup in the HDL simulator such as setting breakpoints or creating signal probes.
3. Start the cosimulation in Simulink by choosing either 'Simulation/Start' or 'Tools/Simulink Debugger'.

```
nclaunch( ...
'rundir', 'TEMPDIR', ...
'tclstart', { ...
[exec ncvtlog -linedebug 'vlogFiles{;}], ...
'exec ncelab -access +rwc manchester', ...
'hdlstimulink-gui manchester' ...
} ...
);
```

Copyright 2006-2009 The MathWorks, Inc.

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

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

- Discrete-Time Scatter Plot Scope block from the Communications Blockset software

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

Introduction to the EDA Simulator Link HDL Cosimulation Block

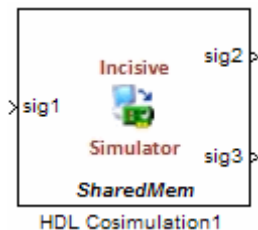
The EDA Simulator Link HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

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

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

The block mask contains panels for entering port and signal information, setting communication modes, adding clocks, specifying pre- and post-simulation Tcl commands, 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. This block, located in the Simulink Library, within the EDA Simulator Link block library, 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. See “Ports Pane” on page 11-4 in the Chapter 11, “Blocks — Alphabetical List”.
- **Connection Pane:** Type of communication and related settings to be used for exchanging data between simulators. See “Connection Pane” on page 11-10 in the Chapter 11, “Blocks — Alphabetical List”.
- **Timescales Pane:** The timing relationship between Simulink software and the HDL simulator. See “Timescales Pane” on page 11-14 in the Chapter 11, “Blocks — Alphabetical List”.
- **Clocks Pane:** Optional rising-edge and falling-edge clocks to apply to your model. See “Clocks Pane” on page 11-18 in the Chapter 11, “Blocks — Alphabetical List”.
- **Tcl Pane:** Tcl commands to run before and after a simulation. See “Tcl Pane” on page 11-21 in the Chapter 11, “Blocks — Alphabetical List”.

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

In this section...
“Creating Your Simulink Model” on page 4-10
“Running and Testing a Hardware Model in Simulink” on page 4-10
“Adding a Value Change Dump (VCD) File (Optional)” on page 4-10

Creating Your Simulink Model

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

Running and Testing a Hardware Model in Simulink

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

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

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

Code an HDL Component for Use with Simulink Test Bench Applications

In this section...

“Overview to Coding HDL Components for Use with Simulink” on page 4-11

“Specifying Port Direction Modes in the HDL Component” on page 4-11

“Specifying Port Data Types in the HDL Component” on page 4-11

Overview to Coding HDL Components for Use with Simulink

The most basic element of communication in the EDA Simulator Link interface is the HDL module. The interface passes all data between the HDL simulator and Simulink as port data. The EDA Simulator Link software works with any existing HDL module. However, when you code an HDL module that is targeted for Simulink verification, you should consider the types of data to be shared between the two environments and the direction modes. These factors are also ones you need to consider when you code an HDL component for use with Simulink as a test bench.

Specifying Port Direction Modes in the HDL Component

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). See “Specifying Port Direction Modes in HDL Components (MATLAB as Test Bench)” on page 2-8 for descriptions of these port directions.

Specifying Port Data Types in the HDL Component

You must specify data types compatible with Simulink for ports in your HDL modules. See “Specifying Port Data Types in HDL Components (MATLAB as Test Bench)” on page 2-8 for descriptions of supported port data types.

Launch HDL Simulator for Test Bench Cosimulation with Simulink

In this section...

“Starting the HDL Simulator from MATLAB” on page 4-12

“Loading an Instance of an HDL Module for Cosimulation” on page 4-12

Starting the HDL Simulator from MATLAB

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

If both tools are running on the same system, start the HDL simulator directly from MATLAB by calling the MATLAB function `nclaunch`. Alternatively, you can start the HDL simulator manually and load the EDA Simulator Link libraries yourself. Either way, see “Starting the HDL Simulator” on page 1-18.

Loading an Instance of an HDL Module for Cosimulation

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

For example:

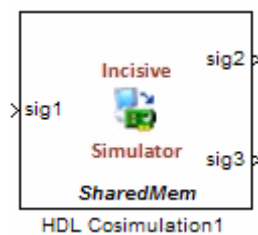
```
hdlsimulink work.manchester
```

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

Add the HDL Cosimulation Block to the Simulink Test Bench Model

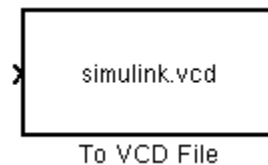
After you code one of your model's components in VHDL or Verilog and simulate it in the HDL simulator environment, integrate the HDL representation into your Simulink model as an HDL Cosimulation block by performing the following steps:

- 1 Open your Simulink model, if it is not already open.
- 2 Delete the model component that the HDL Cosimulation block is to replace.
- 3 In the Simulink Library Browser, click the EDA Simulator Link block library. The browser displays the HDL Cosimulation and To VCD block icons.



HDL
Cosimulation

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



To VCD File

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

- 4 Copy the HDL Cosimulation block icon from the Library Browser to your model. Simulink creates a link to the block at the point where you drop the block icon.
- 5 Connect any HDL Cosimulation block ports to appropriate blocks in your Simulink model.
 - To model a sink device, configure the block with inputs only.
 - To model a source device, configure the block with outputs only.

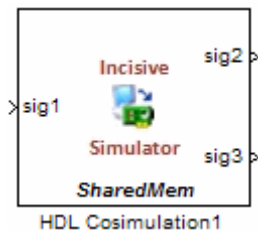
Define the HDL Cosimulation Block Interface (Simulink as Test Bench)

In this section...

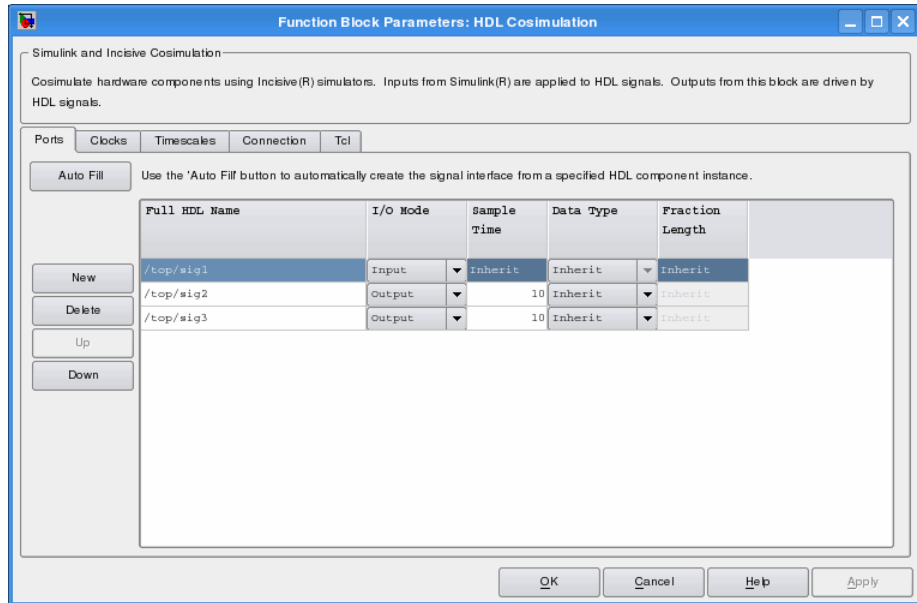
- “Accessing the HDL Cosimulation Block Interface” on page 4-14
- “Mapping HDL Signals to Block Ports” on page 4-15
- “Specifying the Signal Data Types” on page 4-27
- “Configuring the Simulink and HDL Simulator Timing Relationship” on page 4-27
- “Configuring the Communication Link in the HDL Cosimulation Block” on page 4-34
- “Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box” on page 4-36
- “Programmatically Controlling the Block Parameters” on page 4-37

Accessing the HDL Cosimulation Block Interface

To open the block parameters dialog box for the HDL Cosimulation block, double-click the block icon.



Simulink displays the following Block Parameters dialog box.



Mapping HDL Signals to Block Ports

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

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

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

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box (see “Entering Signal Information Manually” on page 4-24). This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.
- Use the **Auto Fill** button to obtain signal information automatically by transmitting a query to the HDL simulator. This approach can save

significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query. See “Obtaining Signal Information Automatically from the HDL Simulator” on page 4-18 for details.

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see product documentation for details. This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Specifying HDL Signal/Port and Module Paths for Cosimulation

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

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

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

- “Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level” on page 4-16
- “Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level” on page 4-17

Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level.

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

The following examples show valid signal and module path specifications:


```

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

```

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

- top.sub/port_or_sig

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

- :sub:port_or_sig

```

:
:sub

```

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

Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level.

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

The following examples show valid signal and module path specifications:

```

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

```

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

- top.sub/port_or_sig

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

- :sub:port_or_sig

:

:sub

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

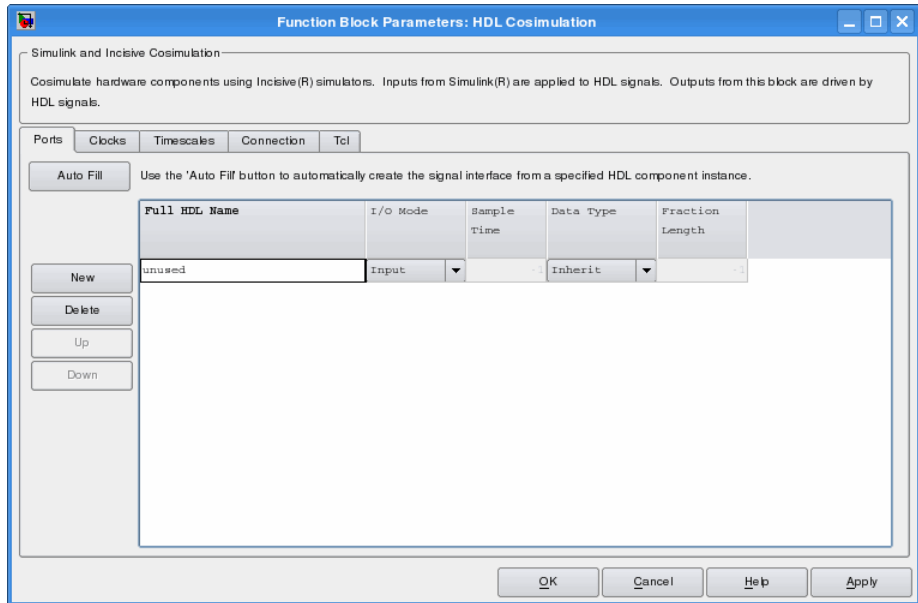
Obtaining Signal Information Automatically from the HDL Simulator

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

The following example describes the required steps.

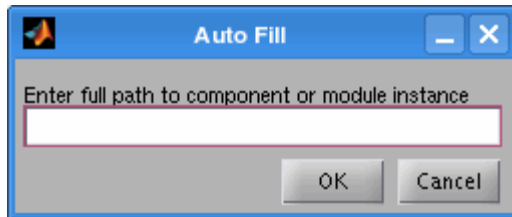
The example is based on a modified copy of the Manchester Receiver model, in which all signals were first deleted from the **Ports** and **Clocks** panes.

- 1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens.



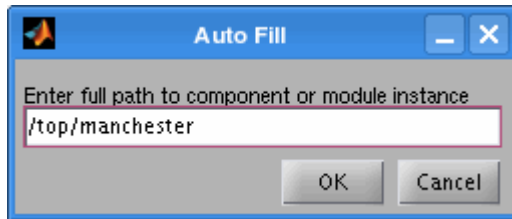
Tip Delete all ports before performing **Auto Fill** to ensure that no unused signal remains in the Ports list at any time.

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

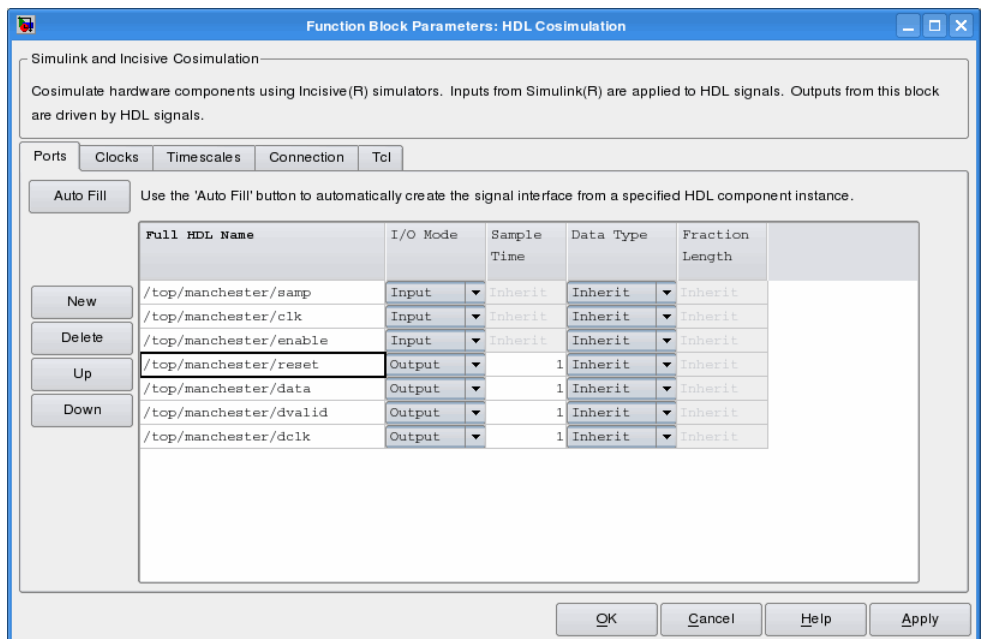


This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. The path you enter is not a file path and has nothing to do with the source files.

- 3 In this example, the Auto Fill feature obtains port data for a VHDL component called manchester. The HDL path is specified as /top/manchester.

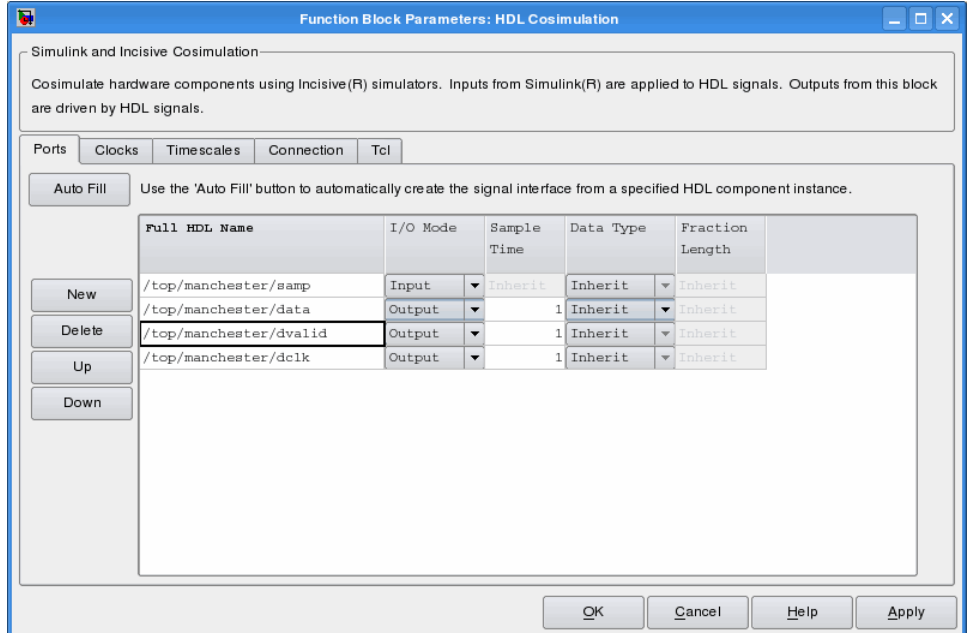


- 4 Click **OK** to dismiss the dialog box and the query is transmitted.
- 5 After the HDL simulator returns the port data, the Auto Fill feature enters it into the **Ports** pane, as shown in the following figure.

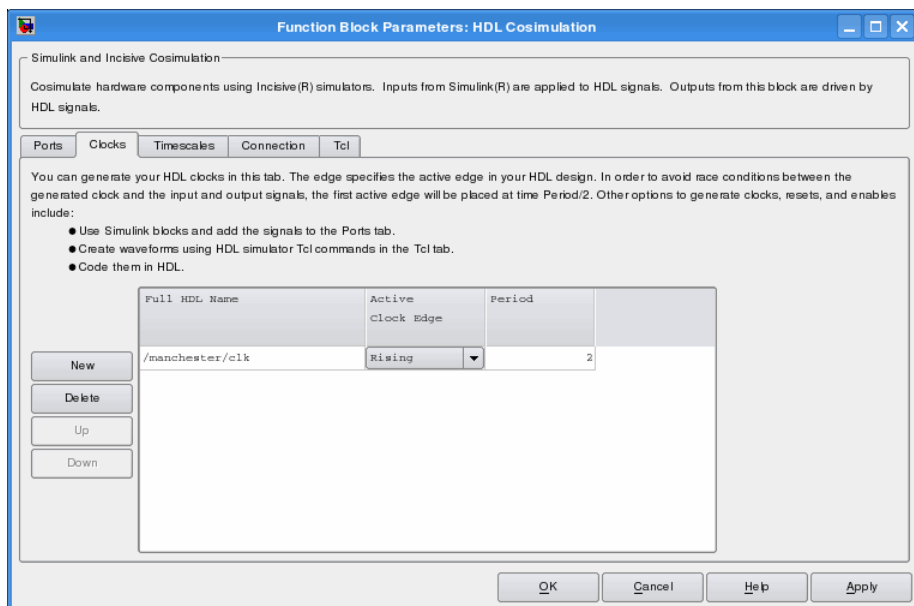


- 6 Click **Apply** to commit the port additions.

The preceding figure shows that the query entered clock, clock enable, and reset ports (labeled `clk`, `enable`, and `reset` respectively) into the ports list. In this example, the `clk` signal is entered in the **Clocks** pane, and the `enable` and `reset` signals are deleted from the **Ports** pane, as the next figures show.



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

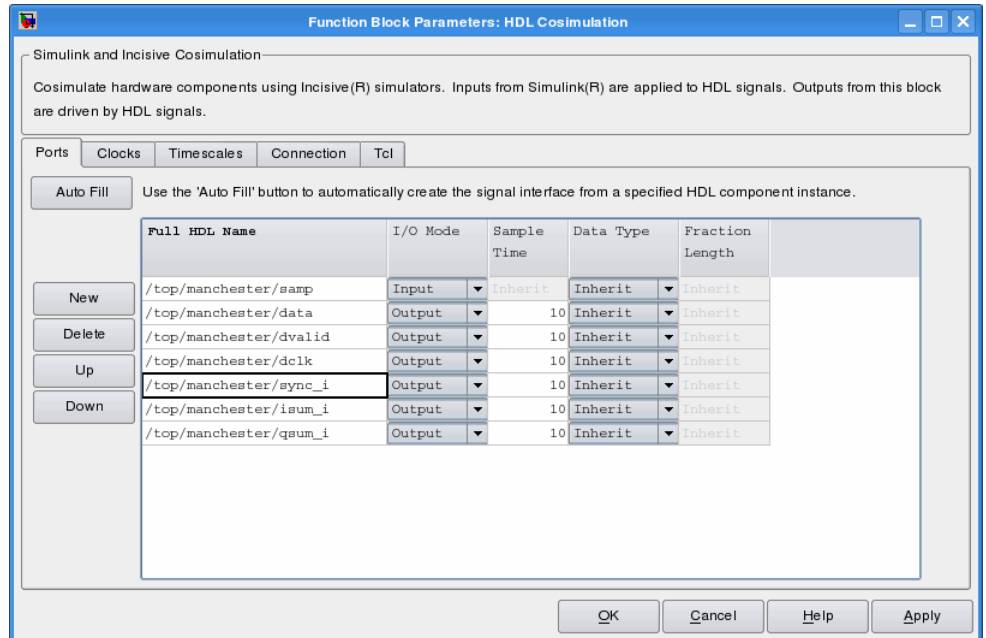


7 Auto Fill returns default values for output ports:

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

You may need to change these values as required by your model. In this example, the **Sample time** should be set to 10 for all outputs. See also “Specifying the Signal Data Types” on page 4-27.

8 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.

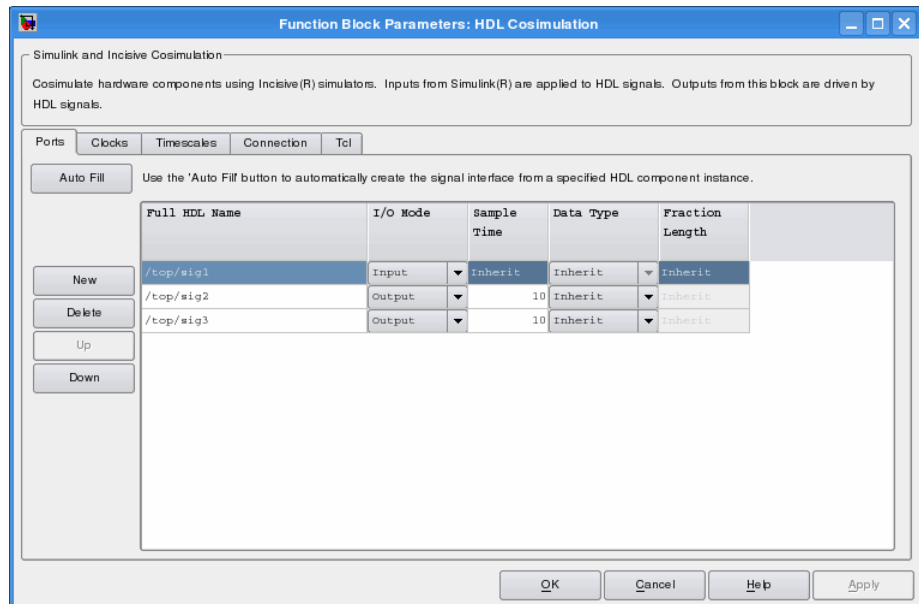
Note that `clk`, `reset`, and `clk_enable` *may* be in the Clocks and Tcl panes but they don't *have* to be. These signals can be ports if you choose to drive them explicitly from Simulink.

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

Entering Signal Information Manually

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

- 1 In the HDL simulator, determine the signal path names for the HDL signals you plan to define in your block.
- 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.



In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for

output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

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

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

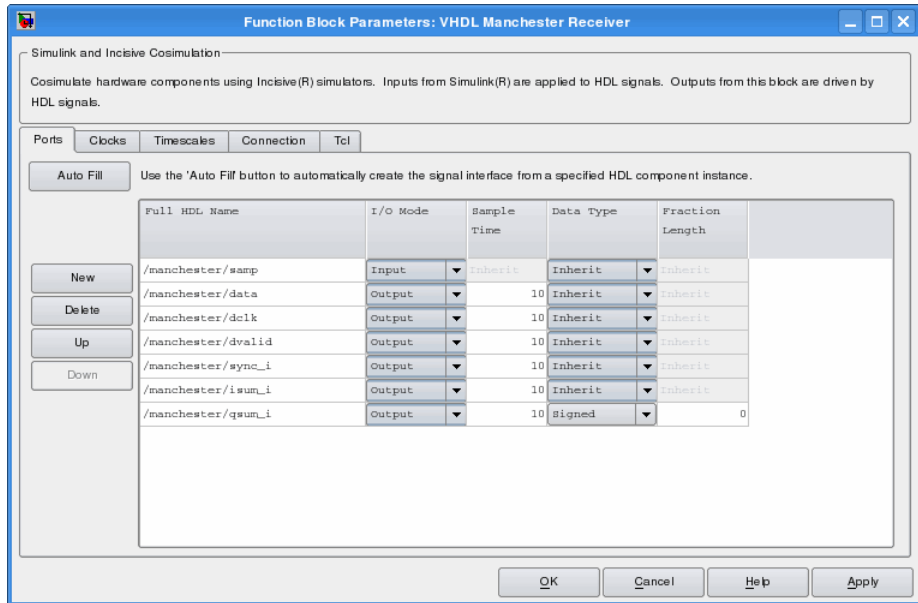
- For a sink device: specify block output ports.
- For a source device: specify block input ports.

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

- Use HDL simulator path name syntax (see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-16).
- 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.



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

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

For more information on sample times in the EDA Simulator Link cosimulation environment, see “Understanding the Representation of Simulation Time” on page 9-15.

- 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.

Specifying the Signal Data Types

The **Data Type** and **Fraction Length** parameters apply only to output signals. See **Data Type** and **Fraction Length** on the Ports pane description of the HDL Cosimulation block.

Configuring the Simulink and HDL Simulator Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog box. Before setting the **Timescales** parameters, you should read “Understanding the Representation of Simulation Time” on page 9-15 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, as described in “Timescales Pane” on page 11-14 of the HDL Cosimulation block reference.

Defining the Simulink and HDL Simulator Timing Relationship

The differences in the representation of simulation time can be reconciled in one of two ways using the EDA Simulator Link interface:

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

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

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

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 9-18.
- 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 9-20.

For more on relative and absolute time, see “Understanding the Representation of Simulation Time” on page 9-15.

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

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

Driving Clocks, Resets, and Enables

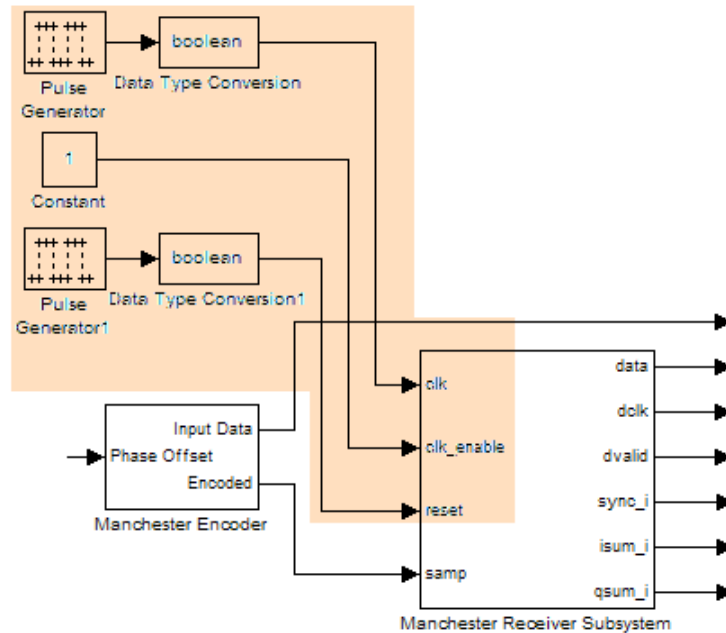
You can create rising-edge or falling-edge clocks, resets, or clock enable signals that apply internal stimuli to your model under cosimulation. You can add these signals in the following ways:

- By “Adding Signals Using Simulink Blocks” on page 4-29

- By “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 4-30
- By “Driving Signals by Adding Force commands” on page 4-33
- By implementing these signals directly in HDL code. If your model is part of a much larger HDL design, you (or the larger model designer) may choose to implement these signals in the Verilog or VHDL files. However, that implementation exceeds the scope of this documentation; see an HDL reference for more information.

Adding Signals Using Simulink Blocks. Add rising-edge or falling-edge clocks, resets, or clock enable signals to your Simulink model using Simulink blocks. See the Simulink User Guide and Reference for instructions on adding Simulink blocks to a Simulink model.

In the following example excerpt, the shaded area shows a clock, a reset, and a clock enable signal as input to a multiple HDL Cosimulation block model. These signals are created using two Simulink data type conversion blocks and a constant source block, which connect to the HDL Cosimulation block labeled "Manchester Receiver Subsystem".



Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block. When you specify a clock in your block definition, Simulink creates a rising-edge or falling-edge clock that drives the specified HDL signal.

Simulink attempts to create a clock that has a 50% duty cycle and a predefined phase that is inverted for the falling edge case. If necessary, Simulink degrades the duty cycle to accommodate odd Simulink sample times, with a worst case duty cycle of 66% for a sample time of $T=3$.

Whether you have configured the **Timescales** pane for relative timing mode or absolute timing mode, the following restrictions apply to clock periods:

- If you specify an explicit clock period, you must enter a sample time equal to or greater than 2 resolution units (ticks).

- If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle, and therefore the EDA Simulator Link software creates the falling edge at

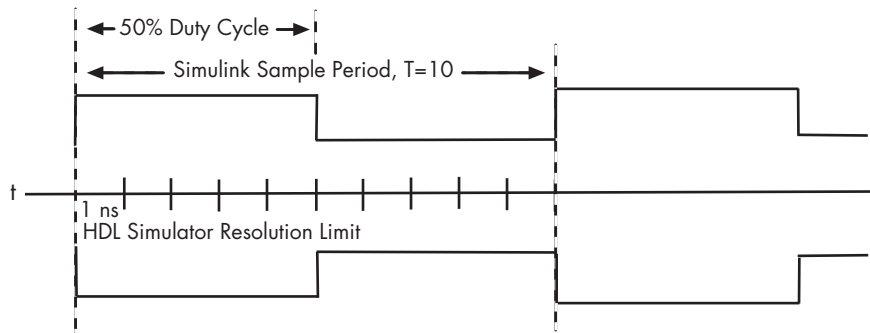
$$\text{clockperiod} / 2$$

(rounded down to the nearest integer).

For more information on calculating relative and absolute timing modes, see “Defining the Simulink and HDL Simulator Timing Relationship” on page 9-16.

The following figure shows a timing diagram that includes rising and falling edge clocks with a Simulink sample time of $T=10$ and an HDL simulator resolution limit of 1 ns. The figure also shows that given those timing parameters, the clock duty cycle is 50%.

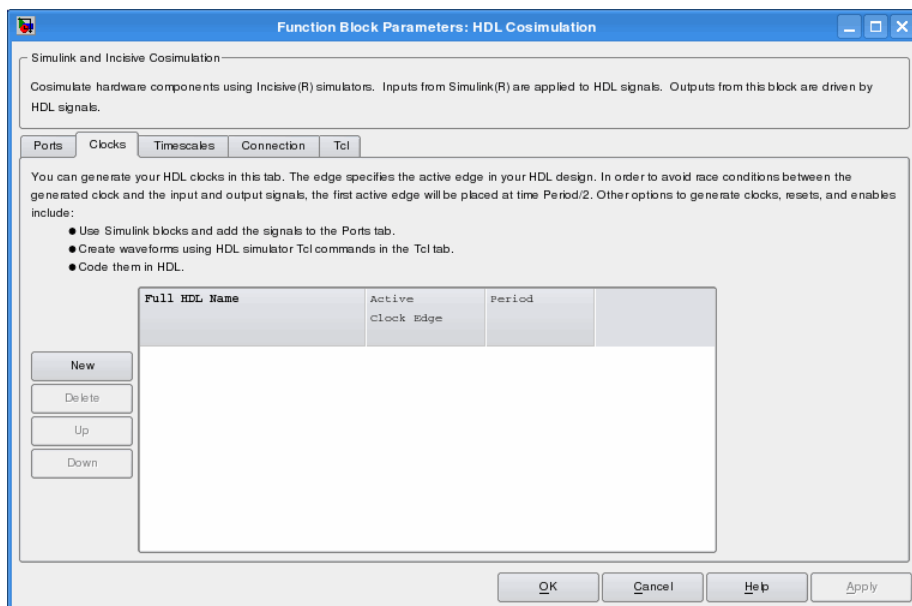
Rising Edge Clock



Falling Edge Clock

To create clocks, perform the following steps:

- 1 In the HDL simulator, determine the clock signal path names you plan to define in your block. To do so, you can use the same method explained for determining the signal path names for ports in step 1 of “Mapping HDL Signals to Block Ports” on page 4-15.
- 2 Select the **Clocks** tab of the Block Parameters dialog box. Simulink displays the dialog box as shown in the next figure.



3 Click **New** to add a new clock signal.

4 Edit the clock signal path name directly in the table under the **Full HDL Name** column by double-clicking the default clock signal name (`/top/c1k`). Then, specify your new clock using HDL simulator path name syntax. See “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-16.

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

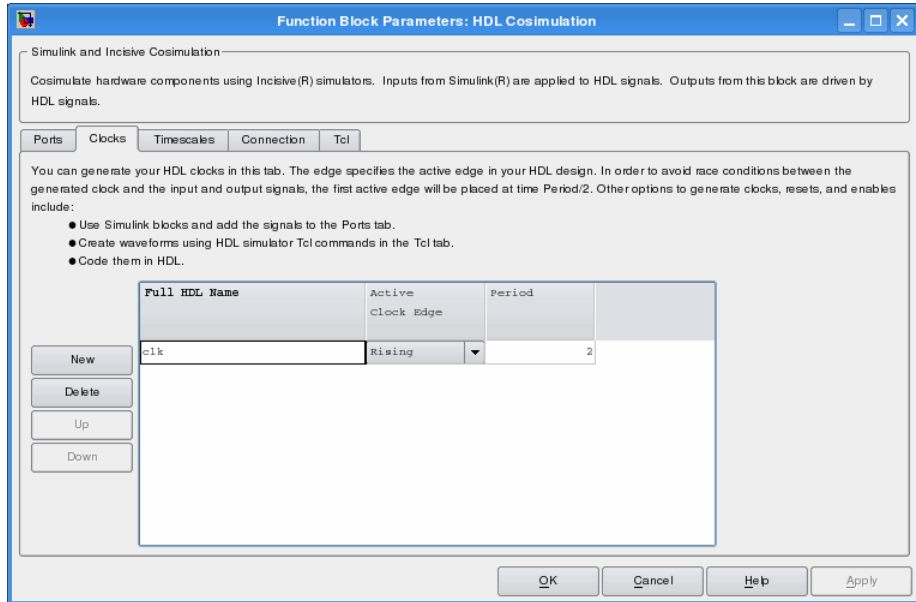
5 To specify whether the clock generates a rising-edge or falling edge signal, select **Rising** or **Falling** from the **Active Clock Edge** list.

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.



Driving Signals by Adding Force commands. Drive clocks, resets, and enable signals by adding force commands to the **Tcl** pane.

For example:

```
@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
```

You can also drive signals with `nclaunch` and the `force` command.

For example:

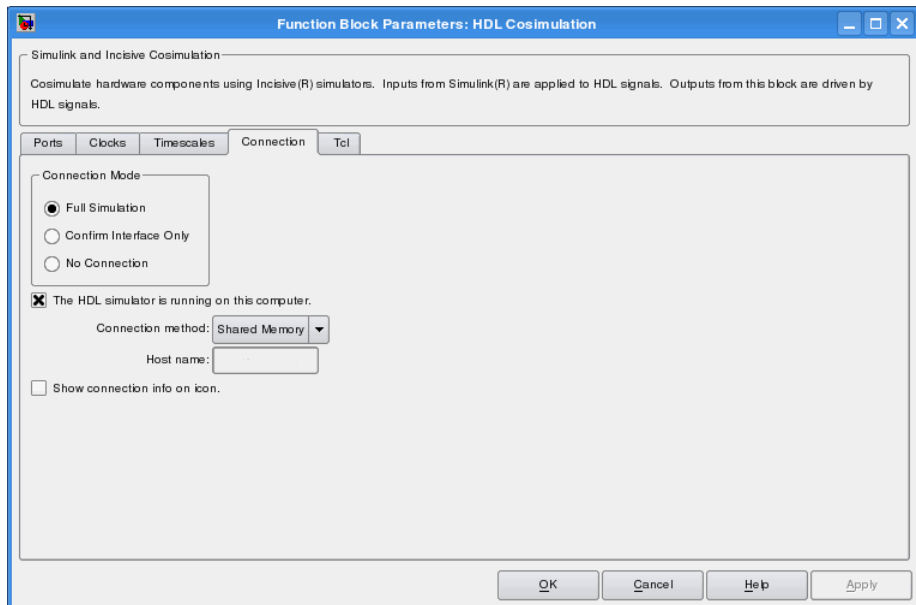
```
nclaunch('tclstart', ['-input "{@force osc_top.clk_enable 1 -after 0ns}"],
```

```
'-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}";');
```

Configuring the Communication Link in the HDL Cosimulation Block

You must select shared memory or socket communication (see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7).

After you decide, configure a block’s communication link with the **Connection** pane of the block parameters dialog box.



The following steps guide you through the communication configuration:

- 1 Determine whether Simulink and the HDL simulator are running on the same computer. If they are, skip to step 4.
- 2 Clear the **The HDL simulator is running on this computer** check box. (This check box defaults to selected.) Because Simulink and the HDL

simulator are running on different computers, **Connection method** is automatically set to **Socket**.

- 3** Enter the host name of the computer that is running your HDL simulation (in the HDL simulator) in the **Host name** text field. In the **Port number or service** text field, specify a valid port number or service for your computer system. For information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16. 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 “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7.

If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

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

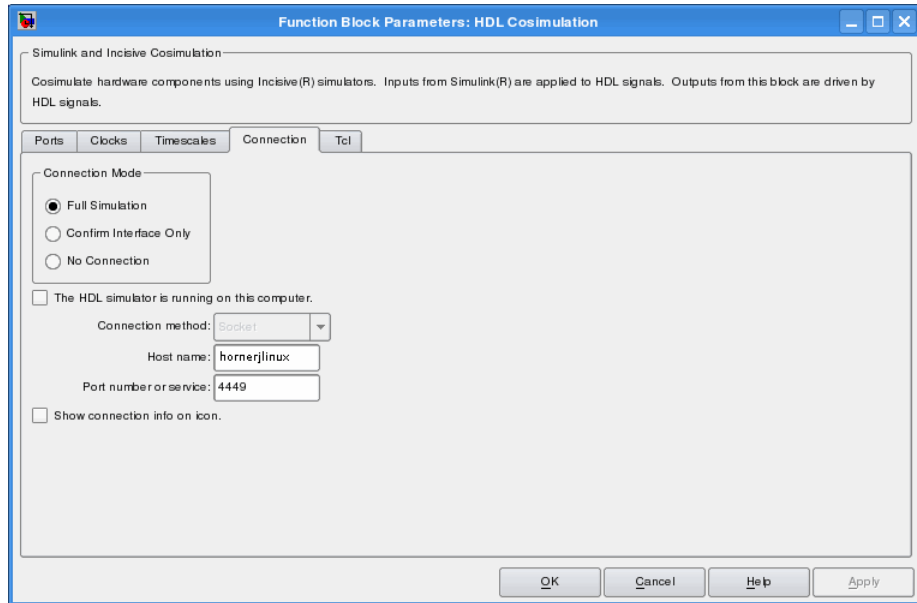
- 5** If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following options:
 - **Full Simulation:** Confirm interface and run HDL simulation (default).
 - **Confirm Interface Only:** Check HDL simulator for proper signal names, dimensions, and data types, but do not run HDL simulation.
 - **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

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

- 6** Click **Apply**.

The following example dialog box shows communication definitions for an HDL Cosimulation block. The block is configured for Simulink and the HDL

simulator running on the same computer, communicating in TCP/IP socket mode over TCP/IP port 4449.



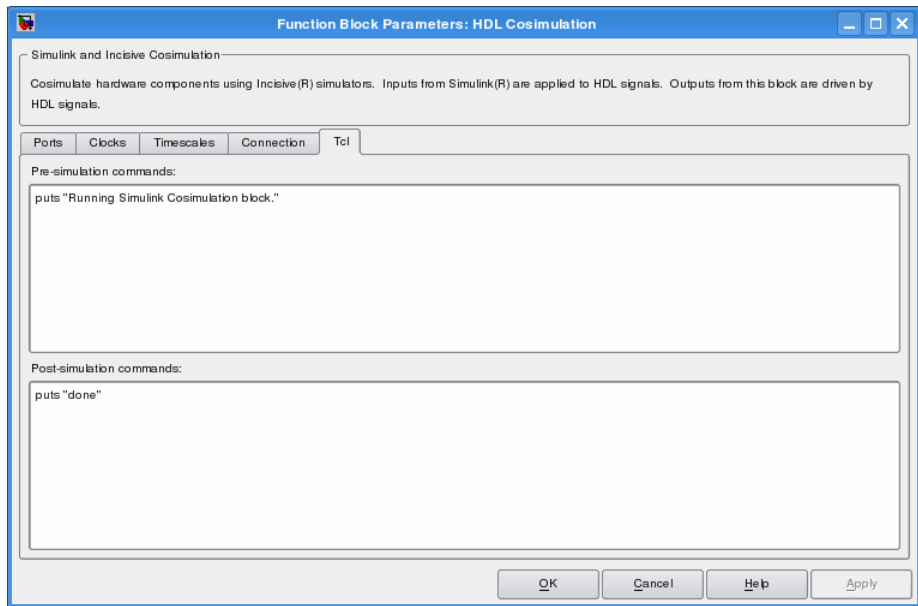
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

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

You can specify the pre-simulation and post-simulation Tcl commands by entering Tcl commands in the **Pre-simulation** commands or **Post-simulation** commands text fields of the HDL Cosimulation block.

To specify Tcl commands, perform the following steps:

- 1 Select the **Tcl** tab of the Block Parameters dialog box. The dialog box appears as follows.



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

- 2 Enter one or more commands in the **Pre-simulation command** and **Post-simulation command** text boxes. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), which is the standard Tcl concatenation operator.
- 3 Click **Apply**.

Programmatically Controlling the Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of

simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.
- Evaluated — The parameter string value undergoes an evaluation to determine its actual value used by the S-Function.

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

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

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

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

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

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

In a regression environment, you may need to determine the socket number for the Simulink/HDL simulator connection during the simulation to avoid collisions with other simulation runs. This example shows code that could handle that task. The script is for a 32-bit Linux platform.

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

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

Start the HDL Simulation

Start the simulation running in the HDL simulator.

Run a Test Bench Cosimulation Session (Simulink as Test Bench)

In this section...

“Setting Simulink Software Configuration Parameters” on page 4-41

“Determining an Available Socket Port Number” on page 4-43

“Checking the Connection Status” on page 4-43

“Running and Testing a Cosimulation Model” on page 4-43

“Avoiding Race Conditions in HDL Simulation When Cosimulating With the EDA Simulator Link HDL Cosimulation Block” on page 4-44

Setting Simulink Software Configuration Parameters

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

You can adjust the parameters individually or you can use the M-file `dspstartup`, which lets you automate the configuration process so that every new model that you create is preconfigured with the following relevant parameter settings:

Parameter	Default Setting
'SingleTaskRateTransMsg'	'error'
'Solver'	'fixedstepdiscrete'
'SolverMode'	'singletasking'
'StartTime'	'0.0'
'StopTime'	'inf'
'FixedStep'	'auto'
'SaveTime'	'off'

Parameter	Default Setting
'SaveOutput'	'off'
'AlgebraicLoopMsg'	'error'

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

You can use `dspstartup` by entering it at the MATLAB command line or by adding it to the Simulink `startup.m` file. You also have the option of customizing `dspstartup` settings. For example, you might want to adjust the `StopTime` to a value that is optimal for your simulations, or set `SaveTime` to "on" to record simulation sample times.

For more information on using and customizing `dspstartup`, see the Signal Processing Blockset documentation. For more information about automating tasks at startup, see the description of the `startup` command in the MATLAB documentation.

Determining an Available Socket Port Number

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

Checking the Connection Status

You can check the connection status by clicking the Update diagram button



or by selecting **Edit > Update Diagram**. If there is a connection error, Simulink will notify you.


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

Running and Testing a Cosimulation Model

In general, the last stage of cosimulation is to run and test your model. There are some steps you must be aware of when changing your model during or between cosimulation sessions. although your testing methods may vary depending on which HDL simulator you have, You can review these steps in “Testing the Cosimulation” on page 4-43.

Cosimulation Using the Simulink and HDL Simulator GUIs

Start the HDL simulator and load your HDL design. In Simulink, click


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

Testing the Cosimulation

If you wish to reset a clock during a cosimulation, you can do so in one of these ways:

- By entering HDL simulator force commands at the HDL simulator command prompt
- By specifying HDL simulatorforce commands in the **Post- simulation command** text field on the **Tcl** pane of the EDA Simulator Link Cosimulation block parameters dialog box.

If you change any part of the Simulink model, including the HDL Cosimulation block parameters, update the diagram to reflect those changes. You can do this update in one of the following ways:

- Rerun the simulation
- Click the Update diagram button 
- Select **Edit > Update Diagram**

Avoiding Race Conditions in HDL Simulation When Cosimulating With the EDA Simulator Link HDL Cosimulation Block

In the Cadence Incisive or NC 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 ensure the relationship between the data and active edges of the clock, you can avoid race conditions that could create nondeterministic cosimulation results.

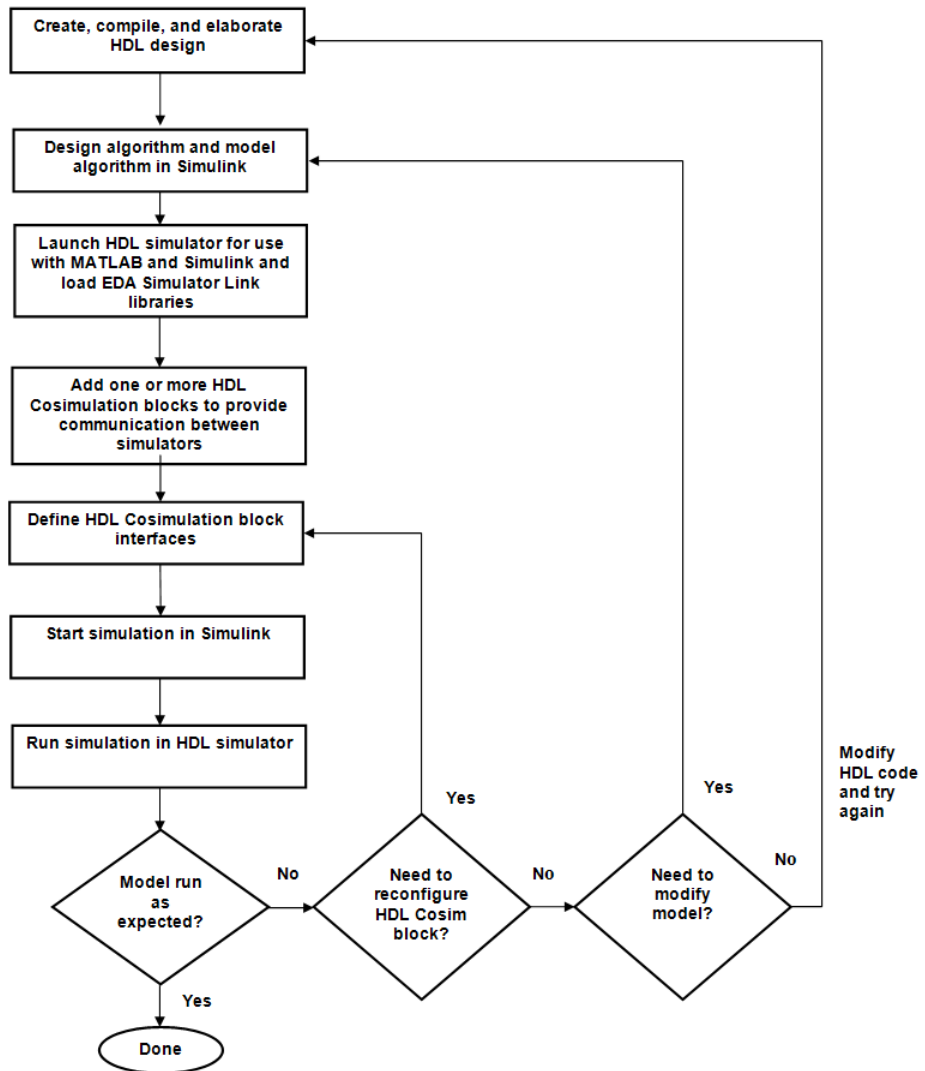
For more on race conditions in hardware simulators, see “Avoiding Race Conditions in HDL Simulators” on page 9-2.

Replacing an HDL Component with a Simulink Algorithm

- “Workflow for Using Simulink as HDL Component” on page 5-2
- “Overview to Component Simulation with Simulink” on page 5-4
- “Code an HDL Component for Use with Simulink Applications” on page 5-7
- “Create Simulink Model for Component Cosimulation with the HDL Simulator” on page 5-8
- “Launch HDL Simulator for Component Cosimulation with Simulink” on page 5-9
- “Add the HDL Cosimulation Block to the Simulink Component Model” on page 5-10
- “Define the HDL Cosimulation Block Interface” on page 5-11
- “Start the Simulation in Simulink” on page 5-12
- “Run a Component Cosimulation Session” on page 5-13

Workflow for Using Simulink as HDL Component

The following workflow shows the steps necessary to cosimulate an HDL design that Simulink software



The workflow is as follows:

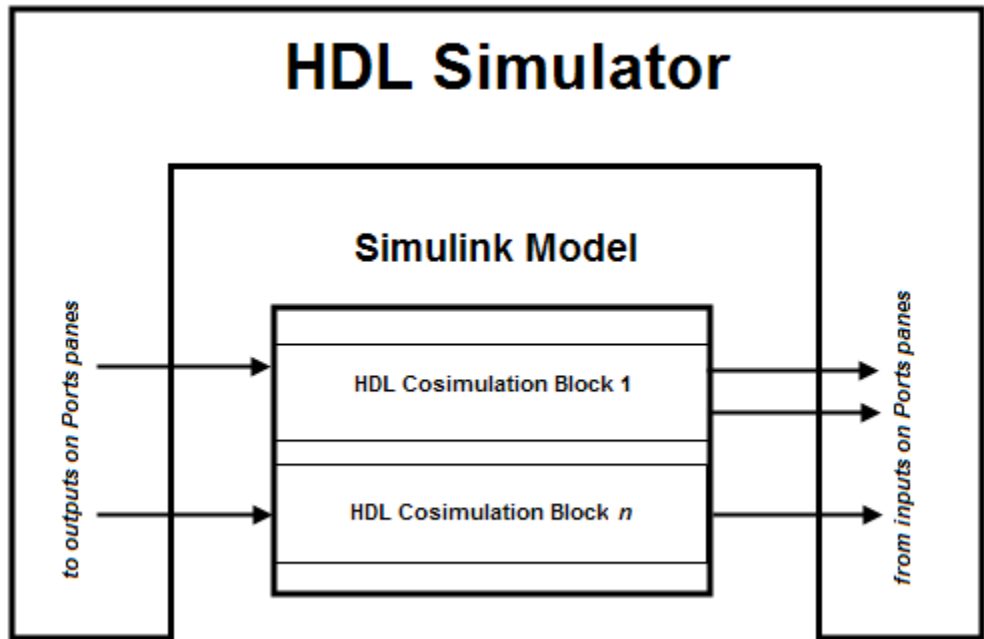
- 1** Create, compile, and elaborate HDL design.
- 2** Design algorithm and model algorithm in Simulink.
- 3** Launch HDL simulator for use with MATLAB and load libraries.
- 4** Add one or more HDL Cosimulation blocks to provide communication between simulators.
- 5** Define HDL Cosimulation block interfaces.
- 6** Start simulation in Simulink.
- 7** Run cosimulation in HDL simulator.

Overview to Component Simulation with Simulink

In this section...
“How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link” on page 5-4
“Creating the HDL Module” on page 5-5
“Introduction to the HDL Cosimulation Block (Simulink as Component)” on page 5-6

How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link

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.



Note The programming and interfacing conventions for using Simulink as a test bench functions (see Chapter 4, “Simulating an HDL Component in a Simulink Test Bench Environment”) and Simulink as a component are essentially the same; therefore, steps in the component workflow map to documentation for steps in the test bench workflow.

Creating the HDL Module

When you code an HDL module that will use Simulink to visualize a particular component, you should consider the types of data to be shared between the two environments and the direction modes.

Specifying Port Direction Modes in the HDL Module

In your module statement, you must specify each port with a direction mode (input, output, or bidirectional). See “Specifying Port Direction Modes in

HDL Components (MATLAB as Test Bench)” on page 2-8 for descriptions of these port directions.

Specifying Port Data Types in the HDL Module

You must specify data types compatible with Simulink for ports in your HDL modules. See “Specifying Port Data Types in HDL Components (MATLAB as Test Bench)” on page 2-8 for descriptions of supported port data types.

Compiling and Elaborating the HDL Design

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

Introduction to the HDL Cosimulation Block (Simulink as Component)

The 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 by using one or more HDL Cosimulation blocks fitted into the framework of a larger Simulink model posing as an HDL component.

For details, see “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 4-13.

Code an HDL Component for Use with Simulink Applications

The most basic element of communication in the interface is the HDL module. The interface passes all data between the HDL simulator and Simulink as port data. The 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. These factors are also ones you need to consider when you code an HDL module for use with Simulink as a component. For details, see “Code an HDL Component for Use with Simulink Test Bench Applications” on page 4-11.

Create Simulink Model for Component Cosimulation with the HDL Simulator

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. For more on how the HDL simulator and Simulink communicate, see “How the HDL Simulator and Simulink Software Communicate Using EDA Simulator Link” on page 5-4.

For more about creating a Simulink model for use in cosimulation with EDA Simulator Link, see “Create a Simulink Model for Test Bench Cosimulation with the HDL Simulator” on page 4-10.

Launch HDL Simulator for Component Cosimulation with Simulink

In order to be able to take advantage of some of the automatic field population on the HDL Cosimulation block, you must first launch the HDL simulator and have it ready with the libraries loaded and the HDL module loaded and ready to run.

See

- “Starting the HDL Simulator from MATLAB” on page 4-12
- “Loading an Instance of an HDL Module for Cosimulation” on page 4-12

Add the HDL Cosimulation Block to the Simulink Component Model

For instructions on adding the HDL Cosimulation block from the Simulink Library Browser, see “Add the HDL Cosimulation Block to the Simulink Test Bench Model” on page 4-13. You may also want to refer to the HDL Cosimulation block reference.

Define the HDL Cosimulation Block Interface

Define input and output ports, clocks, communication settings, and set the timing relationship. See “Define the HDL Cosimulation Block Interface (Simulink as Test Bench)” on page 4-14.

Start the Simulation in Simulink

Set Simulink configuration parameters, check connection, and start and stop the cosimulation. See “Run a Test Bench Cosimulation Session (Simulink as Test Bench)” on page 4-41.

Run a Component Cosimulation Session

Start the simulation running in the HDL simulator. Observe and control simulation behavior in Simulink. See “Start the Simulation in Simulink” on page 5-12.

Recording Simulink Signal State Transitions for Post-Processing

Adding a Value Change Dump (VCD) File

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.

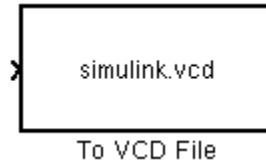
For example, including the extraction of data about a particular section of a design hierarchy or data generated during a specific time interval.

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

Note The To VCD File block logs changes to states '1' and '0' only. The block does *not* log changes to states 'X' and 'Z'.

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 the EDA Simulator Link block library. EDA Simulator Link block library.

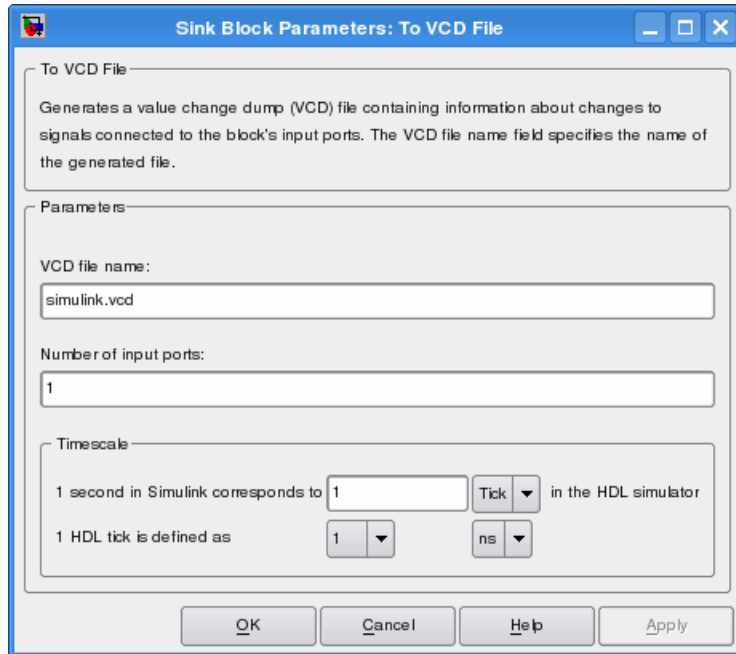


EDA Simulator Link block library.

- 4 Copy the To VCD File block from the Library Browser to your model by clicking the block and dragging it from the browser to your model window.
- 5 Connect the block ports to appropriate blocks in your Simulink model.

Note Because multidimensional signals are not part of the VCD specification, they are flattened to a 1D vector in the file.

- 6 Configure the To VCD File block by specifying values for parameters in the Block Parameters dialog box, as follows:
 - a Double-click the block icon. Simulink displays the following dialog box.



- b** Specify a file name for the generated VCD file in the **VCD file name** text box.
- If you specify a file name only, Simulink places the file in your current MATLAB folder.
 - Specify a complete path name to place the generated file in a different location.
 - If you want the generated file to have a .vcd file type extension, you must specify it explicitly.

Note Do not give the same file name to different VCD blocks. Doing so results in invalid VCD files.

- c** Specify an integer in the **Number of input ports** text box that indicates the number of block input ports on which signal data is to be collected.

The block can handle up to 94^3 (830,584) signals, each of which maps to a unique symbol in the VCD file.

- d** Click **OK**.
- 7** Choose a timing relationship between Simulink and the HDL simulator. The time scale options specify a correspondence between one second of Simulink time and some quantity of HDL simulator time. Choose relative time or absolute time. For more on the To VCD File time scale, see the reference documentation for the To VCD File block.
- 8** Run the simulation. Simulink captures the simulation data in the VCD file as the simulation runs.

For a description of the VCD file format see “VCD File Format” on page 11-27.

Defining EDA Simulator Link M-Functions and Function Parameters

- “M-Function Syntax and Function Argument Definitions” on page 7-2
- “Oscfilter Function Example” on page 7-5
- “Gaining Access to and Applying Port Information” on page 7-7

M-Function Syntax and Function Argument Definitions

The syntax of a MATLAB component function is

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

The syntax of a MATLAB test bench function is

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

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

For more information on using `tnext` and `tnow` for simulation scheduling, see “Scheduling Test Bench M-Functions Using the `tnext` Parameter” on page 2-32.

The following table describes each of the test bench and component M-function parameters and the roles they play in each of the functions.

Parameter	Test Bench	Component
<code>iport</code>	<i>Output</i> Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	<i>Input</i> Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by <code>tnow</code> .
<code>tnext</code>	<i>Output, optional</i> Specifies the time at which the HDL simulator schedules the next callback to MATLAB. <code>tnext</code> should be initialized to an empty value (<code>[]</code>). If <code>tnext</code> is not later updated, no new entries are added to the simulation schedule.	<i>Output, optional</i> Same as test bench.

Parameter	Test Bench	Component
oport	<i>Input</i> Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by tnow.	<i>Output</i> Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.
tnow	<i>Input</i> Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see “Scheduling Test Bench M-Functions Using the tnext Parameter” on page 2-32.	Same as test bench.
portinfo	<i>Input</i> 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 7-7.

Oscfilter Function Example

The following code gives the definition of the `oscfilter` MATLAB component function.

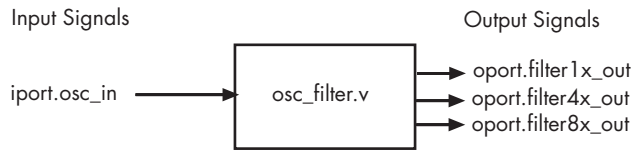
```
function [oport,tnext] = oscfilter(iport, tnow, portinfo)
```

The function name `oscfilter`, differs from the entity name `u_osc_filter`. Therefore, the component function name must be passed in explicitly to the `matlabcp` command that connects the function to the associated HDL instance using the `-mfunc` parameter.

The function definition specifies all required input and output parameters, as listed here:

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

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



Gaining Access to and Applying Port Information

EDA Simulator Link software passes information about the entity or module under test in the `portinfo` structure. The `portinfo` structure is passed as the third argument to the function. It is passed only in the first call to your MATLAB function. You can use the information passed in the `portinfo` structure to validate the entity or module under simulation. Three fields supply the information, as indicated in the next sample. . The content of these fields depends on the type of ports defined for the VHDL entity or Verilog module.

```
portinfo.field1.field2.field3
```

The following table lists possible values for each field and identifies the port types for which the values apply.

HDL Port Information

Field...	Can Contain...	Which...	And Applies to...
<i>field1</i>	in	Indicates the port is an input port	All port types
	out	Indicates the port is an output port	All port types
	inout	Indicates the port is a bidirectional port	All port types
	tscale	Indicates the simulator resolution limit in seconds as specified in the HDL simulator	All types
<i>field2</i>	<i>portname</i>	Is the name of the port	All port types

HDL Port Information (Continued)

Field...	Can Contain...	Which...	And Applies to...
<i>field3</i>	type	Identifies the port type For VHDL: integer, real, time, or enum For Verilog: 'verilog_logic' identifies port types reg, wire, integer	All port types
	right (<i>VHDL only</i>)	The VHDL RIGHT attribute	VHDL integer, natural, or positive port types
	left (<i>VHDL only</i>)	The VHDL LEFT attribute	VHDL integer, natural, or positive port types
	size	VHDL: The size of the matrix containing the data Verilog: The size of the bit vector containing the data	All port types
	label	VHDL: A character literal or label Verilog: the string '01ZX'	VHDL: Enumerated types, including predefined types BIT, STD_LOGIC, STD_ULOGIC, BIT_VECTOR, and STD_LOGIC_VECTOR Verilog: All port types

The first call to the MATLAB function has three arguments including the portinfo structure. Checking the number of arguments is one way that you can ensure that portinfo was passed. For example:

```

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

```


Additional Deployment Options

- “Diagnosing and Customizing Your Setup for Use with the HDL Simulator and EDA Simulator Link Software” on page 8-2
- “Performing Cross-Network Cosimulation” on page 8-6
- “Establishing EDA Simulator Link Machine Configuration Requirements” on page 8-12
- “Specifying TCP/IP Socket Communication” on page 8-15
- “Improving Simulation Speed” on page 8-20

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

EDA Simulator Link software provides a guided set-up script (`syscheckin`) for configuring the MATLAB and Simulink connections to your simulator. This script works whether you have installed the link software and MATLAB on the same machine as the HDL simulator or installed them on different machines.

The setup script creates a configuration file containing the location of the appropriate EDA Simulator Link MATLAB and Simulink libraries. You can then include this configuration with any other calls you make using the command `ncsim` from the HDL simulator. You only need to run this script once.

Refer to “Using the EDA Simulator Link Libraries” on page 1-13 for the correct link application library for your platform. Then see “Starting the HDL Simulator from MATLAB” on page 1-18.

For assistance in performing cross-network cosimulation, see “Performing Cross-Network Cosimulation” on page 8-6.

After you have created your configuration files, see “Starting the Cadence Incisive HDL Simulator from a Shell” on page 1-20.

Using the Configuration and Diagnostic Script for UNIX/Linux

`syscheckin` provides an easy way to configure your simulator setup to work with the EDA Simulator Link software.

The following is an example of running `syscheckin` under the following conditions:

- You have installed EDA Simulator Link on a Linux 64 machine.
- You have moved the EDA Simulator Link libraries to a different location than where you first installed them (either to another directory or to another machine).

- You want to test the TCP/IP connection.

Start syscheckin:

```
% syscheckin
*****

Kernel name: Linux
Kernel release: 2.6.22.8-mw017
Machine: x86_64
*****
```

The script first returns the location of the HDL simulator installation (ncsim.exe). If it does not find an installation, you receive an error message. Either provide the path to the installation or quit the script and install the HDL simulator. You are then prompted to accept this installation or provide a path to another one, after which you receive a message confirming the HDL simulator installation:

```
Found /hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim on the path.
Press Enter to use the path we found or enter another one:

*****

/hub/share/apps/HDLTools/IUS/IUS-61-tmw-000/lnx/tools/bin/64bit/ncsim -version
TOOL: ncsim(64) 06.11-s005
Cadence Incisive mode: 64 bits
*****
```

Next, the script needs to know where it can find the EDA Simulator Link libraries.

```
Select method to search for EDA Simulator Link libraries:
1. Use libraries in a MATLAB installation.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfihdlc_gcc323.so and liblfihdls_gcc323.so:
/tmp/extensions/incisive/linux64
Found /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so
and /tmp/extensions/incisive/linux64/liblfihdls_gcc323.so.
```

The script then runs a dependency checker to check for supporting libraries. If any of the libraries cannot be found, you probably need to append your environment path to find them.

```

*****

Running dependency checker "ldd /tmp/extensions/incisive/linux64/liblfihdlc_gcc323.so".
Dependency checker passed.
Dependency status:
  librt.so.1 => /lib/librt.so.1 (0x00002b6119631000)
  libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x00002b611973a000)
  libm.so.6 => /lib/libm.so.6 (0x00002b6119916000)
  libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00002b6119a99000)
  libc.so.6 => /lib/libc.so.6 (0x00002b6119ba6000)
  libpthread.so.0 => /lib/libpthread.so.0 (0x00002b6119de3000)
  /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
*****

```

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

```

Press Enter to load EDA Simulator Link or enter 'n' to skip this test:

ncvlog(64): 06.11-s005: (c) Copyright 1995-2007 Cadence Design Systems, Inc.
define linux64 /work/matlab/toolbox/incisive/linux64
.
.
.
ncsim> exit

*****

EDA Simulator Link libraries loaded successfully.
*****

```

Next, the script checks a TCP connection. If you choose to skip this step, the configuration file specifies use of shared memory. Both shared memory and socket configurations are in the configuration file; depending on your choice, one configuration or the other is commented out.

Press Enter to check for TCP connection or enter 'n' to skip this test:

Enter an available port [5001]

Enter remote host [localhost]

Press Enter to continue

```
ttcp_glnx -t -p5001 localhost
Connection successful
```

Lastly, the script creates the configuration file, unless for some reason you choose not to do so at this time.

Press Enter to Create Configuration files or 'n' to skip this step:

Created template files simulink9675.arg and matlab8675.arg. Inspect and modify if necessary.

Diagnosis Completed

The template file names, in this example `simulink24255.arg` and `matlab24255.arg`, have different names each time you run this script.

After the script is complete, you can leave the configuration files where they are or move them to wherever it is convenient.

Performing Cross-Network Cosimulation

In this section...
“Why Perform Cross-Network Cosimulation?” on page 8-6
“Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 8-6
“Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB” on page 8-8
“Performing Cross-Network Cosimulation with the HDL Simulator and Simulink” on page 8-10

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 or Solaris machine runs the HDL simulator. However, these procedures apply to any combination of platforms that EDA Simulator Link and the HDL simulator support.

Preparing for Cross-Network Cosimulation (MATLAB or Simulink)

Before you cosimulate between the HDL simulator and MATLAB or Simulink across a network, perform the following steps:

- 1 Create your design and testing files.

Create, compile, and elaborate your HDL design, and create your MATLAB m-function (for MATLAB cosimulation), or Simulink model (for Simulink cosimulation).

- 2 Copy EDA Simulator Link libraries to the machine with the HDL simulator
 - Go to the system where you installed MATLAB. Then, find the folder in the MATLAB distribution where the EDA Simulator Link libraries reside.

You can usually find the libraries in the default installed folder:

```
matlabroot/toolbox/edalink/extensions/incisive/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/ <i>version</i> " where <i>version</i> is the installed release (for example, R2009a).
platform	The operating system of the machine with the HDL simulator, for example, linux32. (For more information, see "Using the EDA Simulator Link Libraries" on page 1-13.)
productlibraryname	The name of the library files for MATLAB and for Simulink (for example, liblfihdlc, liblfihdls). See "Using the EDA Simulator Link Libraries" on page 1-13.

Variable	Value
compiler_tag	The compiler used to create the library (for example, gcc32 or spro). For more information, see “Using the EDA Simulator Link Libraries” on page 1-13.
ext	dll (dynamic link library—Windows only) or so (shared library extension)

For a list of all the EDA Simulator Link HDL shared libraries shipped, see “Default Libraries” on page 1-14 in “Using the EDA Simulator Link Libraries” on page 1-13.

- b** From the MATLAB machine, copy the EDA Simulator Link libraries you plan to use (which you determined in step 2) to the machine where you installed the HDL simulator. Make note of the location to which you copied the link libraries; you’ll need this information when you are actually establishing the link. For purposes of this example, the sample code refers to the destination folder as "HDLSERVER_LIB_LOCATION".

If you now want to cosimulate with MATLAB, see “Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB” on page 8-8. If you want to cosimulate with Simulink, see “Performing Cross-Network Cosimulation with the HDL Simulator and Simulink” on page 8-10.

Performing Cross-Network Cosimulation with the HDL Simulator and MATLAB

To perform an HDL-simulator-to-MATLAB cosimulation session across a network, follow these steps:

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

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

Or assign one:

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


- 2** Create a MATLAB configuration file (for loading the functions used in the HDL simulator) with the following contents:

```
//Command file for MATLAB EDA Simulator Link.
//Loading of foreign Library and HDL simulator functions.

-loadcfc /HDLSERVER_LIB_LOCATION/library_name:matlabclient
//TCL wrappers for MATLAB commands
-input @proc "nomatlabtb" "{args}" "{call" "nomatlabtb" "\$args}
-input @proc "matlabtb" "{args}" "{call" "matlabtb" "\$args}
-input @proc "matlabcp" "{args}" "{call" "matlabcp" "\$args}
-input @proc "matlabtbeval" "{args}" "{call" "matlabtbeval" "\$args}
```

Where *library_name* is the name of the library you copied in “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 8-6. You may name this configuration file anything you like.

- 3** On the machine with the HDL simulator, launch the HDL simulator from a shell with the following command:

```
ncsim -gui -f matlab_config.file design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>matlab_config.file</i>	The name of the MATLAB configuration file (from step 3)
<i>design_name</i>	The VHDL or Verilog design you want to load

- 4** In the HDL simulator, schedule the test bench or component (*matlabcp* or *matlabtb*). Specify the socket port number from step 1 and the name of the host where *hdldaemon* is running.

Performing Cross-Network Cosimulation with the HDL Simulator and Simulink

When you want to perform an HDL-simulator-to-Simulink cosimulation session across a network, follow these steps:

- 1 Launch the HDL simulator from a shell with the following command:

```
ncsim -gui -loadvpi "/HDLSERVER_LIB_LOCATION/library_name:simlinkserver"
        +socket=socket_num design_name
```

where the arguments shown in the following table have the values indicated.

Argument	Value
<i>library_name</i>	The name of the library you copied to the machine with the HDL simulator (in “Preparing for Cross-Network Cosimulation (MATLAB or Simulink)” on page 8-6).
<i>socket_num</i>	The socket number you have chosen for this connection
<i>design_name</i>	The VHDL or Verilog design you want to load

- 2 On the machine with MATLAB and Simulink, start Simulink and open your Simulink model.
- 3 Double-click on the HDL Cosimulation block to open the Function Block Parameters dialog box.
- 4 Click on the **Connections** tab.
 - a Clear the check box labeled **The HDL simulator is running on this computer**. The Connection method is automatically changed to Socket.
 - b In the **Host name** box, enter the host name of the machine where the HDL simulator is located.

- c** In the **Port number or service** box, enter the socket number from step 1.
- d** Click **OK** to exit block dialog box, and save your changes.

Next, run your simulation, add more blocks, or make other desired changes. See Chapter 4, “Simulating an HDL Component in a Simulink Test Bench Environment” or Chapter 5, “Replacing an HDL Component with a Simulink Algorithm”.

Establishing EDA Simulator Link Machine Configuration Requirements

In this section...

“Valid Configurations For Using the EDA Simulator Link Software with MATLAB Applications” on page 8-12

“Valid Configurations For Using the EDA Simulator Link Software with Simulink Software” on page 8-13

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

The following list provides samples of valid configurations for using the Cadence Incisive HDL simulator and the EDA Simulator Link software with MATLAB software. The scenarios apply whether the HDL simulator is running on the same or different computing system as the MATLAB software. In a network configuration, you use an Internet address in addition to a TCP/IP socket port to identify the servers in an application environment.

- An HDL simulator session linked to a MATLAB function `foo` through a single instance of the MATLAB server
- An HDL simulator session linked to multiple MATLAB functions (for example, `foo` and `bar`) through a single instance of the MATLAB server
- An HDL simulator session linked to a MATLAB function `foo` through multiple instances of the MATLAB server (each running within the scope of a unique MATLAB session)
- Multiple HDL simulator sessions each linked to a MATLAB function `foo` through multiple instances of the MATLAB server (each running within the scope of a unique MATLAB session)
- Multiple HDL simulator sessions each linked to a different MATLAB function (for example, `foo` and `bar`) through the same instance of the MATLAB server
- Multiple HDL simulator sessions each linked to MATLAB function `foo` through a single instance of the MATLAB server

Although multiple HDL simulator sessions can link to the same MATLAB function in the same instance of the MATLAB server, as this configuration scenario suggests, such links are not recommended. If the MATLAB function maintains state (for example, maintains global or persistent variables), you may experience unexpected results because the MATLAB function does not distinguish between callers when handling input and output data. If you must apply this configuration scenario, consider deriving unique instances of the MATLAB function to handle requests for each HDL entity.

Notes

- Shared memory communication is an option for configurations that require only one communication link on a single computing system.
 - TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.
 - In any configuration, an instance of MATLAB can run only one instance of the EDA Simulator Link MATLAB server (hdldaemon) at a time.
 - In a TCP/IP configuration, the MATLAB server can handle multiple client connections to one or more HDL simulator sessions.
-

Valid Configurations For Using the EDA Simulator Link Software with Simulink Software

The following list provides samples of valid configurations for using the Cadence Incisive HDL simulator and the EDA Simulator Link software with Simulink software. The scenarios apply whether the HDL simulator is running on the same or different computing system as the MATLAB or Simulink products. In a network configuration, you use an Internet address in addition to a TCP/IP socket port to identify the servers in an application environment.

- An HDL Cosimulation block in a Simulink model linked to a single HDL simulator session

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

Notes

- HDL Cosimulation blocks in a Simulink model can connect to the same or different HDL simulator sessions.
 - TCP/IP socket communication is required for configurations that use multiple communication links on one or more computing systems. Unique TCP/IP socket ports distinguish the communication links.
 - Shared memory communication is an option for configurations that require only one communication link on a single computing system.
-

Specifying TCP/IP Socket Communication

In this section...

“Communication Modes and Socket Ports” on page 8-15

“Choosing TCP/IP Socket Ports” on page 8-16

“Specifying TCP/IP Values” on page 8-18

“TCP/IP Services” on page 8-19

Communication Modes and Socket Ports

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

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

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

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

The port can be a TCP/IP port number, TCP/IP port alias or service name, or the value zero, indicating that the port is to be assigned by the operating system. See “Specifying TCP/IP Values” on page 8-18 for some valid examples.

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

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

For more information on modes of communication, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7. For more information on establishing the HDL simulator end of the communication link, see “Start hdd daemon to Provide Connection to HDL Simulator” on page 2-22.

Choosing TCP/IP Socket Ports

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

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

Consider registering a port you choose to use.

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

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

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

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

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

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

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

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

Specifying TCP/IP Values

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

If the HDL simulator and MATLAB are running on the same system, the TCP/IP specification identifies a unique TCP/IP socket port to be used for the link. If the two applications are running on different systems, you must specify a remote host name or Internet address in addition to the socket port. See “Specifying TCP/IP Socket Communication” on page 8-15 for more detail in specifying TCP/IP values.

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

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

An example of a matlabcp call using port 4449 might look like this:

A remote connection might look like this:

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

TCP/IP Services

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

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

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

Improving Simulation Speed

In this section...

“Obtaining Baseline Performance Numbers” on page 8-20

“Analyzing Simulation Performance” on page 8-20

“Cosimulating Frame-Based Signals with Simulink” on page 8-22

Obtaining Baseline Performance Numbers

You can baseline the performance numbers by timing the execution of the HDL and the Simulink model separately and adding them together; you may not expect better performance than that. Make sure that the separate simulations are representative: running an HDL-only simulator with unrealistic input stimulus could be much faster than when proper input stimulus is provided.

Analyzing Simulation Performance

While cosimulation entails a certain amount of overhead, sometimes the HDL simulation itself also slows performance. Ask yourself these questions when trying to analyze and improve performance:

Consideration	Suggestions for Improving Speed
Are you are using NFS or other remote file systems?	How fast is the file system? Consider using a different type or expect that the file system you're using will impact performance.
Are you using separate machines for Simulink and the HDL simulator?	How fast is the network? Wait until the network is quieter or contact your system administrator for advice on improving the connection.

Consideration	Suggestions for Improving Speed
Are you using the same machine for Simulink and the HDL simulator?	<ul style="list-style-type: none"> • Are you using shared pipes instead of sockets? Shared memory is faster. • Are the Simulink and HDL processes large enough to cause swaps to disk? Consider adding more memory; otherwise be aware that you're running a huge process and expect it to impact performance.
Are you using <i>optimal</i> (that is, as large as possible) Simulink sample rates on the HDL Cosimulation block?	<p>For example, if you set the output sample rate to 1 but only use every 10th sample, you could make the rate 10 and reduce the traffic between Simulink and the HDL simulator.</p> <p>Another example is if you place a very fast clock as an input to the HDL Cosimulation block, but have none of the other inputs need such a fast rate. In that case, you should generate the clock in HDL or via the Clocks or Tcl pane on the HDL Cosimulation block.</p>
Are you using Simulink Accelerator™ mode?	Acceleration mode can speed up the execution of your model. See "Accelerating Models" in the <i>Simulink User's Guide</i> .
If you have the Communications Blockset software, have you considered using Framed signals?	Framed signals reduce the number of Simulink/HDL interactions.

Cosimulating Frame-Based Signals with Simulink

Overview to Cosimulation with Frame-Based Signals

Frame-based processing can improve the computational time of your Simulink models, because multiple samples can be processed at once. Use of frame-based signals also lets you simulate the behavior of frame-based systems more accurately. The HDL Cosimulation block supports processing of single-channel frame-based signals.

A *frame* of data is a collection of sequential samples from a single channel or multiple channels. One frame of a single-channel signal is represented by a M-by-1 column vector. A signal is *frame based* if it is propagated through a model one frame at a time.

Frame-based processing requires the Signal Processing Blockset software. Source blocks from the Signal Processing Sources library let you specify a frame-based signal by setting the **Samples per frame** block parameter. Most other signal processing blocks preserve the frame status of an input signal. You can use the Buffer block to buffer a sequence of samples into frames.

See “Working with Signals” in the Signal Processing Blockset documentation for detailed information about frame-based processing.

Using Frame-Based Processing

You do not need to configure the HDL Cosimulation 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 Cosimulation block. All such signals must meet the requirements described in “Frame-Based Processing Requirements and Restrictions” on page 8-23. The HDL Cosimulation block automatically configures any outputs for frame-based operation at the appropriate frame size.

Use of frame-based signals affects only the Simulink side of the cosimulation. The behavior of the HDL code under simulation in the HDL simulator does not change in any way. Simulink assumes that HDL simulator processing is sample based. Simulink assembles samples acquired from the HDL simulator into frames as required. Conversely, Simulink transmits output data to the

HDL simulator in frames, which are unpacked and processed by the HDL simulator one sample at a time.

Frame-Based Processing Requirements and Restrictions

Observe the following restrictions and requirements when connecting frame-based signals in to an HDL Cosimulation block:

- Connection of mixed frame-based and sample-based signals to the same HDL Cosimulation block is not supported.
- Only single-channel frame-based signals can be connected to the HDL Cosimulation block. Use of multichannel (matrix) frame-based signals is not supported in this release.
- All frame-based signals connected to the HDL Cosimulation 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 scalar values, not vectors or arrays (with the exception of bit vectors, as VHDL and Verilog bit vectors are converted to the appropriately sized fixed-point scalar data type by the HDL Cosimulation block).

Advanced Operational Topics

- “Avoiding Race Conditions in HDL Simulators” on page 9-2
- “Performing Data Type Conversions” on page 9-5
- “Understanding How Simulink Software Drives Cosimulation Signals” on page 9-14
- “Understanding the Representation of Simulation Time” on page 9-15
- “Handling Multirate Signals” on page 9-24
- “Understanding Block Simulation Latency” on page 9-25
- “Interfacing with Continuous Time Signals” on page 9-27

Avoiding Race Conditions in HDL Simulators

In this section...

“Overview to Avoiding Race Conditions” on page 9-2

“Potential Race Conditions in Simulink Link Sessions” on page 9-2

“Potential Race Conditions in MATLAB Link Sessions” on page 9-3

“Further Reading” on page 9-4

Overview to Avoiding Race Conditions

A well-known issue in hardware simulation is the potential for nondeterministic results when race conditions are present. Because the HDL simulator is a highly parallel execution environment, you must write the HDL such that the results do not depend on the ordering of process execution.

Although there are well-known coding idioms for ensuring successful simulation of a design under test, you must always take special care at the testbench/DUT interfaces for applying stimulus and reading results, even in pure HDL environments. For an HDL/foreign language interface, such as with a Simulink or MATLAB link session, the problem is compounded if there is no common synchronization signal, such as a clock coordinating the flow of data.

Potential Race Conditions in Simulink Link Sessions

All the signals on the interface of an HDL Cosimulation block in the Simulink library have an intrinsic sample rate associated with them. This sample rate can be thought of as an implicit clock that controls the simulation time at which a value change can occur. Because this implicit clock is completely unknown to the HDL engine (that is, it is not an HDL signal), the times at which input values are driven into the HDL or output values are sampled from the HDL are asynchronous to any clocks coded in HDL directly, even if they are nominally at the same frequency.

For Simulink value changes scheduled to occur at a specific simulation time, the HDL simulator does not make any guarantees as to the order that value change occurs versus some other blocking signal assignment. Thus, if the

Simulink values are driven/sampled at the same time as an active clock edge in the HDL, there is a race condition.

For cases where your active HDL clock edge and your intrinsic Simulink active clock edges are at the same frequency, you can ensure proper data propagation by offsetting one of those edges. Because the Simulink sample rates are always aligned with time 0, you can accomplish this offset by shifting the active clock edge in the HDL off of time 0. If you are coding the clock stimulus in HDL, use a delay operator ("after" or "#") to accomplish this offset.

When using a Tcl "force" command to describe the clock waveform, you can simply put the first active edge at some nonzero time. Using a nonzero value allows a Simulink sample rate that is the same as the fundamental clock rate in your HDL. This example shows a 20 ns clock (so the Simulink sample rates will also be every 20 ns) with an active positive edge that is offset from time 0 by 2 ns:

```
Cadence> force top.clk = 1'b0 -after 0 ns 1'b1 -after 2 ns 1'b0
        -after 12 ns -repeat 20 ns
```

For HDL Cosimulation Blocks with Clock panes, you can define the clock period and active edge in that pane. The waveform definition places the **non-active** edge at time 0 and the **active** edge at time T/2. This placement ensures the maximum setup and hold times for a clock with a 50% duty cycle.

If the Simulink sample rates are at a different frequency than the HDL clocks, then you must synchronize the signals between the HDL and Simulink as you would do with any multiple time-domain design, even one in pure HDL. For example, you can place two synchronizing flip-flops at the interface.

If your cosimulation does not include clocks, then you must also treat the interfacing of Simulink and the HDL code as being between asynchronous time domains. You may need to over-sample outputs to ensure that all data transitions are captured.

Potential Race Conditions in MATLAB Link Sessions

When you use the `-sensitivity`, `-rising_edge`, or `-falling_edge` scheduling options to `matlabtb` or `matlabcp` to trigger MATLAB function calls, the propagation of values follow the same semantics as a pure HDL design;

you are guaranteed that the triggers must occur before the results can be calculated. You still can have race conditions, but they can be analyzed within the HDL alone.

However, when you use the `-time` scheduling option to `matlabtb` or `matlabcp`, or use `"tnext"` within the MATLAB function itself, the driving of signal values or sampling of signal values cannot be guaranteed in relation to any HDL signal changes. It is as if the potential race conditions in that time-based scheduling are like an implicit clock that is unknown to the HDL engine and not visible by just looking at the HDL code.

The remedies are the same as for the Simulink signal interfacing: ensure the sampling and driving of signals does not occur at the same simulation times as the MATLAB function calls.

Further Reading

Problems interfacing designs from testbenches and foreign languages, including race conditions in pure HDL environments, are well-known and extensively documented. Some texts that describe these issues include:

- The documentation for each vendor's HDL simulator product
- The HDL standards specifications
- *Writing Testbenches: Functional Verification of HDL Models*, Janick Bergeron, 2nd edition, © 2003
- *Verilog and SystemVerilog Gotchas*, Stuart Sutherland and Don Mills, © 2007
- *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, Chris Spear, © 2007
- *Principles of Verifiable RTL Design*, Lionel Bening and Harry D. Foster, © 2001

Performing Data Type Conversions

In this section...

“Converting HDL Data to Send to MATLAB” on page 9-5

“Array Indexing Differences Between MATLAB and HDL” on page 9-7

“Converting Data for Manipulation” on page 9-9

“Converting Data for Return to the HDL Simulator” on page 9-10

Converting HDL Data to Send to MATLAB

If your HDL application needs to send HDL data to a MATLAB function, you may first need to convert the data to a type supported by MATLAB and the EDA Simulator Link software.

To program a MATLAB function for an HDL model, you must understand the type conversions required by your application. You may also need to handle differences between the array indexing conventions used by the HDL you are using and MATLAB (see following section).

The data types of arguments passed in to the function determine the following:

- The types of conversions required before data is manipulated
- The types of conversions required to return data to the HDL simulator

The following table summarizes how the EDA Simulator Link software converts supported VHDL data types to MATLAB types based on whether the type is scalar or array.

VHDL-to-MATLAB Data Type Conversions

VHDL Types...	As Scalar Converts to...	As Array Converts to...
STD_LOGIC, STD_ULOGIC, and BIT	A character that matches the character literal for the desired logic state.	

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED		A column vector of characters (as defined in VHDL Conversions for the HDL Simulator on page 9-11) with one bit per character.
Arrays of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED		An array of characters (as defined above) with a size that is equivalent to the VHDL port size.
INTEGER and NATURAL	Type int32.	Arrays of type int32 with a size that is equivalent to the VHDL port size.
REAL	Type double.	Arrays of type double with a size that is equivalent to the VHDL port size.
TIME	Type double for time values in seconds and type int64 for values representing simulator time increments (see the description of the 'time' option in hdldaemon).	Arrays of type double or int64 with a size that is equivalent to the VHDL port size.
Enumerated types	Character array (string) that contains the MATLAB representation of a VHDL label or character literal. For example, the label high converts to 'high' and the character literal 'c' converts to ''c''.	Cell array of strings with each element equal to a label for the defined enumerated type. Each element is the MATLAB representation of a VHDL label or character literal. For example, the vector (one, '2', three) converts to the column vector ['one'; ''2''; 'three']. A user-defined enumerated type that contains only character literals, and then converts to a vector or array of

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
		characters as indicated for the types <code>STD_LOGIC_VECTOR</code> , <code>STD_ULONGIC_VECTOR</code> , <code>BIT_VECTOR</code> , <code>SIGNED</code> , and <code>UNSIGNED</code> .

The following table summarizes how the EDA Simulator Link software converts supported Verilog data types to MATLAB types. The software supports only scalar data types for Verilog.

Verilog-to-MATLAB Data Type Conversions

Verilog Types...	Converts to...
wire, reg	A character or a column vector of characters that matches the character literal for the desired logic states (bits).
integer	A 32-element column vector of characters that matches the character literal for the desired logic states (bits).

Array Indexing Differences Between MATLAB and HDL

In multidimensional arrays, the same underlying OS memory buffer maps to different elements in MATLAB and the HDL simulator (this mapping only reflects different ways the different languages offer for naming the elements of the same array). When you use both `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 <code>double</code>	Use the <code>double</code> function to convert the data to type <code>double</code> before performing the computation. For example: <pre>datas(inc+1) = double(idata);</pre>
Convert a standard logic or bit vector to an unsigned integer or positive decimal	Use the <code>mv12dec</code> function to convert the data to an unsigned decimal value. For example: <pre>uval = mv12dec(oport.val)</pre> <p>This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent.</p> <p>The <code>mv12dec</code> function converts the binary data that the MATLAB function receives from the entity's <code>osc_in</code> port to unsigned decimal values that MATLAB can compute.</p> <p>See <code>mv12dec</code> for more information on this function.</p>
Convert a standard logic or bit vector to a negative decimal	Use the following application of the <code>mv12dec</code> function to convert the data to a signed decimal value. For example: <pre>suval = mv12dec(oport.val, true);</pre>

Required Data Conversions (Continued)

If You Need the Function to...	Then...
	This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent.

Examples

The following code excerpt illustrates data type conversion of data passed in to a callback:

```
InDelayLine(1) = InputScale * mvl2dec(iport.osc_in',true);
```

This example tests port values of VHDL type `STD_LOGIC` and `STD_LOGIC_VECTOR` by using the `all` function as follows:

```
all(oport.val == '1' | oport.val
== '0')
```

This example returns `True` if all elements are '1' or '0'.

Converting Data for Return to the HDL Simulator

If your simulation MATLAB function needs to return data to the HDL simulator, you may first need to convert the data to a type supported by the EDA Simulator Link software. The following tables list circumstances under which such conversions are required for VHDL and Verilog.

Note When data values are returned to the HDL simulator, the char array size must match the HDL type, including leading zeroes, if needed. For example:

```
oport.signal = dec2mv1(2)
```

will only work if `signal` is a 2-bit type in HDL. If the HDL type is anything else, you *must* specify the second argument:

```
oport.signal = dec2mv1(2, N)
```

where `N` is the number of bits in the HDL data type.

VHDL Conversions for the HDL Simulator

To Return Data to an IN Port of Type...	Then...
STD_LOGIC, STD_ULOGIC, or BIT	<p>Declare the data as a character that matches the character literal for the desired logic state. For <code>STD_LOGIC</code> and <code>STD_ULOGIC</code>, the character can be 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'. For <code>BIT</code>, the character can be '0' or '1'. For example:</p> <pre>iport.s1 = 'X'; %STD_LOGIC iport.bit = '1'; %BIT</pre>
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:</p> <pre>iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR iport.bitv = '10100'; %BIT_VECTOR iport.uns = dec2mv1(10,8); %UNSIGNED, 8 bits</pre>
Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	<p>Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See “Array Indexing Differences Between MATLAB and HDL” on page 9-7.</p>

VHDL Conversions for the HDL Simulator (Continued)

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

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...
Enumerated types	<p>Declare the data as a string for scalar ports or a cell array of strings for array ports with each element equal to a label for the defined enumerated type. The 'label' field of the portinfo structure lists all valid labels (see “Gaining Access to and Applying Port Information” on page 7-7). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as in the first example shown here. .</p> <pre data-bbox="556 690 1053 777"> iport.char = {'A', 'B'}; %Character %literal iport.undef = 'mylabel'; %User-defined label </pre>
Character array for standard logic or bit representation	<p>Use the dec2mv1 function to convert the integer. For example:</p> <pre data-bbox="556 887 1023 916"> oport.slva =dec2mv1([23 99],8)'; </pre> <p>This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.</p>

Verilog Conversions for the HDL Simulator

To Return Data to an input Port of Type...	Then...
reg, wire	<p>Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state. For example:</p> <pre data-bbox="556 1329 793 1359"> iport.bit = '1'; </pre>
integer	<p>Declare the data as a 32-element column vector of characters (as defined above) with one bit per character.</p>

Understanding How Simulink Software Drives Cosimulation Signals

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

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

Note Verify that signals used in cosimulation have read/write access. You can check read/write access through the HDL simulator—see HDL simulator documentation for details. This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Understanding the Representation of Simulation Time

In this section...

“Overview to the Representation of Simulation Time” on page 9-15

“Defining the Simulink and HDL Simulator Timing Relationship” on page 9-16

“Setting the Timing Mode with EDA Simulator Link” on page 9-16

“Relative Timing Mode” on page 9-18

“Absolute Timing Mode” on page 9-20

“Timing Mode Usage Considerations” on page 9-21

“Setting HDL Cosimulation Port Sample Times” on page 9-23

Overview to the Representation of Simulation Time

The representation of simulation time differs significantly between the HDL simulator and Simulink. Each application has its own timing engine and the link software must synchronize the simulation times between the two.

In the HDL simulator, the unit of simulation time is referred to as a *tick*. The duration of a tick is defined by the HDL simulator *resolution limit*. The default resolution limit is 1 ns, but may vary depending on the simulator.

To determine the current HDL simulator resolution limit, enter `echo $timescale` at the HDL simulator prompt. See the HDL simulator documentation for the application you are using 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 EDA Simulator Link interface:

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

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

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

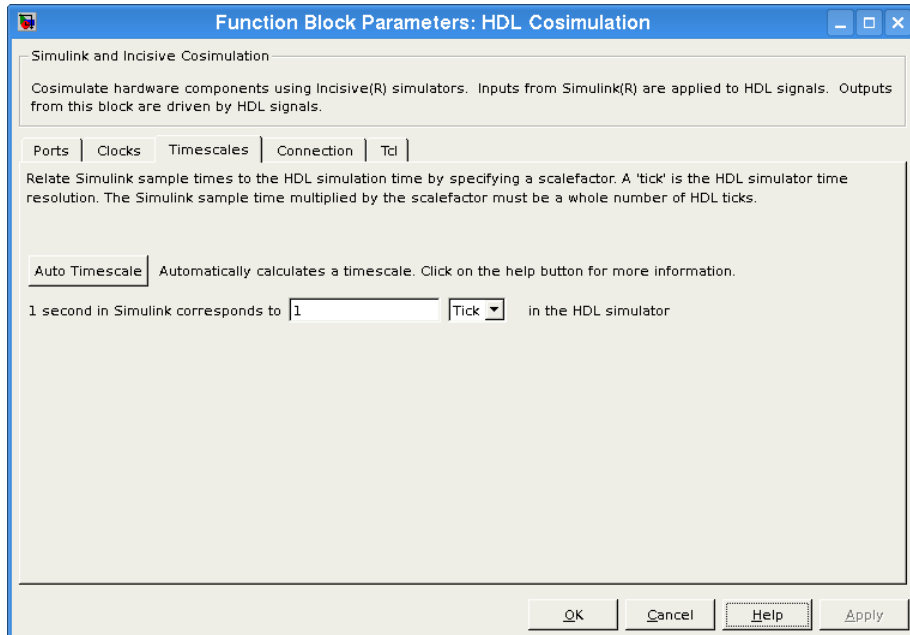
When you allow the link software to define the timing relationship, it attempts to set the timescale factor between the HDL simulator and Simulink to be as close as possible to 1 second in the HDL simulator = 1 second in Simulink. If this setting is not possible, the link product attempts to set the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

Setting the Timing Mode with EDA Simulator Link

The **Timescales** pane of the HDL Cosimulation block parameters dialog box defines a correspondence between one second of Simulink time and some quantity of HDL simulator time. This quantity of HDL simulator time can be expressed in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. The HDL Cosimulation block defaults to relative timing mode for cosimulation. For more on relative timing mode, see “Relative Timing Mode” on page 9-18.
- 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 9-20.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the HDL simulator, either by entering the HDL simulator equivalent or by clicking on **Auto Timescale**. The next figure shows the default settings of the **Timescales** pane.



For instructions on setting the timing mode either automatically or manually, see “Timescales Pane” on page 11-14 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
```

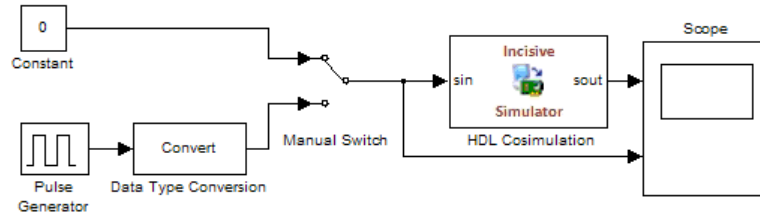
where *InTicks* is the HDL simulator time in ticks, *tInSecs* is the Simulink time in seconds, and *N* is a scale factor.

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 *T_{si}* seconds, new values will be deposited on the HDL input port at exact multiples of *T_{si}* ticks. If an output port has an explicitly specified sample time of *T_{so}* seconds, values will be read from the HDL simulator at multiples of *T_{so}* ticks.

Relative Timing Mode Example.



The model contains an HDL Cosimulation block (labeled HDL_Cosimulation1) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following code excerpt lists the Verilog code for the inverter:

```

module inverter_clock_v1(sin, sout,clk);

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

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

```

A cosimulation of this model might have the following settings:

- Simulation parameters in Simulink:
 - **Timescales** parameters: 1 Simulink second = 10 HDL simulator ticks
 - Total simulation time: 30 s
 - Input port (inverter_clock_v1.sin) sample time: N/A
 - Output port (inverter_clock_v1.sout) sample time: 1 s
 - Clock (inverter_clock_v1.clk) period: 5 s
- HDL simulator resolution limit: 1 ns

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

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 9-18. Suppose that the model is reconfigured as follows:

- Simulation parameters in Simulink:
 - **Timescale** parameters: 1 s of Simulink time corresponds to 1 s of HDL simulator time.
 - Total simulation time: $60e-9$ s (60ns)
 - Input port (/inverter/inport) sample time: $24e-9$ s (24 ns)
 - Output port (/inverter/outport) sample time: $12e-9$ s (12 ns)
 - Clock (inverter/clock) period: $10e-9$ s (10 ns)
- HDL simulator resolution limit: 1 ns

Given these simulation parameters, the Simulink software will cosimulate with the HDL simulator for 60 ns, during which Simulink will sample inputs at intervals of 24 ns, update outputs at intervals of 12 ns, and drive clocks at intervals of 10 ns.

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 9-21
- “Non-Integer Time Periods” on page 9-22

Timing Mode Usage Restrictions

The following restrictions apply to the use of absolute and relative timing modes:

- When multiple HDL Cosimulation blocks in a model are communicating with a single instance of the HDL simulator, all HDL Cosimulation blocks must have the same **Timescales** pane settings.

- If you change the **Timescales** pane settings in an HDL Cosimulation block between consecutive cosimulation runs, you must restart the simulation in the HDL simulator.
- If you specify a Simulink sample time that cannot be expressed as a whole number of HDL ticks, you will get an error.

Non-Integer Time Periods

When using non-integer time periods, the HDL simulator cannot represent such an infinitely repeating value. So the simulator truncates the time period, but it does so differently than how Simulink truncates the value, and the two time periods no longer match up.

The following example demonstrates how to set the timing relationship in the following scenario: you want to use a sample period of $\frac{1}{3Hz}$ in Simulink, which corresponds to a non-integer time period.

The key idea here is that you must always be able to relate a Simulink time with an HDL tick. The HDL tick is the finest time slice the HDL simulator recognizes.

However, a 3 Hz signal actually has a period of 333.333333333... ms, which is not a valid tick period for the HDL simulator. The HDL simulator will truncate such numbers. But Simulink does not make the same decision; thus, for cosimulation where you are trying to keep two independent simulators in synchronization, you should not assume anything. Instead you have to decide whether it is convenient to truncate or round the number.

Therefore, the solution is to "snap" either the Simulink sample time or the HDL sample time (via the timescale) to valid numbers. There are infinite possibilities, but here are some possible ways to perform a snap:

- Change Simulink sample times from 1/3 sec to 0.33333 sec and set the cosimulation block timescale to '1 second in Simulink = 1 second in the HDL simulator'. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 0.33333 sec.
- Keep Simulink sample times at 1/3 sec. and 1 second in Simulink = 6 ticks in the HDL simulator. If you are specifying a clock in the HDL

Cosimulation block **Clocks** pane, its period should be 1/3. Briefly, this specification tells Simulink to make each Simulink sample time correspond to every $(1/3 * 6) = 2$ ticks, regardless of the HDL time resolution. If your default HDL simulator resolution is 1 ns, that means your HDL sample times are every 2 ns. This sample time will work in a way so that for every Simulink sample time there is a corresponding HDL sample time; however, Simulink thinks in terms of 1/3 sec periods and the HDL in terms of 2 ns periods. Thus, you could get confused during debug. If you want this to match the real period (such as to 5 places, i.e. 333.33ms), you can follow the next option listed.

- Keep Simulink sample times at 1/3 sec and 1 second in Simulink = 0.99999e9 ticks in the HDL simulator. If you are specifying a clock in the HDL Cosimulation block **Clocks** pane, its period should be 1/3.

Setting HDL Cosimulation 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.

Handling Multirate Signals

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

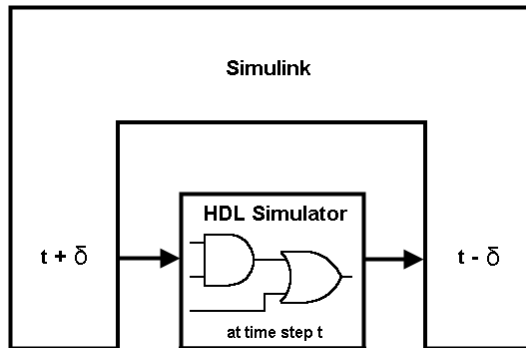
The HDL Cosimulation block also lets you specify an independent sample time for each output port. You must explicitly set the sample time for each output port, or accept the default. Using this setting lets you control the rate at which Simulink updates an output port by reading the corresponding signal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

Understanding Block Simulation Latency

Overview to Block Simulation Latency

Simulink and the EDA Simulator Link Cosimulation blocks supplement the hardware simulator environment, rather than operate as part of it. During cosimulation, Simulink does not participate in the HDL simulator delta-time iteration. From the Simulink perspective, all signal drives (reads) occur during a single delta-time cycle. For this reason, and due to fundamental differences between the HDL simulator and Simulink with regard to use and treatment of simulation time, some degree of latency is introduced when you use EDA Simulator Link Cosimulation blocks. The latency is a time lag that occurs between when Simulink begins the deposit of a signal and when the effect of the deposit is visible on cosimulation block output.

As the following figure shows, Simulink cosimulation block input affects signal values just after the current HDL simulator time step ($t+\delta$) and block output reflects signal values just before the current HDL simulator step time ($t-\delta$).



Regardless of whether you specify your HDL code with latency, the cosimulation block has a minimum latency that is equivalent to the cosimulation block's output sample time. For large sample times, the delay can appear to be quite long, but this length occurs as an artifact of the cosimulation block, which exchanges data with the HDL simulator at the block's output sample time only. Such length may be reasonable for a cosimulation block that models a device that operates on a clock edge only, such as a register-based device.

For cosimulation blocks that model combinatorial circuits, you may want to experiment with a faster sampling frequency for output ports in order to reduce this latency.

Interfacing with Continuous Time Signals

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

Functions — Alphabetical List

dec2mvl

Purpose (For Incisive) Convert decimal integer to binary string

Syntax `dec2mvl(d)`
`dec2mvl(d,n)`

Description `dec2mvl(d)` returns the binary representation of `d` as a multivalued logic string. `d` must be an integer smaller than 2^{52} .
`dec2mvl(d,n)` produces a binary representation with at least `n` bits.

Examples The following function call returns the string '10111':

```
dec2mvl(23)
```

The following function call returns the string '01001':

```
dec2mvl(-23)
```

The following function call returns the string '11101001':

```
dec2mvl(-23,8)
```

See Also `mvl2dec`

Purpose (For Incisive) Control MATLAB server that supports interactions with Incisive simulator

Syntax

```
hdldaemon
s=hdldaemon
hdldaemon('ParameterName',ParameterValue)
s=hdldaemon('ParameterName',ParameterValue)
hdldaemon('Option')
```

Description hdldaemon starts the HDL Link MATLAB server using shared memory interprocess communication. Only one hdldaemon per MATLAB session can be running at any given time.

s=hdldaemon starts the MATLAB server using shared memory and returns the server status connection in structure s.

hdldaemon('ParameterName',ParameterValue) starts the MATLAB server using shared memory and accepts optional inputs as one or more comma-separated parameter-value pairs. *ParameterName* is the name of the parameter inside single quotes. *ParameterValue* is the value corresponding to *ParameterName*. To start the server in socket mode, use the 'socket' parameter.

s=hdldaemon('ParameterName',ParameterValue) works the same as hdldaemon('ParameterName',ParameterValue) and returns the server status connection in structure s.

hdldaemon('Option') accepts a single optional input. Only one option may be specified in a single call. You must establish the server connection before calling hdldaemon with one of these options.

Inputs

Option

Select one of the following options:

- 'kill'

Shuts down the MATLAB server without shutting down MATLAB.

- 'stop'
Shuts down the MATLAB server without shutting down MATLAB. There is no difference between using 'kill' and 'stop'.
- 'status'
Displays status of the MATLAB server. You can also use `s=hdldaemon('status')`, which displays MATLAB server status and returns status in structure `s`.

Parameter/Value Pairs

'time'

Specifies how the MATLAB server sends and returns time values.

- 'int64'
Specifies that the MATLAB server send and return time values in the MATLAB function callbacks as 64-bit integers representing the number of simulation steps.
See the `matlabcp/matlabtb` `know` parameter reference (Chapter 7, “Defining EDA Simulator Link M-Functions and Function Parameters”).
- 'sec'
Specifies that the MATLAB server sends and returns time values in the MATLAB function callbacks as `double` values that EDA Simulator Link scales to seconds based on the current HDL simulation resolution.

Default: 'sec'

'quiet'

Suppresses printing diagnostic messages. Errors still appear. Use this option to suppress the MATLAB server shutdown message when using `hdldaemon` to get an unused socket number.

- 'true'
Suppress printing diagnostic messages.
- 'false'
Do not suppress printing diagnostic messages.

Default: 'false'

'socket'

Defines the TCP/IP port used for communication. The socket value can be:

- 0, indicating the host automatically chooses a valid TCP/IP port
- An explicit port number (1024 < port < 49151)
- A service name (that is, alias) from `/etc/services` file

If you specify the operating system option (0), use `hdldaemon('status')` to acquire the assigned socket port number.

See “Specifying TCP/IP Socket Communication” on page 8-15 for more information about TCP/IP ports.

'tclcmd'

Transmits a Tcl command to all connected clients.

You may specify any valid Tcl command string. The Tcl command string you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the string cannot include commands such as `start`, `stop`, or `restart`.

Note You can issue this command only after the software establishes a server connection.

hdldaemon

Caution

Do not call `hdldaemon(tclcmd, 'tclcmd')` from inside a `matlabtb` or `matlabcp` function. Doing so results in a race condition, and the simulator hangs.

Outputs

`s`

`s` is a structure with these fields:

- `comm`
Shared memory or sockets
- `connections`
Number of open connections
- `ipc_id`
File system name (for shared memory communication channel) or TCP/IP port number (for socket)

Examples

Start the MATLAB server using shared memory communication and use an integer representation of time:

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

Start MATLAB server and specify socket communication on port 4449:

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

Start MATLAB server with socket communication and use a 64-bit representation of time:

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

Check hdldaemon server status:

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

Returns, for example,

```
HDLDaemon socket server is running on port 4449 with 1 connections
```

Or

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

Or

```
HDLDaemon is NOT running
```

Check connection information for communication mode, number of existing connections, and the interprocess communication identifier (`ipc_id`) the MATLAB server is using for a link:

```
x=hdldaemon('status')
```

For a socket connection, returns:

```
x =  
      comm: 'sockets'  
 connections: 0  
      ipc_id: '4449'
```

For shared memory, returns:

```
x =  
      comm: 'shared memory'  
 connections: 0  
      ipc_id: [1x45 char]
```

hdldaemon

You can examine `ipc_id` by entering it at the MATLAB command prompt:

```
>>x.ipc_id
```

Shut down server without shutting down MATLAB:

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

Issue simple Tcl commands:

```
hdldaemon('tclcmd','puts "This is a test"')
```

Issue complex Tcl commands:

See the demo for [Implementing the Filter Component of an Oscillator in MATLAB](#) for an extensive example of a compound Tcl command.

See Also

`nclaunch`

How To

- “Start `hdldaemon` to Provide Connection to HDL Simulator” on page 2-22
- “Run MATLAB Test Bench Simulation” on page 2-34
- “Stop Test Bench Simulation” on page 2-39
- “Run MATLAB Component Simulation” on page 3-17

Purpose	(For Incisive) Load instantiated HDL design for verification with MATLAB
Syntax	<code>hdlsimmatlab <instance> [<ncsim_args>]</code>
Description	<p>The <code>hdlsimmatlab</code> command loads the specified instance of an HDL design for verification and sets up the Cadence Incisive simulator so it can establish a communication link with MATLAB. The Cadence Incisive simulator opens a simulation workspace as it loads the HDL design.</p> <p>This command may be run from the HDL simulator prompt or from a Tcl script shell (<code>tclsh</code>).</p> <p>This command is issued in the HDL simulator.</p>
Arguments	<p><code><instance></code> Specifies the instance of an HDL design to load for verification.</p> <p><code><ncsim_args></code> Specifies one or more <code>ncsim</code> command arguments. For details, see the description of <code>ncsim</code> in the Cadence Incisive simulator documentation.</p>
Examples	<p>The following command loads the module instance <code>parse</code> from library <code>work</code> for verification and sets up the Cadence Incisive simulator so it can establish a communication link with MATLAB:</p> <pre>tclshell> hdlsimmatlab work.parse</pre>

hdlsimulink

Purpose (For Incisive) Load instantiated HDL design for cosimulation with Simulink

Syntax `hdlsimulink [<ncsim_args>] <instance>
[-socket <tcp_spec>]`

Description The `hdlsimulink` command loads the specified instance of an HDL design for cosimulation and sets up the Cadence Incisive simulator so it can establish a communication link with Simulink. The Cadence Incisive simulator opens a simulation workspace into which it loads the HDL design.

This command is issued in the HDL simulator.

Argument `<ncsim_args>`
Specifies one or more `ncsim` command arguments. At a minimum, either `-gui` or `-tcl` is required. If you specify `-gui`, the Simulink GUI launches when the HDL design is loaded. If you specify `-tcl`, a Tcl script shell launches instead. If you do not specify either of these arguments, the HDL simulator runs the simulation without Simulink. Other valid `ncsim` arguments may be specified in addition to `-gui` or `-tcl`. For more information on `-gui`, `-tcl`, or other `ncsim` arguments, see the description of `ncsim` in the Cadence Incisive simulator documentation.

`<instance>`
Specifies the instance of an HDL design to load for cosimulation.

`-socket <tcp_spec>`
Specifies TCP/IP socket communication for the link between the Cadence Incisive simulator and MATLAB. This setting overrides the setting specified with the MATLAB `nclaunch` function. The `<tcp_spec>` can consist of a TCP/IP socket port number or service name (alias). For example, you might specify port number 4449 or the service name `matlabSERVICE`.

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

Note The communication mode that you specify with the `hdlsimulink` command must match what you specify for the communication mode when you configure EDA Simulator Link blocks in your Simulink model. For more information on modes of communication, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7. For more information on establishing the Simulink end of the communication link, see “Configuring the Communication Link in the HDL Cosimulation Block” on page 4-34.

Examples

The following command loads the module instance `parse` from library `work` for cosimulation, sets up the Cadence Incisive simulator so it can establish a communication link with Simulink, and opens a Tcl script shell:

```
tclshell> hdlsimulink -gui work.parse
```

Purpose (For Incisive) Associate MATLAB component function with instantiated HDL design

Syntax

```
matlabcp <instance>  
[<time-specs>]  
[-socket <tcp-spec>]  
[-rising <port>[,<port>...]]  
[-falling <port> [,<port>,...]]  
[-sensitivity <port>[,<port>,...]]  
[-mfunc <name>]  
[-use_instance_obj]  
[-argument]
```

Description The matlabcp command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator Link software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous matlabcp command that specified the same instance. For example, if you issue the command matlabcp for instance foo, all previously scheduled events initiated by matlabcp on foo are canceled.

This command is issued in the HDL simulator.

MATLAB component functions simulate the behavior of modules 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. See Chapter 3, “Replacing an HDL Component with a MATLAB Component Function”.

Notes The communication mode that you specify for `matlabcp` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabcp` command in the HDL simulator.

Arguments

<instance>

Specifies an instance of an HDL design that is associated with a MATLAB function. By default, `matlabcp` associates the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabcp` associates the instance with the MATLAB function `myfirfilter` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabcp` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Do not specify an instance of an HDL module that has already been associated with a MATLAB M-function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

<time-specs>

Specifies a combination of time specifications consisting of any or all of the following:

<timen>,...

Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space. For example:

```
matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec
```

The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero.

Note For time-based parameters, you can specify any standard time units (ns, us, and so on). If you do not specify units, the command treats the time value as a value of HDL simulation ticks.

-repeat <time>

Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified <timen>, ... pattern. The time values are relative to the value of tnow at the time the HDL simulator first calls the MATLAB function.

-cancel <time>

Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of tnow at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a nomatlabtb call.

Note The -cancel option works only with the <time-specs> arguments. It does not affect any of the other scheduling arguments for matlabcp.

Note Place time specifications after the `matlabcp` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the `matlabcp` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. When you provide TCP/IP information for `matlabcp`, you can choose a TCP/IP port number or TCP/IP port alias or service name for the `<tcp_spec>` parameter. If you are setting up communication between computers, you must also specify the name or Internet address of the remote host that is running the MATLAB server (`hdldaemon`). See “Specifying TCP/IP Values” on page 8-18 for some valid `tcp_spec` examples.

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp_spec>` on the command line.

Note The communication mode that you specify with the `matlabcp` command must match what you specify for the communication mode when you issue the `hdldaemon` command in MATLAB.

For more information on modes of communication, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7. For more information on establishing the MATLAB end of the communication link, see “Start `hdldaemon` to Provide Connection to HDL Simulator” on page 2-22.

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- Verilog: Rising edge is the transition from 0, x, or z to 1.
- VHDL: Rising edge is '0'-'>'1'. Z and X will not create a rate transition.

Specify `-rising` with the path names of one or more signals defined as a logic type.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a falling edge—changes from '1' to '0'. Specify `-falling` with the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on).

For determining signal transition in:

- Verilog: Falling edge is the transition from 1 to x, z or 0.
- VHDL: Falling edge is '1'-'0'. Values 'X', 'Z', 'H', and 'L' will be ignored.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level in the HDL model hierarchy.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

`-mfunc <name>`

The name of the MATLAB function that is associated with the HDL module instance you specify for instance. By default, the

EDA Simulator Link software invokes a MATLAB function that has the same name as the specified HDL instance. Thus, if the names are the same, you can omit the `-mfunc` option. If the names are not the same, use this argument when you call `matlabcp`. If you omit this argument and `matlabcp` does not find a MATLAB function with the same name, the command generates an error message.

`-use_instance_obj` (*Beta Feature*)

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by EDA Simulator Link to the M-function. You include the `-use_instance_obj` argument with `matlabcp` in the following format:

```
matlabcp modelname -mfunc funcname -use_instance_obj
```

When you call `matlabcp` with the `use_instance_obj` argument, the M-function 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
<code>tnext</code>	Write only	Used to schedule a callback during the set time value. This field is equivalent to old <code>tnext</code> . For example: <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</pre> will schedule a callback at time equals 5 nanoseconds from <code>tnow</code> .
<code>userdata</code>	Read/Write	Stores state variables of the current <code>matlabcp</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.

Field	Read/Write Access	Description
simstatus	Read only	<p>Stores the status of the HDL simulator. The EDA Simulator Link software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. simstatus is a read-only property.</p> <pre data-bbox="644 505 1074 626"> >> hdl_instance_obj.simstatus ans= Init </pre>
instance	Read only	<p>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:</p> <p>In the HDL simulator:</p> <pre data-bbox="644 857 1237 878"> hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj </pre> <p>In MATLAB:</p> <pre data-bbox="644 979 1059 1100"> >> hdl_instance_obj.instance ans= osc_top </pre>

Field	Read/Write Access	Description
argument	Read only	<p>Stores the argument set by the <code>-argument</code> option of <code>matlabcp</code>. For example:</p> <pre>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</pre> <p>The link software supports the <code>-argument</code> option only when it is used with <code>-use_instance_obj</code>, otherwise the argument is ignored. <code>argument</code> is a read-only property.</p> <pre>>>hdl_instance_obj.argument ans= foo</pre>
portinfo	Read only	<p>Stores information about the VHDL and Verilog ports associated with this instance. <code>portinfo</code> is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information" on page 7-7.</p> <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <hr/> <p>Note When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p> <hr/>

Field	Read/Write Access	Description
tscal	Read only	<p>Stores the resolution limit (tick) in seconds of the HDL simulator. tscal is a read-only property.</p> <pre>>> hdl_instance_obj.tscal</pre> <pre>ans = 1.0000e-009</pre> <hr/> <p>Note When you use use_instance_obj, you access tscal through the HDL instance object. If you do not use use_instance_obj, you can still access tscal through portinfo.</p> <hr/>
tnow	Read only	<p>Stores the current time. tnow is a read-only property.</p> <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + faststrate;</pre>
portvalues	Read/Write	<p>Stores the current values of and sets new values for the output and input ports for a matlabcp instance. For example:</p> <pre>>> hdl_instance_obj.portvalues</pre> <pre>ans = Read Only Input ports: clk_enable: [] clk: [] reset: [] Read/Write Output ports: sine_out: [22x1 char]</pre>
linkmode	Read only	<p>Stores the status of the callback. The EDA Simulator Link software sets this field to 'testbench' if the callback is</p>

Field	Read/Write Access	Description
		associated with <code>matlabtb</code> and 'component' if the callback is associated with <code>matlabcp</code> . <code>linkmode</code> is a read-only property. <pre>>> hdl_instance_obj.linkmode ans= component</pre>

-argument (*Beta Feature*)

Used to pass user-defined arguments from the `matlabcp` invocation on the HDL side to the M-function callbacks. Supported with `-use_instance_obj` only. See the field listing under the `-use_instance_obj` property.

Examples

The following examples demonstrate some ways you might use the `matlabcp` function.

Using `matlabcp` with the `-mfunc` option to Associate an HDL Component with a MATLAB M-Function of a Different Name

This example explicitly associates the Verilog module `vlogtestbench_top.u_matlab_component` with the MATLAB function `vlogmatlabcp` using the `-mfunc` option. The `'-socket'` option specifies using socket communication on port 4449.

```
matlabcp vlogtestbench_top.u_matlab_component -mfunc vlogmatlabcp -socket 4449
```

Using `matlabcp` with Explicit Times and the `-cancel` Option

This example implicitly associates the Verilog module, `vtestbench_top`, with the MATLAB function `vlogtestbench_top`, and includes explicit times with the `-cancel` option.

```
matlabcp vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns
```

Using matlabcp with Rising and Falling Edges

This example implicitly associates the Verilog module, `vlogtestbench_top`, with the MATLAB function `vlogtestbench_top`, and also uses rising and falling edges.

```
hldsimsim> matlabcp vlogtestbench_top 1 2 3 4 5 6 7 -rising outclk3
-falling u_matlab_component/inoutclk
```

Using matlabcp and the HDL Instance Object (*Beta Feature*)

In this example, the HDL simulator makes repeated calls to `matlabcp` to bind multiple HDL instances to the same M-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 M-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  
  
...
```

Purpose (For Incisive) Schedule MATLAB test bench session for instantiated HDL module

Syntax

```
matlabtb <instance>
[<time-specs>]
[-socket <tcp-spec>]
[-rising <port>[,<port>...]]
[-falling <port> [,<port>,...]]
[-sensitivity <port>[,<port>,...]]
[-mfunc <name>]
[-use_instance_obj]
[-argument]
```

Description The matlabtb command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator Link software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous matlabtb command that specified the same instance. For example, if you issue the command matlabtb for instance foo, all previously scheduled events initiated by matlabtb on foo are canceled.

This command is issued in the HDL simulator.

MATLAB test bench functions mimic stimuli passed to entities in the HDL model. You force stimulus from MATLAB or HDL scheduled with matlabtb.

Notes The communication mode that you specify for `matlabtb` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabtb` command in the HDL simulator.

Arguments

<instance>

Specifies the instance of an HDL module that the EDA Simulator Link software associates with a MATLAB test bench function. By default, `matlabtb` associates the instance with a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabtb` associates the instance with the MATLAB function `myfirfilter` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabtb` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Do not specify an instance of an HDL module that has already been associated with a MATLAB M-function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

<time-specs>

Specifies a combination of time specifications consisting of any or all of the following:

<timen>,...

Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space. For example:

```
matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec
```

The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero.

Note For time-based parameters, you can specify any standard time units (ns, us, and so on). If you do not specify units, the command treats the time value as a value of HDL simulation ticks.

-repeat <time>

Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified <timen>, ... pattern. The time values are relative to the value of tnow at the time the HDL simulator first calls the MATLAB function. For example:

```
matlabtb vlogtestbench_top 5 ns -repeat 10 ns
```

The MATLAB function executes at time equals 0 ns, 5 ns, 15 ns, 25 ns, and so on.

-cancel <time>

Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of tnow at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a nomatlabtb call.

Note The `-cancel` option works only with the `<time-specs>` arguments. It does not affect any of the other scheduling arguments for `matlabtb`.

Note Place time specifications after the `matlabtb` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the `matlabtb` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. When you provide TCP/IP information for `matlabtb`, you can choose a TCP/IP port number or TCP/IP port alias or service name for the `<tcp_spec>` parameter. If you are setting up communication between computers, you must also specify the name or Internet address of the remote host that is running the MATLAB server (`hdldaemon`). See “Specifying TCP/IP Values” on page 8-18 for some valid `tcp_spec` examples.

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp_spec>` on the command line.

Note The communication mode that you specify with the `matlabtb` command must match what you specify for the communication mode when you issue the `hdldaemon` command in MATLAB. For more information on modes of communication, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7. For more information on establishing the MATLAB end of the communication link, see “Start `hdldaemon` to Provide Connection to HDL Simulator” on page 2-22.

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on).

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a

falling edge—changes from '1' to '0'. Specify `-falling` with the path names of one or more signals defined as a logic type (STD_LOGIC, BIT, X01, and so on).

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level of the HDL design.

If you specify the option with no signals, the interface is sensitive to value changes for all signals.

Note Use of this option for INOUT ports can result in double calls.

For example:

```
-sensitivity /randnumgen/dout
```

The MATLAB function executes if the value of `dout` changes.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full path name format. If you do not specify a full path name, the command applies the HDL simulator rules to resolve signal specifications.

-mfunc <name>

The name of the associated MATLAB function. If you omit this argument, `matlabtb` associates the HDL module instance to a MATLAB function that has the same name as the HDL instance. If you omit this argument and `matlabtb` does not find a MATLAB function with the same name, the command generates an error message.

-use_instance_obj (*Beta Feature*)

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by EDA Simulator Link to the M-function. The `-use_instance_obj` argument is called with `matlabtb` in the following format:

```
matlabtb modelname -mfunc funcname -use_instance_obj
```

When you call `matlabcp` with the `use_instance_obj` argument, the M-function 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
<code>tnext</code>	Write only	Used to schedule a callback during the set time value. This field is equivalent to old <code>tnext</code> . For example: <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</pre> will schedule a callback at time equals 5 nanoseconds from <code>tnow</code> .
<code>userdata</code>	Read/Write	Stores state variables of the current <code>matlabtb</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.

Field	Read/Write Access	Description
simstatus	Read only	<p>Stores the status of the HDL simulator. The EDA Simulator Link software sets this parameter to 'Init' during the first callback for this particular instance and to 'Running' thereafter. simstatus is a read-only property.</p> <pre>>> hdl_instance_obj.simstatus ans= Init</pre>
instance	Read only	<p>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:</p> <p>In the HDL simulator:</p> <pre>hdlsim> matlabtb osc_top -mfunc oscfilter use_instance_obj</pre> <p>In MATLAB:</p> <pre>>> hdl_instance_obj.instance ans= osc_top</pre>

Field	Read/Write Access	Description
argument	Read only	<p>Stores the argument set by the <code>-argument</code> option of <code>matlabtb</code>. For example:</p> <pre>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</pre> <p>The link software supports the <code>-argument</code> option only when it is used with <code>-use_instance_obj</code>, otherwise the argument is ignored. <code>argument</code> is a read-only property.</p> <pre>>> hdl_instance_obj.argument ans= foo</pre>
portinfo	Read only	<p>Stores information about the VHDL and Verilog ports associated with this instance. <code>portinfo</code> is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size. For more information on port data, see “Gaining Access to and Applying Port Information” on page 7-7.</p> <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <hr/> <p>Note When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p> <hr/>

Field	Read/Write Access	Description
tscale	Read only	<p>Stores the resolution limit (tick) in seconds of the HDL simulator. tscale is a read-only property.</p> <pre>>> hdl_instance_obj.tscale ans= 1.0000e-009</pre> <hr/> <p>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.</p> <hr/>
tnow	Read only	<p>Stores the current time. tnow is a read-only property.</p> <pre>hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;</pre>

Field	Read/Write Access	Description
portvalues	Read/Write	<p>Stores the current values of and sets new values for the output and input ports for a matlabtb instance. For example:</p> <pre data-bbox="709 482 1151 765"> >> hdl_instance_obj.portvalues ans = Read/Write Input ports: clk_enable: [] clk: [] reset: [] Read Only Output ports: sine_out: [22x1 char]</pre> <p>For example, you can set the reset port to 1 by calling <code>hdl_instance_obj.portvalues.reset = '1'</code>.</p>
linkmode	Read only	<p>Stores the status of the callback. The EDA Simulator Link software sets this parameter to 'testbench' if the callback is associated with matlabtb and 'component' if the callback is associated with matlabcp. linkmode is a read-only property.</p> <pre data-bbox="709 1199 1121 1321"> >> hdl_instance_obj.linkmode ans= testbench</pre>

-argument (*Beta Feature*)

Used to pass user-defined arguments from the matlabtb instantiation on the HDL side to the M-function callbacks. Supported with -use_instance_obj only. See the field listing for argument under the -use_instance_obj property.

Examples

The following examples demonstrate some ways you might use the matlabtb function.

Using matlabtb with the -socket Argument and Time Parameters

The following command starts the HDL simulator client component of EDA Simulator Link, associates an instance of the entity, myfirfilter, with the MATLAB function myfirfilter, and begins a local TCP/IP socket-based test bench session using TCP/IP port 4449. Based on the specified test bench stimuli, myfirfilter.m executes 5 nanoseconds from the current time, and then repeatedly every 10 nanoseconds:

```
hdlsim> matlabtb myfirfilter 5 ns -repeat 10 ns -socket 4449
```

Applying Rising Edge Clocks and State Changes with matlabtb

The following command starts the HDL simulator client component of EDA Simulator Link, and begins a remote TCP/IP socket-based session using remote MATLAB host compb and TCP/IP port 4449. Based on the specified test bench stimuli, myfirfilter.m executes 10 nanoseconds from the current time, each time the signal /top/fclk experiences a rising edge, and each time the signal /top/din changes state.

```
hdlsim> matlabtb /top/myfirfilter 10 ns -rising /top/fclk -sensitivity /top/din  
-socket 4449@computer123
```

Specifying a MATLAB M-Function Name and Sensitizing Signals with matlabtb

The following command starts the HDL simulator client component of the EDA Simulator Link software. The '-mfunc' option specifies the

M-function to connect to and the '-socket' option specifies the port number for socket connection mode. '-sensitivity' indicates that the test bench session is sensitized to the signal sine_out.

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

matlabtbeval

Purpose (For Incisive) Call specified MATLAB function once and immediately on behalf of instantiated HDL module

Syntax matlabtbeval <instance> [-socket <tcp_spec>]
[-mfunc <name>]

Description The matlabtbeval command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator Link software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Executes the specified MATLAB function once and immediately on behalf of the specified module instance.

This command is issued in the HDL simulator.

Note The matlabtbeval command executes the MATLAB function immediately, while matlabtb provides several options for scheduling MATLAB function execution.

Notes The communication mode that you specify for matlabtbeval must match the communication mode you specified for hdldaemon when you established the server connection.

For socket communications, specify the port number you selected for hdldaemon when you issue a link request with the matlabtbeval command in the HDL simulator.

Arguments <instance>
Specifies the instance of an HDL module that is associated with a MATLAB function. By default, matlabtbeval associates the

HDL module instance with a MATLAB function that has the same name as the HDL module instance. For example, if the HDL module instance is `myfirfilter`, `matlabtbeval` associates the HDL module instance with the MATLAB function `myfirfilter`. Alternatively, you can specify a different MATLAB function with the `-mfunc` property.

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. For TCP/IP socket communication on a single computer, the `<tcp_spec>` can consist of just a TCP/IP port number or service name (alias). If you are setting up communication between computers, you must also specify the name or Internet address of the remote host. See “Specifying TCP/IP Values” on page 8-18 for some valid `tcp_spec` examples.

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

Note The communication mode that you specify with the `matlabtbeval` command must match what you specify for the communication mode when you call the `hdldaemon` command to start the MATLAB server. For more information on communication modes, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7. For more information on establishing the MATLAB end of the communication link, see “Start `hdldaemon` to Provide Connection to HDL Simulator” on page 2-22.

`-mfunc <name>`

The name of the associated MATLAB function. If you omit this argument, `matlabtbeval` associates the HDL module instance with a MATLAB function that has the same name as the HDL module instance.. If you omit this argument and `matlabtbeval` does not find a MATLAB function with the same name, the command displays an error message.

Examples

This example starts the HDL simulator client component of the link software, associates an instance of the module `myfirfilter` with the function `myfirfilter.m`, and uses a local TCP/IP socket-based communication link to TCP/IP port 4449 to execute the function `myfirfilter.m`:

```
> matlabtbeval myfirfilter -socket 4449:
```

Purpose	(For Incisive) Convert multivalued logic to decimal
Syntax	<pre>mvl2dec('mv_logic_string') mvl2dec('mv_logic_string', signed)</pre>
Description	<p><code>mvl2dec('mv_logic_string')</code> converts a multivalued logic string to a positive decimal. If <i>mv_logic_string</i> contains any character other than '0' or '1', NaN is returned. <i>mv_logic_string</i> must be a vector.</p> <p><code>mvl2dec('mv_logic_string', signed)</code> converts a multivalued logic string to a positive or a negative decimal. If <i>signed</i> is true, this function assumes the first character <i>mv_logic_string</i>(1) to be a signed bit of a 2s complement number. If <i>signed</i> is missing or false, the multivalued logic string becomes a positive decimal.</p>
Examples	<p>The following function call returns the decimal value 23:</p> <pre>mvl2dec('010111')</pre> <p>The following function call returns NaN:</p> <pre>mvl2dec('xxxxxx')</pre> <p>The following function call returns the decimal value -9:</p> <pre>mvl2dec('10111', true)</pre>
See Also	<code>dec2mvl</code>

Purpose (For Incisive) Start and configure Cadence Incisive simulators for use with EDA Simulator Link software

Syntax `nclaunch('PropertyName', 'PropertyValue'...)`

Description `nclaunch('PropertyName', 'PropertyValue'...)` starts the Cadence Incisive simulator for use with the MATLAB and Simulink features of the EDA Simulator Link software. The first folder in the Cadence Incisive simulator matches your MATLAB current folder if you do not specify an explicit `rundir` parameter.

After you call this function, you can use EDA Simulator Link functions for the HDL simulator (for example, `hdlsimmatlab`, `hdlsimulink`) to do interactive debug setup.

The property name/property value pair settings allow you to customize the Tcl commands used to start the Cadence Incisive simulator, the `ncsim` executable to be used, the path and name of the Tcl script that stores the start commands, and for Simulink applications, details about the mode of communication to be used by the applications. You must use a property name/property value pair with `nclaunch`.

Property Name/Property Value Pairs

- `'hdlsimdir', 'pathname'`
Specifies the path name to the Cadence Incisive simulator executable to be started. By default, the function uses the first version of the simulator that it finds on the system path (defined by the path variable). Use this option to start different versions of the Cadence Incisive simulator or if the version of the simulator you want to run does not reside on the system path.
- `'hdlsimexe', 'simexename'`
Specifies the name of a Cadence Incisive simulator executable. By default, this function uses `'ncsim'`. You can specify a custom-built simulator executable with `'simexename.'`
- `'libdir', 'folder'`
Specifies the folder containing MATLAB shared libraries. This property creates an entry in the startup Tcl file that points to the

folder with the shared libraries needed for the Cadence Incisive simulator to communicate with MATLAB when the Cadence Incisive simulator runs on a machine that does not have MATLAB.

'libfile', 'library_file_name'

Specifies a particular library file. This value defaults to the version of the library file that was built using the same compiler that MATLAB itself uses. If the HDL simulator links other libraries, including SystemC libraries, that were built using a compiler supplied with the HDL simulator, you can specify an alternate library file with this property. See “Using the EDA Simulator Link Libraries” on page 1-13 for versions of the library built using other compilers.

'rundir', 'dirname'

Specifies where to run the HDL simulator. By default, the function uses the current working folder.

The following conditions apply to this name/value pair:

- If the value of `dirname` is “TEMPDIR”, the function creates a temporary folder in which it runs the HDL simulator.
- If you specify `dirname` and the folder does *not* exist, you will get an error.

'socketsimulink', 'tcp_spec'

Specifies TCP/IP socket communication for links between the Cadence Incisive simulator and Simulink. See “Specifying TCP/IP Values” on page 8-18 for valid TCP/IP examples. For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

If the Cadence Incisive simulator and Simulink run on the same computing system, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

'starthdlsim', ['yes' | 'no']

Determines whether the Cadence Incisive simulator is launched. This parameter defaults to `yes`, which launches the Cadence Incisive simulator and creates a startup Tcl file. If you set `starthdlsim` to `no`, the function does not launch the Cadence Incisive simulator, but it still creates a startup Tcl file.

This startup Tcl file contains pointers to MATLAB and Simulink shared libraries. To run the Cadence Incisive simulator manually, see “Starting the HDL Simulator” on page 1-18.

'startupfile', 'pathname'

Each invocation of `nclaunch` creates a Tcl script that, when executed, compiles and launches the HDL simulator. By default, this function generates a filename of `compile_and_launch.tcl` in the folder specified by `rundir`. With this property, you can specify the name and location of the generated Tcl script. If the file name already exists, that file’s contents are overwritten. You can edit and use the generated file in a regular shell outside of MATLAB. For example:

```
sh> tclsh compile_and_launch.tcl
```

'tclstart', 'tcl_commands'

Specifies one or more Tcl commands to execute before the Cadence Incisive simulator launches. Specify a command string or a cell array of command strings. You must specify at least one command; otherwise, no action occurs.

Note You must type `exec` in front of non-Tcl system shell commands. For example:

```
exec -ncverilog -c +access+rw +linedebug top.v  
hdlsimulink -gui work.top
```

Examples

The following function call sequence compiles the design and starts Simulink with a GUI from the "proj" folder with the model loaded. Simulink is instructed to communicate with the EDA Simulator Link interface on socket port 4449. All of these commands are specified in a single string as the property value to `tclstart`.

```
nclaunch(...
'tclstart',...
{'exec ncverilog -c +access+rw +linedebug top.v',...
'hdlstimulink -gui work.top'},...
'socketsimulink','4449',...
'rundir','/proj');
```

In this next example, `tclcmd` is used to build the sequence of Tcl commands that are executed in a Tcl shell after calling `nclaunch` from MATLAB, as follows:

- `tclcmd{1}` compiles `vlogtestbench_top`.
- `tclcmd{2}` elaborates the model.
- `tclcmd{3}` calls `hdlstimmatlab` in gui mode and loads the elaborated `vlogtestbench_top` in the simulator.

The function executes the arguments being passed with `-input` (`matlabtb` and `run`) in the `ncsim` Tcl shell. In this example, `matlabcp` associates the M-function `vlogmatlabcp` to the module instance `u_matlab_component`. It assumes that the `hdlldaemon` in MATLAB is listening on port 32864. The `run` function will run 50 resolution units (ticks).

```
tclcmd{1} = 'exec ncvlog vlogtestbench_top.v'
tclcmd{2} = 'exec ncelab -access +wc vlogtestbench_top'
tclcmd{3} = ['hdlstimmatlab -gui vlogtestbench_top ' ...
            '-input "{@matlabcp vlogtestbench_top.u_matlab_component...
            -mfunc vlogmatlabcp -socket 32864}" '...
            '-input "{@run 50}"']
nclaunch('hdlsimdir', 'local.IUS.glnx.tools.bin', 'tclstart',tclcmd);
```

The following example shows using the property `startupfile` to designate a Tcl script that the function then uses to start the HDL simulator from the Tcl shell.

In MATLAB:

```
nclaunch ('tclstart', 'xxx', 'startupfile', 'mytclscript',...  
         'starthdlsim', 'no')
```

In Tcl shell:

```
shell> tclsh mytclscript
```

Purpose	(For Incisive) End active MATLAB test bench and MATLAB component sessions
Syntax	nomatlabtb
Description	<p>The nomatlabtb command ends all active MATLAB test bench and MATLAB component sessions that were previously initiated by matlabtb or matlabcp commands.</p> <p>This command is issued in the HDL simulator.</p>
Examples	<p>The following command ends all MATLAB test bench and MATLAB component sessions:</p> <pre>> nomatlabtb</pre>
See Also	matlabtb, matlabcp

pingHdlSim

Purpose (For Incisive) Block cosimulation until HDL simulator is ready for simulation

Syntax
`pingHdlSim(timeout)`
`pingHdlSim(timeout, 'portnumber')`
`pingHdlSim(timeout, 'portnumber', 'hostname')`

Description `pingHdlSim(timeout)` blocks cosimulation by not returning until the HDL server loads or until the specified time-out occurs. `pingHdlSim` returns the process ID of the HDL simulator or -1 if a time-out occurs. You must enter a time-out value.

You may find this function useful if you are trying to automate a cosimulation and need to know that the HDL server has loaded before your script continues the simulation.

`pingHdlSim(timeout, 'portnumber')` tries to connect to the local host on port *portnumber* and times out after *timeout* seconds you specify.

`pingHdlSim(timeout, 'portnumber', 'hostname')` tries to connect to the host *hostname* on port *portname*. It times out after *timeout* seconds you specify.

Examples The following function call blocks further cosimulation until the HDL server loads or until 30 seconds have passed:

```
pingHdlSim(30)
```

If the server loads within 30 seconds, `pingHdlSim` returns the process ID. If it does not, `pingHdlSim` returns -1.

The following function call blocks further cosimulation on port 5678 until the HDL server loads or until 20 seconds have passed:

```
pingHdlSim(20, '5678')
```

The following function call blocks further cosimulation on port 5678 on host name msuser until the HDL server loads or until 20 seconds pass:

```
pingHdlSim(20, '5678', 'msuser')
```

tclHdlSim

Purpose (For Incisive) Execute Tcl command in HDL simulator

Syntax
`tclHdlSim(tclCmd)`
`tclHdlSim(tclCmd, 'portNumber')`
`tclHdlSim(tclCmd, 'portnumber', 'hostname')`

Description `tclHdlSim(tclCmd)` executes a Tcl command on the HDL simulator using a shared connection.

`tclHdlSim(tclCmd, 'portNumber')` executes a Tcl command on the HDL simulator by connecting to the local host on port *portNumber*.

`tclHdlSim(tclCmd, 'portnumber', 'hostname')` executes a Tcl command on the HDL simulator by connecting to the host *hostname* on port *portname*.

The HDL simulator must be connected to MATLAB using the EDA Simulator Link software for this function to work (see “Starting the HDL Simulator” on page 1-18.) If you start from within MATLAB, you must use `hdlsimmatlab`.

Examples The following function call displays a message in the HDL simulator command window using port 5678 on host name msuser:

```
tclHdlSim('puts "Done"', '5678', 'msuser')
```

See Also `hdldaemon`, `nclaunch`

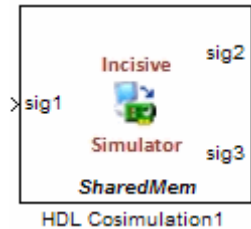
Blocks — Alphabetical List

HDL Cosimulation

Purpose (For Incisive) Cosimulate hardware component by communicating with HDL module instance executing in HDL simulator

Library EDA Simulator Link

Description The 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. You can use this block to model a source or sink device by configuring the block with input or output ports only.



The tabbed panes on the block's dialog box let you configure:

- Block input and output ports that correspond to signals (including internal signals) of an HDL module. You must specify a sample time for each output port; you can also specify a data type for each output port.
- Type of communication and communication settings used to exchange data between simulators.
- The timing relationship between units of simulation time in Simulink and the HDL simulator.
- Rising-edge or falling-edge clocks to apply to your model. You can specify the period for each clock signal.
- Tcl commands to run before and after the simulation.

The **Ports** pane provides fields for mapping signals of your HDL design to input and output ports in your block. The signals can be at any level of the HDL design hierarchy.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the HDL simulator. You can configure either of the following timing relationships:

- *Relative* timing relationship (Simulink seconds correspond to an HDL simulator-defined tick interval)

- *Absolute* timing relationship (Simulink seconds correspond to an absolute unit of HDL simulator time)

The **Connection** pane specifies the communications mode used between Simulink and the HDL simulator. If you use TCP socket communication, this pane provides fields for specifying a socket port and for the host name of a remote computer running the HDL simulator. The **Connection** pane also provides the option for bypassing the cosimulation block during Simulink simulation.

The **Clocks** pane lets you create optional rising-edge and falling-edge clocks that apply stimuli to your cosimulation model.

The **Tcl** pane provides a way of specifying tools command language (Tcl) commands to be executed before and after the HDL simulator simulates the HDL component of your Simulink model. You can use the **Pre-simulation commands** field on this pane for simulation initialization and startup operations, but you cannot use it to change simulation state.

Note You must make sure that signals being used in cosimulation have read/write access. (Verify such access through the HDL simulator—see product documentation for details). This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Dialog Box

The Block Parameters dialog box consists of the following tabbed panes of configuration options:

- “Ports Pane” on page 11-4
- “Connection Pane” on page 11-10
- “Timescales Pane” on page 11-14
- “Clocks Pane” on page 11-18
- “Tcl Pane” on page 11-21

Ports Pane

Specify fields for mapping signals of your HDL design to input and output ports in your block. Simulink deposits an input port signal on an HDL simulator signal at the signal's sample rate. Conversely, Simulink reads an output port signal from a specified HDL simulator signal at the specified sample rate.

In general, Simulink handles port sample periods as follows:

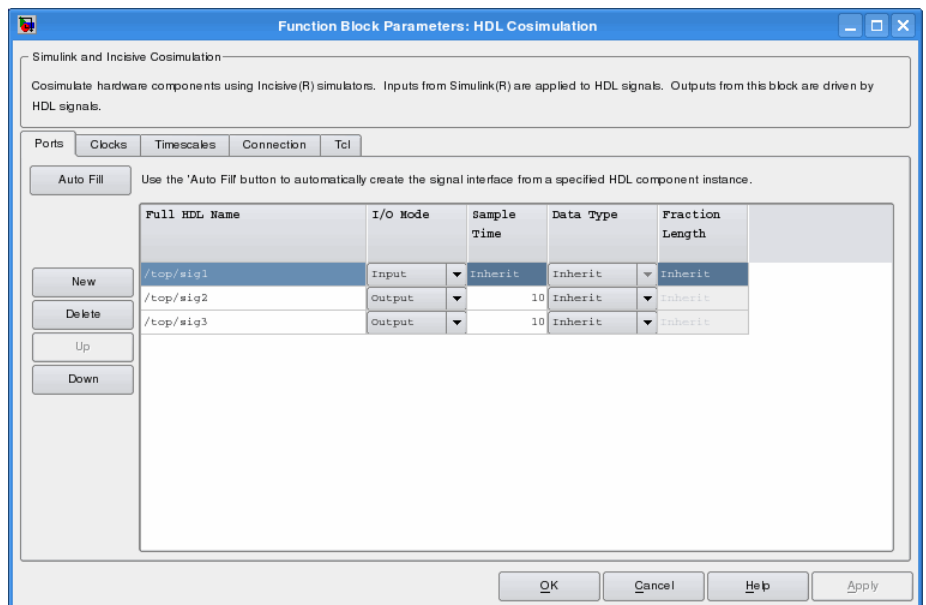
- If you connect an input port to a signal that has an explicit sample period, based on forward propagation, Simulink applies that rate to the port.
- If you connect an input port to a signal that does not have an explicit sample period, Simulink assigns a sample period that is equal to the least common multiple (LCM) of all identified input port sample periods for the model.
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. You must specify an explicit sample time for each output port.

In addition to specifying output port sample times, you can force the fixed-point data types on output ports. For example, setting the **Data Type** property of an 8-bit output port to **Signed** and setting its **Fraction Length** property to 5 would force the data type to `sfixed8_E5`. You can not force width; the width is always inherited from the HDL simulator.

Note The **Data Type** and **Fraction Length** properties apply only to the following signals:

- VHDL signals of any logic type, such as STD_LOGIC or STD_LOGIC_VECTOR
- Verilog signals of wire or reg type

You can set input/output ports in the Parts pane also. To do so, specify port as both input and output.



The list at the center of the pane displays HDL signals corresponding to ports on the HDL Cosimulation block.

Maintain this list with the buttons on the left of the pane:

HDL Cosimulation

- **Auto Fill** — Transmit a port information request to the HDL simulator. The port information request returns port names and information from an HDL model (or module) under simulation in the HDL simulator and automatically enters this information into the ports list. See “Obtaining Signal Information Automatically from the HDL Simulator” on page 4-18 for a detailed description of this feature.
- **New** — Add a new signal to the list and select it for editing.
- **Delete** — Remove a signal from the list.
- **Up** — Move the selected signal up one position in the list.
- **Down** — Move the selected signal down one position in the list.

To commit edits to the Simulink model, you must also click **Apply** after selecting parameter values.

Note When you import VHDL signals from the HDL simulator, EDA Simulator Link returns the signal names in all capitals.

To edit a signal name, double-click on the name. Set the signal properties on the same line and in the appropriate columns. The properties of a signal are as follows.

Full HDL Name

Specifies the signal path name, using the HDL simulator path name syntax. For example, a path name for an input port might be `manchester.samp`. The signal can be at any level of the HDL design hierarchy. The HDL Cosimulation block port corresponding to the signal is labeled with the **Full HDL Name**.

For rules on specifying signal/port and module path specifications in Simulink, see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-16.

Note You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field, using the standard copy and paste commands in the HDL simulator and Simulink. You must use the Path.Name view and not Db::Path.Name view. After pasting a signal path name into the **Full HDL Name** field, you must click the **Apply** button to complete the paste operation and update the signal list.

I/O Mode

Select either Input, Output, or both.

Input designates signals of your HDL module that Simulink will drive. Simulink deposits values on the specified the HDL simulator signal at the signal's sample rate.

Note When you define a block input port, make sure that only one source is set up to drive input to that signal. For example, you should avoid defining an input port that has multiple instances. If multiple sources drive input to a single signal, your simulation model may produce unexpected results.

Output designates signals of your HDL module that Simulink will read. For output signals, you must specify an explicit sample time. You can also specify any data type (except width). For details on specifying a data type, see Data Type and Fraction Length in a following section.

Because Simulink signals do not have the semantic of tri-states (there is no 'Z' value), you will gain no benefit by connecting to a bidirectional HDL signal directly. To interface with bidirectional signals, you can first interface to the input of the output driver, then the enable of the output driver and the output of the input driver. This approach leaves the actual tri-state buffer in HDL

where resolution functions can handle interfacing with other tri-state buffers.

Sample Time

This property becomes available only when you specify an output signal. You must specify an explicit sample time.

Sample Time represents the time interval between consecutive samples applied to the output port. The default sample time is 1. The exact interpretation of the output port sample time depends on the settings of the **Timescales** pane of the HDL Cosimulation block (see “Timescales Pane” on page 11-14). See also “Understanding the Representation of Simulation Time” on page 9-15.

Data Type

Fraction Length

These two related parameters apply only to output signals.

The **Data Type** property is enabled only for output signals. You can direct Simulink to determine the data type, or you can assign an explicit data type (with option fraction length). By explicitly assigning a data type, you can force fixed-point data types on output ports of an HDL Cosimulation block.

The **Fraction Length** property specifies the size, in bits, of the fractional part of the signal in fixed-point representation. **Fraction Length** becomes available if you do not set the **Data Type** property to Inherit.

The data type specification for an output port depends on the signal width and by the **Data Type** and **Fraction Length** properties of the signal.

Note The **Data Type** and **Fraction Length** properties apply only to the following signals:

- VHDL signals of any logic type, such as `STD_LOGIC` or `STD_LOGIC_VECTOR`
 - Verilog signals of wire or reg type
-

To assign a port data type, set the **Data Type** and **Fraction Length** properties as follows:

- Select **Inherit** from the **Data Type** list if you want Simulink to determine the data type.

This property defaults to **Inherit**. When you select **Inherit**, the **Fraction Length** edit field becomes unavailable.

Simulink always double checks that the word-length back propagated by Simulink matches the word length queried from the HDL simulator. If they do not match, Simulink generates an error message. For example, if you connect a Signal Specification block to an output, Simulink will force the data type specified by Signal Specification block on the output port.

If Simulink cannot determine the data type of the signal connected to the output port, it will query the HDL simulator for the data type of the port. As an example, if the HDL simulator returns the VHDL data type `STD_LOGIC_VECTOR` for a signal of size N bits, the data type `ufixN` is forced on the output port. (The implicit fraction length is 0.)

- Select **Signed** from the **Data Type** list if you want to explicitly assign a signed fixed point data type. When you select **Signed**, the **Fraction Length** edit field becomes available. EDA Simulator Link assigns the port a fixed-point type `sfixN_EnF`, where N is the signal width and F is the **Fraction Length**.

For example, if you specify **Data Type** as **Signed** and a **Fraction Length** of 5 for a 16-bit signal, Simulink forces the

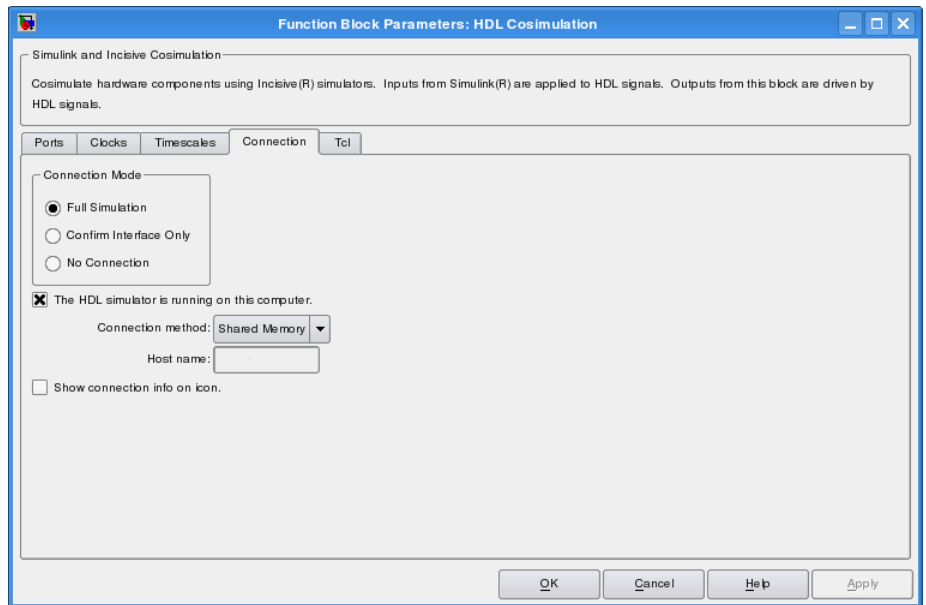
data type to `sfix16_En5`. For the same signal with a **Data Type** set to **Signed** and **Fraction Length** of -5, Simulink forces the data type to `sfix16_E5`.

- Select **Unsigned** from the **Data Type** list if you want to explicitly assign an unsigned fixed point data type. When you select **Unsigned**, the **Fraction Length** edit field becomes available. EDA Simulator Link assigns the port a fixed-point type `ufixN_EnF`, where N is the signal width and F is the **Fraction Length**.

For example, if you specify **Data Type** as **Unsigned** and a **Fraction Length** of 5 for a 16-bit signal, Simulink forces the data type to `ufix16_En5`. For the same signal with a **Data Type** set to **Unsigned** and **Fraction Length** of -5, Simulink forces the data type to `ufix16_E5`.

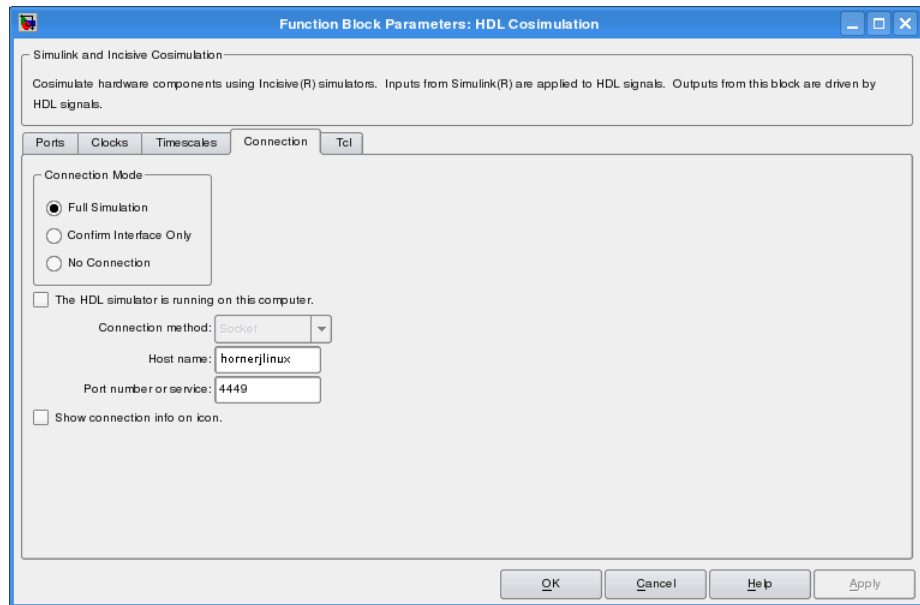
Connection Pane

This figure shows the default configuration of the **Connection** pane. The block defaults to a shared memory configuration for communication between Simulink and the HDL simulator, when they run on a single computer.



If you select TCP/IP socket mode communication, the pane displays additional properties, as shown in the following figure.

HDL Cosimulation



Connection Mode

If you want to bypass the HDL simulator when you run a Simulink simulation, use these 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:** Connect to the HDL simulator and check for proper signal names, dimensions, and data types, but do not run HDL simulation.
- **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

With the second and third options, the EDA Simulator Link cosimulation interface does not communicate with the HDL simulator during Simulink simulation.

The HDL Simulator is running on this computer

Select this option if you want to run Simulink and the HDL simulator on the same computer. When both applications run on the same computer, you have the choice of using shared memory or TCP sockets for the communication channel between the two applications. If you do not select this option, only TCP/IP socket mode is available, and the **Connection method** list becomes unavailable.

Connection method

This list becomes available when you select **The HDL Simulator is running on this computer**. Select **Socket** if you want Simulink and the HDL simulator to communicate via a designated TCP/IP socket. Select **Shared memory** if you want Simulink and the HDL simulator to communicate via shared memory. For more information on these connection methods, see “Communicating with MATLAB or Simulink and the HDL Simulator” on page 1-7.

Host name

If you run Simulink and the HDL simulator on different computers, this text field becomes available. The field specifies the host name of the computer that is running your HDL simulation in the HDL simulator.

Port number or service

Indicate a valid TCP socket port number or service for your computer system (if not using shared memory). For information on choosing TCP socket ports, see “Choosing TCP/IP Socket Ports” on page 8-16.

Show connection info on icon

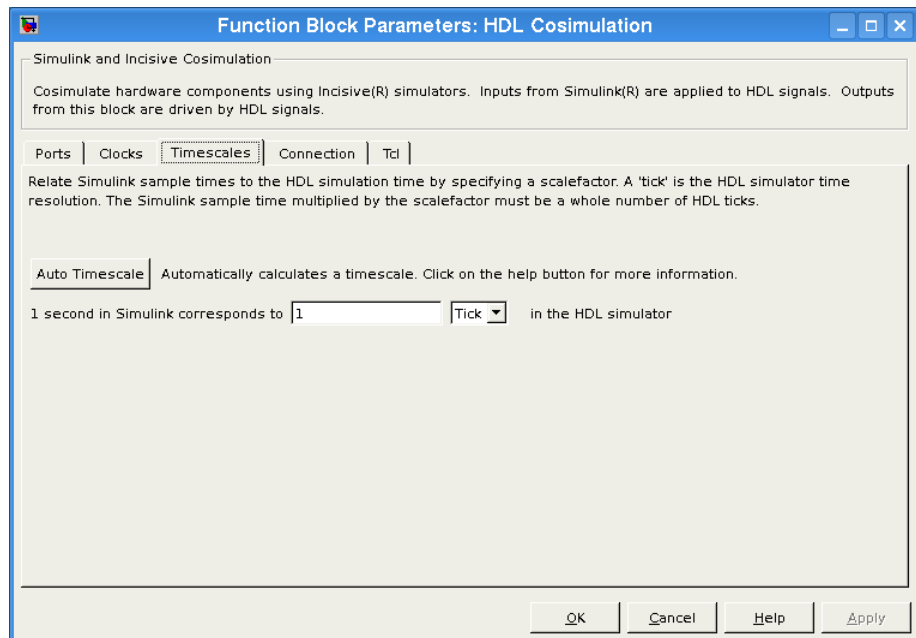
When you select this option, Simulink indicates information about the selected communication method and (if applicable) communication options information on the HDL Cosimulation block icon. If you select shared memory, the icon displays the string **SharedMem**. If you select TCP socket communication, the icon displays the string **Socket** and displays the host name and port number in the format **hostname:port**.

HDL Cosimulation

In a model that has multiple HDL Cosimulation blocks, with each communicating to different instances of the HDL simulator in different modes, this information helps to distinguish between different cosimulation sessions.

Timescales Pane

The **Timescales** pane of the HDL Cosimulation block parameters dialog box lets you choose a timing relationship between Simulink and the HDL simulator, either manually or automatically. The following figure shows the default settings of the **Timescales** pane.



The **Timescales** pane specifies 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:

- Using *relative timing mode*. EDA Simulator Link defaults to relative timing mode.
- Using *absolute timing mode*

For more information on calculating relative and absolute timing modes, see “Defining the Simulink and HDL Simulator Timing Relationship” on page 9-16.

For detailed information on the relationship between Simulink and the HDL simulator during cosimulation, and on the operation of relative and absolute timing modes, see “Understanding the Representation of Simulation Time” on page 9-15.

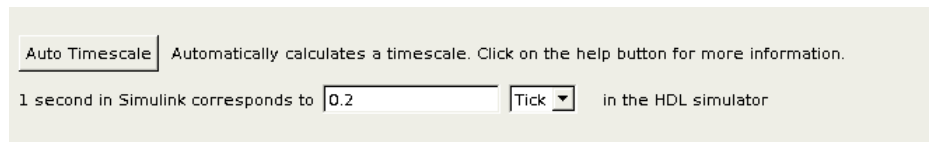
The following sections describe how to specify the timing relationship, either automatically or manually.

Automatically Specifying the Timing Relationship

You can have the EDA Simulator Link software calculate the timing relationship for you by performing the following steps:

- 1** Verify that the HDL simulator is running. EDA Simulator Link software can get the resolution limit of the HDL simulator only when that simulator is running.
- 2** Click on **Auto Timescale**.

The following graphic shows the result of clicking **Auto Timescale** in the **Timescales** pane of the HDL Cosimulation block in the Manchester Receiver demo.



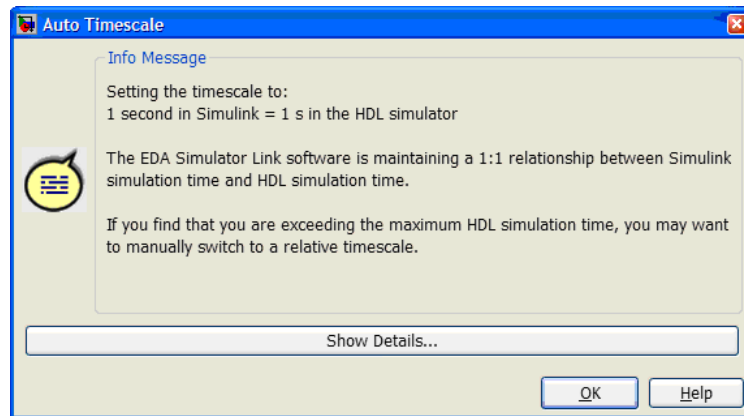
EDA Simulator Link software analyzes all the clock and port signal rates from the HDL Cosimulation block when it calculates the scale factor.

HDL Cosimulation

Note EDA Simulator Link cannot automatically calculate a sample timescale based on any signals driven via Tcl commands or in the HDL simulator. The link software cannot perform such calculations because it cannot know the rates of these signals.

The link software returns the sample rate in either seconds or ticks. If the results are in seconds, then the link software was able to resolve the timing differences in favor of fidelity (absolute time). If the results are in ticks, then the link software was best able to resolve the timing differences in favor of efficiency (relative time).

Each time you press Auto Timescale, the EDA Simulator Link software opens an informational GUI display that explains the results of Auto Timescale. If the link software cannot calculate a timescale for the given sample times, use the information in this dialog box to adjust your sample times.



Click **Show Details...** for information specific to your model's signals. Click **OK** to exit the informational dialog box.

3 Click **Apply** to commit your changes.

Note EDA Simulator Link does not support Auto Timescale calculated from frame-based signals.

For more on the timing relationship between the HDL simulator and Simulink, see “Understanding the Representation of Simulation Time” on page 9-15.

Manually Specifying a Relative Timing Relationship

To manually configure relative timing mode for a cosimulation, perform the following steps:

- 1 Select the **Timescales** tab of the HDL Cosimulation block parameters dialog box.
- 2 Verify that **Tick**, the default setting, is selected. If it is not, then select it from the list on the right.
- 3 Enter a scale factor in the text box on the left. The default scale factor is 1. For example, the next figure, shows the **Timescales** pane configured for a relative timing correspondence of 10 HDL simulator ticks to 1 Simulink second.



1 second in Simulink corresponds to Tick ▾ in the HDL simulator

- 4 Click **Apply** to commit your changes.

Manually Specifying an Absolute Timing Relationship

To manually configure absolute timing mode for a cosimulation, perform the following steps:

- 1 Select the **Timescales** tab of the HDL Cosimulation block parameters dialog box.

HDL Cosimulation

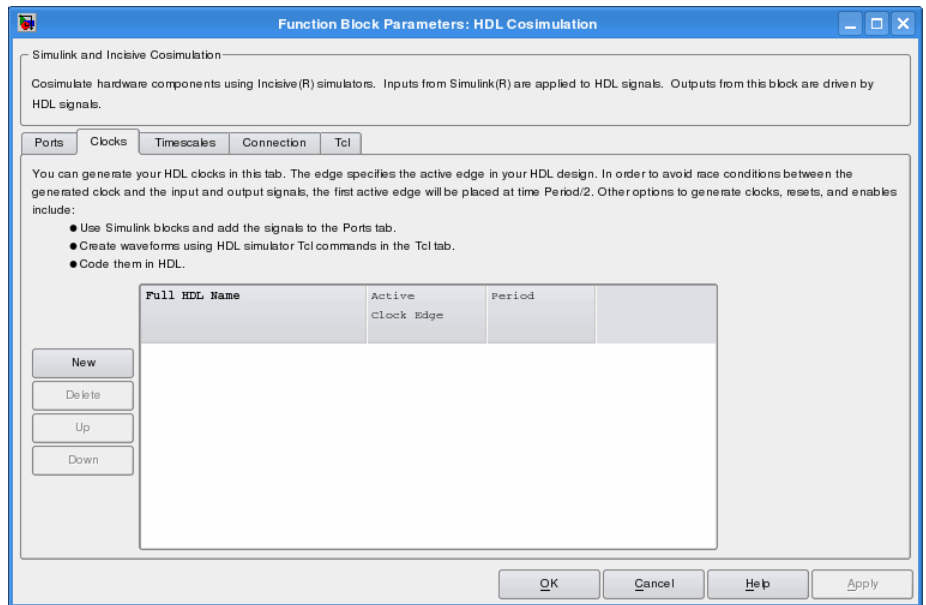
- 2 Select a unit of absolute time from the list on the right. The units available include fs (femtoseconds), ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), and s (seconds).
- 3 Enter a scale factor in the text box on the left. The default scale factor is 1. For example, in the next figure, the **Timescales** pane is configured for an absolute timing correspondence of 1 HDL simulator second to 1 Simulink second.

1 second in Simulink corresponds to in the HDL simulator

- 4 Click **Apply** to commit your changes.

Clocks Pane

You can create optional rising-edge and falling-edge clocks that apply stimuli to your cosimulation model. To do so, use the Clocks pane of the HDL Cosimulation block.



The scrolling list at the center of the pane displays HDL clocks that drive values to the HDL signals that you are modeling, using the deposit method.

Maintain the list of clock signals with the buttons on the left of the pane:

- **New** — Add a new clock signal to the list and select it for editing.
- **Delete** — Remove a clock signal from the list.
- **Up** — Move the selected clock signal up one position in the list.
- **Down** — Move the selected clock signal down one position in the list.

To commit edits to the Simulink model, you must also click **Apply**.

A clock signal has the following properties.

HDL Cosimulation

Full HDL Name

Specify each clock as a signal path name, using the HDL simulator path name syntax. For example: `manchester.clk`.

For information about and requirements for path specifications in Simulink, see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 4-16.

Note You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field, using the standard copy and paste commands in the HDL simulator and Simulink. You must use the Path.Name view and not Db::Path.Name view. After pasting a signal path name into the **Full HDL Name** field, you must click the **Apply** button to complete the paste operation and update the signal list.

Edge

Select **Rising** or **Falling** to specify either a rising-edge clock or a falling-edge clock.

Period

You must either specify the clock period explicitly or accept the default period of 2.

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. Instead, the EDA Simulator Link software creates the falling edge at

$$\text{clockperiod} / 2$$

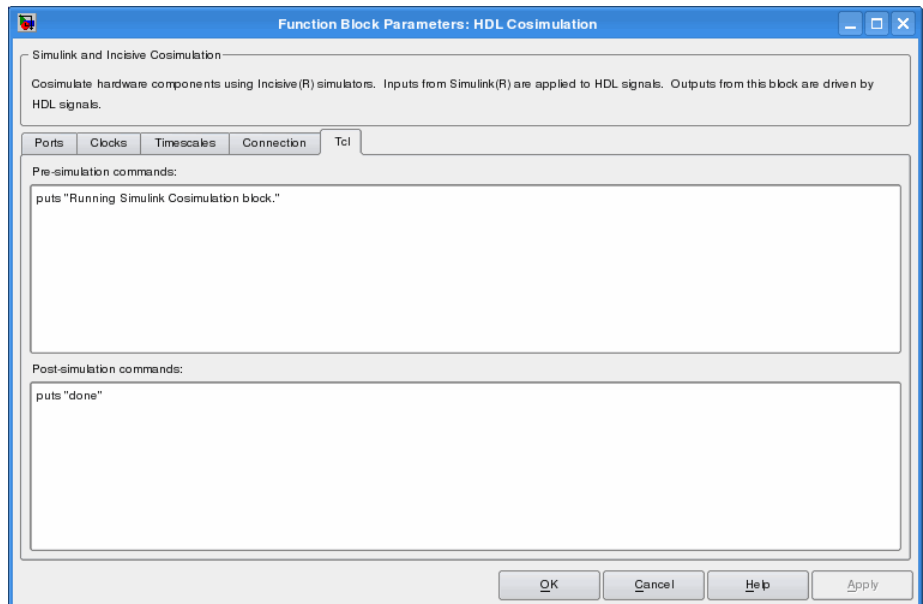
(rounded down to the nearest integer).

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

For instructions on adding and editing clock signals, see “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block” on page 4-30.

Tcl Pane

Specify tools command language (Tcl) commands to be executed before and after the HDL simulator simulates the HDL component of your Simulink model.



Pre-simulation commands

Contains Tcl commands to be executed before the HDL simulator simulates the HDL component of your Simulink model. You can

HDL Cosimulation

specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), the standard Tcl concatenation operator.

Alternatively, you can create an HDL simulator Tcl script that lists Tcl commands and then specify that file with the HDL simulator source command as follows:

```
source mycosimstartup.script_extension
```

Use of this field can range from something as simple as a one-line echo command to confirm that a simulation is running to a complex script that performs an extensive simulation initialization and startup sequence.

Note The command string or Tcl script that you specify for this parameter cannot include commands that load an HDL simulator project or modify simulator state. For example, neither can include commands such as run, stop, or reset.

Post-simulation commands

Contains Tcl commands to be executed after the HDL simulator simulates the HDL component of your Simulink model. 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 (;), the standard Tcl concatenation operator.

Alternatively, you can create an HDL simulator Tcl script that lists Tcl commands and then specify that file with the HDL simulator source command as follows:

```
source mycosimcleanup.script_extension
```

You can include the exit command in an after-simulation Tcl script to force the HDL simulator to shut down at the end of a

cosimulation session. To ensure that all other after-simulation Tcl commands specified for the model will execute, specify all after simulation Tcl commands in a single cosimulation block and place `exit` at the end of the command string or Tcl script.

The following example shows a Tcl script when the `-gui` argument was used with `hdlsimmatlab` or `hdlsimulink`:

```
after 1000 {ncsim -submit exit}
```

This next example is of a Tcl exit script to use when the `-tcl` argument was used with `hdlsimmatlab` or `hdlsimulink`:

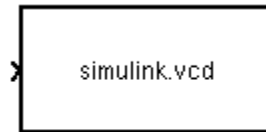
```
after 1000 {exit}
```

Note With the exception of `exit`, the command string or Tcl script that you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, they cannot include commands such as `run`, `stop`, or `reset`.

To VCD File

Purpose (For Incisive) Generate value change dump (VCD) file

Library



Description To VCD File

The To VCD File block generates a VCD file that contains information about changes to signals connected to the block's input ports and names the file with the specified file name. You can use VCD files during design verification in the following ways:

- 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

In addition, VCD files include data that can be graphically displayed or analyzed with postprocessing tools. Examples of postprocessing include the extraction of data pertaining to a particular section of a design hierarchy or data generated during a specific time interval.

Using the Block Parameters dialog box, you can specify the following parameters:

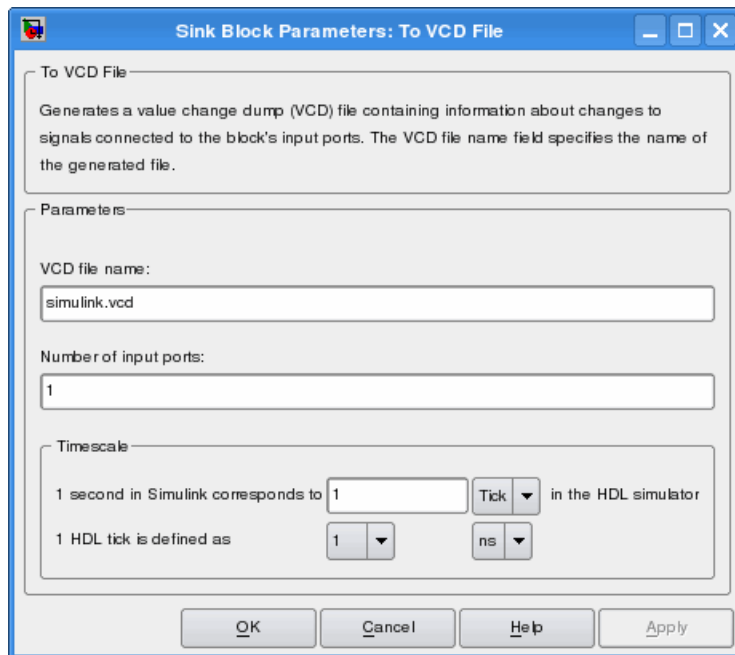
- The file name to be used for the generated file
- The number of block input ports that are to receive signal data
- The timescale to relate Simulink sample times with HDL simulator ticks

VCD files can grow very large for larger designs or smaller designs with longer simulation runs. However, the only limitation on the size of a

VCD file generated by the To VCD File block is the maximum number of signals (and symbols) supported, which is 94^3 (830,584).

For a description of the VCD file format, see “VCD File Format” on page 11-27.

Note The toVCD File block is integrated into the Simulink Signal & Scope Manager. See the *Simulink User's Guide* for more information on using the Signal & Scope Manager.



Dialog Box

VCD file name

The file name to be used for the generated VCD file. 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 specify the same name for multiple To VCD File blocks, Simulink automatically adds a numeric postfix to identify each instance uniquely.

Note If you want the generated file to have a .vcd file type extension, you must specify it explicitly.

Do not give the same file name to different VCD blocks. Doing so results in invalid VCD files.

Number of input ports

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.

In some cases, a single input port maps to multiple signals (and symbols). This multiple mapping occurs when the input port receives a multidimensional signal.

Because the VCD specification does not include multidimensional signals, Simulink flattens them to a 1D vector in the file.

Timescale

Choose an optimal timing relationship between Simulink and the HDL simulator.

The timescale options specify a correspondence between one second of Simulink time and some quantity of HDL simulator time. You can express this quantity of HDL simulator time in one of the following ways:

- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation operates in *relative timing mode*, which is the timing mode default.

To use relative mode, select **Tick** from the pop-up list at the label **in the HDL simulator**, and enter the desired number of ticks in the edit box at **1 second in Simulink corresponds to**. The default value is 1 Tick.

- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation operates in *absolute timing mode*.

To use absolute mode, select the desired resolution unit from the pop-up list at the label **in the HDL simulator** (available units are **fs, ps, ns, us, ms, s**), and enter the desired number of resolution units in the edit box at **1 second in Simulink corresponds to**. Then, set the value of the HDL simulator tick by selecting 1, 10, or 100 from the pop-up list at **1 HDL Tick is defined as** and the resolution unit from the pop-up list at **defined as**.

VCD File Format

The format of generated VCD files adheres to IEEE Std 1364-2001. The following table describes the format.

Generated VCD File Format

File Content	Description
<pre>\$date 23-Sep-2003 14:38:11 \$end</pre>	Data and time the file was generated.
<pre>\$version EDA Simulator Link version 1.0 \$ end</pre>	Version of the VCD block that generated the file.
<pre>\$timescale 1 ns \$ end</pre>	The time scale that was used during the simulation.

Generated VCD File Format (Continued)

File Content	Description
<code>\$scope module manchestermodel \$end</code>	The scope of the module being dumped.
<code>\$var wire 1 ! Original Data [0] \$end</code> <code>\$var wire 1 " Recovered Clock [0] \$end</code> <code>\$var wire 1 # Recovered Data [0] \$end</code> <code>\$var wire 1 \$ Data Validity [0] \$end</code>	Variable definitions. Each definition associates a signal with character identification code (symbol). The symbols are derived from printable characters in the ASCII character set from ! to ~. Variable definitions also include the variable type (wire) and size in bits.
<code>\$upscope \$end</code>	Marks a change to the next higher level in the HDL design hierarchy.
<code>\$enddefinitions \$end</code>	Marks the end of the header and definitions section.
<code>#0</code>	Simulation start time.

Generated VCD File Format (Continued)

File Content	Description
<pre>\$dumpvars 0! 0" 0# 0\$ \$end</pre>	<p>Lists the values of all defined variables at time equals 0.</p>
<pre>#630 1!</pre>	<p>The starting point of logged value changes from checks of variable values made at each simulation time increment.</p> <p>This entry indicates that at 63 nanoseconds, the value of signal <code>Original Data</code> changed from 0 to 1.</p>
<pre>. . . #1160 1# 1\$</pre>	<p>At 116 nanoseconds the values of signals <code>Recovered Data</code> and <code>Data Validity</code> changed from 0 to 1.</p>
<pre>\$dumpoff x! x" x# x\$</pre>	<p>Marks the end of the file by dumping the values of all variables as the value x.</p>

To VCD File

Generated VCD File Format (Continued)

File Content	Description
\$end	

A

- Absolute timing mode
 - defining the timing relationship with Simulink 9-20
- addresses, Internet 8-15
- application software 1-9
- application specific integrated circuits (ASICs) 1-2
- applications 1-2
 - coding for EDA Simulator Link™ software 2-12 3-8
 - component
 - coding for EDA Simulator Link™ software 3-5
 - programming with EDA Simulator Link™ software 3-5
 - test bench
 - coding for EDA Simulator Link™ software 2-2
 - programming with EDA Simulator Link™ software 2-2
- arguments
 - for hdlSimatlab command 10-9
 - for hdlSimulink command 10-10
 - for matlabcp command 10-12
 - for matlabb command 10-25
 - for matlabbtbeval command 10-38
 - for pingHdlSim function 10-48
 - for tclHdlSim function 10-50
- array data types
 - conversions of 9-5
 - VHDL 2-8
- array indexing
 - differences between MATLAB and VHDL 9-5
- arrays
 - converting to 9-10
 - indexing elements of 9-5
 - of VHDL data types 2-8
- ASICs (application specific integrated circuits) 1-2

- Auto fill
 - in Ports pane of HDL Cosimulation block 11-2
 - using in Ports pane 4-15

B

- behavioral model 1-2
- BIT data type 2-8
 - conversion of 9-5
 - converting to 9-10
- bit vectors
 - converting for MATLAB 9-9
 - converting to 9-10
- BIT_VECTOR data type 2-8
 - conversion of 9-5
 - converting for MATLAB 9-9
 - converting to 9-10
- block input ports parameter
 - description of 11-2
- Block input ports parameter
 - mapping signals with 4-15
- block latency 9-25
- block library
 - description of 4-8
- block output ports parameter
 - description of 11-2
- Block output ports parameter
 - mapping signals with 4-15
- block parameters
 - setting programmatically 4-37
- Block Parameters dialog
 - for HDL Cosimulation block 4-14
- block ports
 - mapping signals to 4-15
- blocks
 - HDL Cosimulation
 - description of 11-2
 - To VCD File
 - description of 11-24

- blocksets
 - for creating hardware models 4-5
 - for EDA applications 4-5
 - installing 1-12
- breakpoints
 - setting in MATLAB 2-35
- bypass
 - HDL Cosimulation block 4-34
- C**
- Cadence Incisive®
 - in EDA Simulator Link™ cosimulation environment 1-4
 - working with MATLAB links to 1-4
- callback specification 7-1
- callback timing 2-32
 - cancel option 10-25
- CHARACTER data type 2-8
 - conversion of 9-5
- client
 - for MATLAB and HDL simulator links 1-4
 - for Simulink and HDL simulator links 1-6
- client/server environment
 - MATLAB and HDL simulator 1-4
 - Simulink and HDL simulator 1-6
- clocks
 - specifying for HDL Cosimulation blocks 4-30
- Clocks pane
 - configuring block clocks with 4-30
 - description of 11-2
- column-major numbering 9-5
- comm status field
 - checking with hdd daemon function 2-34
- communication
 - configuring for blocks 4-34
 - modes of 1-7
 - socket ports for 8-15
- communication channel
 - checking identifier for 2-34
- communication modes
 - checking 2-34
 - specifying for Simulink links 4-12
 - specifying with hdd daemon function 2-22
- Communications Blockset
 - as optional software 1-9
 - using for EDA applications 4-5
- compilation, VHDL code 2-10
- compiler, VHDL 2-10
- component applications
 - coding for EDA Simulator Link™ software overview of 3-5
 - programming with EDA Simulator Link™ software overview of 3-5
- component function
 - associating with HDL module 3-13
 - matlabcp 3-8
 - programming for HDL verification 2-12
- component functions
 - adding to MATLAB search path 3-10
 - coding for HDL visualization 3-8
 - naming 3-13
 - scheduling invocation of 2-30
- configuration file
 - using with Cadence Incisive simulatorsncsim 1-20
- configurations
 - deciding on for MATLAB 8-12
 - deciding on for Simulink 8-13
 - MATLAB
 - multiple-link 8-12
 - Simulink
 - multiple-link 8-13
 - single-system for MATLAB 8-12
 - single-system for Simulink 8-13
 - valid for MATLAB and HDL simulator 8-12
 - valid for Simulink and HDL simulator 8-13
- Connection pane
 - configuring block communication with 4-34

- description of 11-2
- connections status field
 - checking with `hdlldaemon` function 2-34
- connections, link
 - checking number of 2-34
 - TCP/IP socket 8-15
- continuous signals 9-15
- cosimulation
 - bypassing 4-34
 - controlling MATLAB
 - overview of 2-34
 - loading HDL modules for 4-12
 - logging changes to signal values during 6-2
 - starting MATLAB
 - overview of 2-34
 - starting with Simulink 4-43
- cosimulation environment
 - MATLAB and HDL simulator 1-4
 - Simulink and HDL simulator 1-6
- cosimulation output ports
 - specifying 4-27
- Cosimulation timing
 - absolute mode 11-2
 - relative mode 11-2

D

- data types
 - conversions of 9-5
 - converting for MATLAB 9-9
 - converting for the HDL simulator 9-10
 - HDL port
 - verifying 7-7
 - unsupported VHDL 2-8
 - VHDL port 2-8
- `dec2mv1` function
 - description of 10-2
- delta time 9-25
- demos 1-23
 - for EDA Simulator Link™ 1-21

- deposit
 - changing signals with 9-14
 - for `iport` parameter 7-1
 - with `force` commands 2-37
- design process, hardware 1-2
- dialogs
 - for HDL Cosimulation block 11-2
 - for To VCD File block 11-24
- discrete blocks 9-15
- `do` command 4-36
- DO files
 - specifying for HDL Cosimulation blocks 4-36
- documentation
 - overview 1-21
- double values
 - as representation of time 2-32
 - converting for MATLAB 9-9
 - converting for the HDL simulator 9-10
- `dspstartup` M-file 4-41
- duty cycle 4-30

E

- EDA (Electronic Design Automation) 1-2
- EDA Simulator Link™
 - default libraries 1-13
- EDA Simulator Link™ libraries
 - using 1-13
- EDA Simulator Link™ software
 - block library
 - using to add HDL to Simulink software
 - with 4-8
 - definition of 1-2
 - installing 1-12
 - setting up the HDL simulator for 8-2
- Electronic Design Automation (EDA) 1-2
- entities
 - coding for MATLAB verification 2-7
 - coding for MATLAB visualization 3-7
 - loading for cosimulation with Simulink 4-12

- sample definition of 2-10
- entities or modules
 - getting port information of 7-1
- enumerated data types 2-8
 - conversion of 9-5
 - converting to 9-10
- environment
 - cosimulation with MATLAB and HDL simulator 1-4
 - cosimulation with Simulink and HDL simulator 1-6
- examples 4-5
 - dec2mvl function 10-2
 - hdlsimmatlab command 10-9
 - hdlsimulink command 10-10
 - matlabcp command 10-12
 - matlabtb command 10-25
 - matlabtbeval command 10-38
 - mv12dec function 10-41
 - nclaunch function 10-42
 - nomatlabtb command 10-47
 - pingHdlSim function 10-48
 - tc1HdlSim function 10-50
 - test bench function 2-13
 - See also* Manchester receiver Simulink model

F

- falling option 10-25
- falling-edge clocks
 - creating for HDL Cosimulation blocks 4-30
 - description of 11-2
 - specifying as scheduling options 2-32
- Falling-edge clocks parameter
 - specifying block clocks with 4-30
- field programmable gate arrays (FPGAs) 1-2
- files
 - VCD 11-27
- force command
 - applying simulation stimuli with 2-37

- resetting clocks during cosimulation
 - with 4-43
- FPGAs (field programmable gate arrays) 1-2
- Frame-based processing 8-22
 - in cosimulation 8-22
 - performance improvements gained from 8-22
 - requirements for use of 8-22
 - restrictions on use of 8-22
- functions 10-1
 - hdlsimmatlab
 - description of 10-9
 - loading HDL modules for verification with 2-24
 - loading HDL modules for visualization with 3-12
 - hdlsimulink
 - description of 10-10
 - matlabcp
 - description of 10-12
 - matlabtb
 - description of 10-25
 - matlabtbeval
 - description of 10-38
 - nomatlabtb 10-47
 - resolution 9-14
 - vsimulinkhdlsimulink
 - loading HDL modules for cosimulation with 4-12
- See also* MATLAB functions

H

- hardware description language (HDL). *See* HDL
- hardware design process 1-2
- hardware model design
 - creating in Simulink 4-5
 - running and testing in Simulink 4-10
- HDL (hardware description language) 1-2
- HDL cosimulation block
 - configuring ports for 4-15

- opening Block Parameters dialog for 4-14
- HDL Cosimulation block
 - adding to a Simulink model 4-8
 - black boxes representing 4-5
 - bypassing 4-34
 - configuration requirements for 8-13
 - configuring clocks for 4-30
 - configuring communication for 4-34
 - configuring Tcl commands for 4-36
 - description of 11-2
 - design decisions for 4-5
 - handling of signal values for 4-41
 - in EDA Simulator Link™ environment 1-6
 - scaling simulation time for 9-15
 - valid configurations for 8-13
- HDL entities
 - loading for cosimulation with Simulink 4-12
- HDL models 1-2
 - adding to Simulink models 4-8
 - compiling 2-10
 - configuring Simulink for 4-41
 - cosimulation 1-2
 - debugging 2-10
 - porting 6-2
 - running in Simulink 4-43
 - testing in Simulink 4-43
 - verifying 1-2
 - See also* VHDL models
- HDL module
 - associating with component function 3-13
 - associating with test bench function 2-28
- HDL modules
 - coding for MATLAB verification 2-7
 - coding for MATLAB visualization 3-7
 - getting port information of 7-1
 - loading for verification 2-24
 - loading for visualization 3-12
 - naming 2-7
 - using port information for 7-7
 - validating 7-7
 - verifying port direction modes for 7-7
- HDL simulator
 - handling of signal values for 4-41
 - simulation time for 9-15
 - starting 1-18
 - starting for use with Simulink 4-12
- HDL simulator commands
 - force**
 - applying simulation stimuli with 2-37
 - resetting clocks during cosimulation with 4-43
 - run** 2-35
- HDL simulator running on this computer
 - parameter description of 11-2
- HDL simulators
 - in EDA Simulator Link™ environment 1-6
 - installing 1-12
 - invoking for use with EDA Simulator Link™ software 1-18
 - setting up during installation 8-2
 - specifying a specific executable or version 1-18
 - starting from MATLAB 2-24
 - working with Simulink links to 1-6
- hdldaemon function
 - checking link status of 2-34
 - configuration restrictions for 8-12
 - starting 2-22
- hdlsimdir property
 - with nclaunch function 10-42
- hdlsimmatlab command
 - description of 10-9
 - loading HDL modules for verification with 2-24
 - loading HDL modules for visualization with 3-12
- hdlsimulink command
 - description of 10-10
- help

- for EDA Simulator Link™ software 1-21
- Host name parameter
 - description of 11-2
 - specifying block communication with 4-34
- host names
 - identifying HDL simulator server 4-34
 - identifying server with 8-15

I

- IN direction mode 2-8
 - verifying 7-7
- Incisive or NC simulators
 - as required software 1-9
- Incisive simulator commands
 - hdlsimmatlab
 - description of 10-9
- INOUT direction mode 2-8
 - verifying 7-7
- INOUT ports
 - specifying 11-2
- input 2-8
 - See also* input ports
- input ports
 - attaching to signals 9-14
 - for HDL model 2-8
 - for MATLAB component function 7-1
 - for MATLAB test bench function 7-1
 - for test bench function 7-1
 - mapping signals to 4-15
 - simulation time for 9-15
- installation
 - of EDA Simulator Link™ software 1-12
 - of related software 1-12
- installation of EDA Simulator Link™ 1-12
- int64 values 2-32
- INTEGER data type 2-8
 - conversion of 9-5
 - converting to 9-10
- Internet address 8-15

- identifying server with 8-15
- interprocess communication identifier 2-34
- ipc_id status field
 - checking with hlldaemon function 2-34
- ipport parameter 7-1

L

- latency
 - block 9-25
 - clock signal 4-44
- link status
 - checking MATLAB server 2-34
- links
 - MATLAB and HDL simulator 1-4
 - Simulink and HDL simulator 1-6

M

- MATLAB
 - as required software 1-9
 - in EDA Simulator Link™ cosimulation
 - environment 1-4
 - installing 1-12
 - quitting 2-39
 - working with HDL simulator links to 1-4
- MATLAB component functions
 - adding to MATLAB search path 3-10
 - defining 7-1
 - specifying required parameters for 7-1
- MATLAB data types
 - conversion of 9-5
- MATLAB functions 10-1
 - coding for HDL verification 2-12
 - coding for HDL visualization 3-8
 - dec2mv1
 - description of 10-2
 - defining 7-1
 - hlldaemon 2-22
 - mv12dec

- description of 10-41
- naming 2-28
- nclaunch 4-12
 - description of 10-42
- pingHdlSim
 - description of 10-48
- programming for HDL verification 2-12
- programming for HDL visualization 3-8
- sample of 2-13
- scheduling invocation of 2-32
- specifying required parameters for 7-1
- tc1HdlSim
 - description of 10-50
- test bench 1-4
- which 2-21
- MATLAB link sessions
 - controlling
 - overview 2-34
 - starting
 - overview 2-34
- MATLAB search path 2-21
- MATLAB server
 - checking link status with 2-34
 - configuration restrictions for 8-12
 - configurations for 8-12
 - function for invoking 1-4
 - identifying in a network configuration 8-15
 - starting 2-22
- MATLAB test bench functions
 - defining 7-1
 - specifying required parameters for 7-1
- matlabcp command
 - description of 10-12
- matlabtb command
 - description of 10-25
 - specifying scheduling options with 2-32
- matlabtbeval command
 - description of 10-38
 - specifying scheduling options with 2-32
- mfunc option

- with matlabcp command 10-12
- with matlabtb command 10-25
- with matlabtbeval command 10-38
- models
 - compiling VHDL 2-10
 - debugging VHDL 2-10
- modes
 - communication 2-22
 - port direction 7-7
- module names
 - specifying paths
 - for MATLAB link sessions 2-26
 - in Simulink 4-16
- modules
 - coding for MATLAB verification 2-7
 - coding for MATLAB visualization 3-7
 - loading for verification 2-24
 - loading for visualization 3-12
 - naming 2-7
- multirate signals 9-24
- mv12dec function
 - description of 10-41

N

- names
 - for component functions 3-13
 - for HDL modules 2-7
 - for test bench functions 2-28
 - shared memory communication channel 2-34
 - verifying port 7-7
- NATURAL data type 2-8
 - conversion of 9-5
 - converting to 9-10
- nclaunch
 - using 1-18
- nclaunch function
 - description of 10-42
 - starting HDL simulator with 2-24
- ncsim

- for the Cadence Incisive simulators
 - using configuration file with 1-20
- network configuration 8-15
- nomatlabtb command 10-47
- Number of input ports parameter 11-24
- Number of output ports parameter
 - description of 11-24
- numeric data
 - converting for MATLAB 9-9
 - converting for the HDL simulator 9-10

O

- online help
 - where to find it 1-21
- oport parameter 7-1
- options
 - for hdlstimulink command 10-10
 - for matlabcp command 10-12
 - for matlabtb command 10-25
 - for matlabtbeval command 10-38
 - property
 - with nclaunch function 10-42
- OS platform. *See* EDA Simulator Link™ product requirements page on The MathWorks web site
- OUT direction mode 2-8
 - verifying 7-7
- output ports
 - for HDL model 2-8
 - for MATLAB component function 7-1
 - for MATLAB test bench function 7-1
 - for test bench function 7-1
 - mapping signals to 4-15
 - simulation time for 9-15
- Output sample time parameter
 - description of 11-2
 - specifying sample time with 4-15

P

- parameters
 - for HDL Cosimulation block 11-2
 - for To VCD File block 11-24
 - required for MATLAB component functions 7-1
 - required for MATLAB test bench functions 7-1
 - required for test bench functions 7-1
 - setting programmatically 4-37
- path specification
 - for ports/signals and modules
 - for MATLAB link sessions 2-26
 - in Simulink 4-16
 - for ports/signals and modules in Simulink
 - with HDL Cosimulation block 11-2
- phase, clock 4-30
- pingHdlSim function
 - description of 10-48
- platform support
 - required 1-9
- port names
 - specifying paths
 - for MATLAB link sessions 2-26
 - in Simulink 4-16
 - specifying paths in Simulink
 - with HDL Cosimulation block 11-2
 - verifying 7-7
- Port number or service parameter
 - description of 11-2
 - specifying block communication with 4-34
- port numbers 8-15
 - checking 2-34
 - specifying for MATLAB server 2-22
 - specifying for the HDL simulator 2-32
- portinfo parameter 7-1
- portinfo structure 7-7
- ports
 - getting information about 7-1
 - specifying direction modes for 2-8

- specifying VHDL data types for 2-8
 - using information about 7-7
 - verifying data type of 7-7
 - verifying direction modes for 7-7
 - Ports pane
 - Auto fill option 11-2
 - configuring block ports with 4-15
 - description of 11-2
 - using Auto fill 4-15
 - ports, block
 - mapping signals to 4-15
 - Post- simulation command parameter
 - specifying block Tcl commands with 4-36
 - Post-simulation command parameter
 - description of 11-2
 - postprocessing tools 6-2
 - Pre-simulation command parameter
 - specifying block simulation Tcl commands with 4-36
 - prerequisites
 - for using EDA Simulator Link™ software 1-9
 - properties
 - for nclaunch function 10-42
 - for starting HDL simulator for use with Simulink 4-12
 - for starting MATLAB server 2-22
 - nclaunchdir
 - with nclaunch function 10-42
 - socketsimulink 10-42
 - startupfile 10-42
 - tclstart
 - with nclaunch function 10-42
 - property option
 - for nclaunch function 10-42
 - Prsimulation command parameter
 - description of 11-2
- R**
- race conditions
 - in HDL simulation 4-44
 - rate converter 9-24
 - real data
 - converting for MATLAB 9-9
 - converting for the HDL simulator 9-10
 - REAL data type 2-8
 - conversion of 9-5
 - converting to 9-10
 - real values, as time 2-32
 - relative timing mode
 - definition of 9-18
 - operation of 9-18
 - repeat option 10-12
 - requirements
 - application software 1-9
 - checking product 1-9
 - platform 1-9
 - resolution functions 9-14
 - resolution limit 7-7
 - rising option 10-12
 - rising-edge clocks
 - creating for HDL Cosimulation blocks 4-30
 - description of 11-2
 - specifying as scheduling options 2-32
 - Rising-edge clocks parameter
 - specifying block clocks with 4-30
 - run command 2-35
- S**
- sample periods 4-5
 - See also* sample times
 - sample times 9-25
 - design decisions for 4-5
 - handling across simulation domains 4-41
 - specifying for block output ports 4-15
 - Sample-based processing 8-22
 - scalar data types
 - conversions of 9-5
 - VHDL 2-8

- scheduling options 2-32
 - component sessions 2-30
 - test bench sessions 2-30
- script
 - HDL simulator setup 8-2
- search path 2-21
- sensitivity lists 2-32
- sensitivity option 10-12
- server, MATLAB
 - checking link status of MATLAB 2-34
 - for MATLAB and HDL simulator links 1-4
 - for Simulink and HDL simulator links 1-6
 - identifying in a network configuration 8-15
 - starting MATLAB 2-22
- Set/Clear Breakpoint option, MATLAB 2-35
- set_param
 - for specifying post-simulation Tcl commands 4-36
- shared memory communication 1-7
 - as a configuration option for MATLAB 8-12
 - as a configuration option for Simulink 8-13
 - for Simulink applications 4-12
 - specifying for HDL Cosimulation blocks 4-34
 - specifying with hdldaemon function 2-22
- Shared memory parameter
 - description of 11-2
 - specifying block communication with 4-34
- signal data types
 - specifying 4-27
- signal names
 - specifying paths
 - for MATLAB link sessions 2-26
 - in Simulink 4-16
 - specifying paths in Simulink
 - with HDL Cosimulation block 11-2
- signal path names
 - displaying 4-15
 - specifying for block clocks 4-30
 - specifying for block ports 4-15
- Signal Processing Blockset
 - as optional software 1-9
 - using for EDA applications 4-5
- signals
 - continuous 9-15
 - defining ports for 2-8
 - driven by multiple sources 9-14
 - exchanging between simulation domains 4-41
 - handling across simulation domains 4-41
 - how Simulink drives 9-14
 - logging changes to 6-2
 - logging changes to values of 6-2
 - mapping to block ports 4-15
 - multirate 9-24
 - read/write access
 - mapping 4-15
 - required 9-14
 - read/write access required 11-2
- signed data 9-9
- SIGNED data type 9-10
- simulation analysis 6-2
- simulation time 7-1
 - guidelines for 9-15
 - representation of 9-15
 - scaling of 9-15
- simulations
 - comparing results of 6-2
 - ending 2-39
 - logging changes to signal values during 6-2
 - quitting 2-39
- simulator communication
 - options 4-34
- simulator resolution limit 7-7
- simulators
 - handling of signal values between 4-41
 - HDL
 - starting from MATLAB 2-24
- Simulink
 - as optional software 1-9
 - configuration restrictions for 8-13

- configuring for HDL models 4-41
- creating hardware model designs with 4-5
- driving cosimulation signals with 9-14
- in EDA Simulator Link™ environment 1-6
- installing 1-12
- running and testing hardware model in 4-10
- simulation time for 9-15
- starting the HDL simulator for use with 4-12
- working with HDL simulator links to 1-6
- Simulink Fixed Point
 - as optional software 1-9
 - using for EDA applications 4-5
- Simulink models
 - adding HDL models to 4-8
- sink device
 - adding to a Simulink model 4-8
 - specifying block ports for 4-15
 - specifying clocks for 4-30
 - specifying communication for 4-34
 - specifying Tcl commands for 4-36
- socket numbers 2-34
 - See also* port numbers
- socket option
 - with `hdlsimulink` command 10-10
 - with `matlabcp` command 10-12
 - with `matlabtb` command 10-25
 - with `matlabtbeval` command 10-38
- socket port numbers 8-15
 - as a networking requirement 8-15
 - checking 2-34
 - specifying for HDL Cosimulation blocks 4-34
 - specifying for TCP/IP link 4-12
- socket property
 - specifying with `hdldaemon` function 2-22
- sockets 1-7
 - See also* TCP/IP socket communication
- socketsimulink property
 - description of 10-42
 - specifying TCP/IP socket for HDL simulator with 4-12
- software
 - installing EDA Simulator Link™ 1-12
 - installing related application software 1-12
 - optional 1-9
 - required 1-9
- source device
 - adding to a Simulink model 4-8
 - specifying block ports for 4-15
 - specifying clocks for 4-30
 - specifying communication for 4-34
 - specifying Tcl commands for 4-36
- standard logic data 9-9
- standard logic vectors
 - converting for MATLAB 9-9
 - converting for the HDL simulator 9-10
- start time 9-15
- startupfile property
 - description of 10-42
- status option
 - checking value of 2-34
- status, link 2-34
- STD_LOGIC data type 2-8
 - conversion of 9-5
 - converting to 9-10
- STD_LOGIC_VECTOR data type 2-8
 - conversion of 9-5
 - converting for MATLAB 9-9
 - converting to 9-10
- STD_ULONGIC data type 2-8
 - conversion of 9-5
 - converting to 9-10
- STD_ULONGIC_VECTOR data type 2-8
 - conversion of 9-5
 - converting for MATLAB 9-9
 - converting to 9-10
- stimuli, block internal 4-30
- stop time 9-15
- strings, time value 2-32
- subtypes, VHDL 2-8

T

Tcl commands

- added to startup script via `nclaunch` 10-42
- configuring for block simulation 4-36
- configuring the HDL simulator to start with 4-12
- `hdlsimmatlab` 10-9
- `hdlsimulink` 10-10
- post-simulation
 - using `set_param` 4-36
- pre-simulation
 - using `set_param` 4-36
- specified in Tcl pane of HDL Cosimulation block 11-2

Tcl pane

- description of 11-2

`tclHdlSim` function

- description of 10-50

`tclstart` property

- with `nclaunch` function 10-42

TCP/IP networking protocol 1-7

- as a networking requirement 8-15
- See also* TCP/IP socket communication

TCP/IP socket communication

- as a communication option for MATLAB 8-12
- as a communication option for Simulink 8-13
- for Simulink applications 4-12
- mode 1-7
- specifying with `hdldaemon` function 2-22

TCP/IP socket ports 8-15

- specifying for HDL Cosimulation blocks 4-34

test bench applications

- coding for EDA Simulator Link™ software
 - overview of 2-2
- programming with EDA Simulator Link™ software
 - overview of 2-2

test bench function

- associating with HDL module 2-28
- `matlabtb` 2-12

- `matlabtb` 2-12

- scheduling invocation of 2-30

test bench functions

- adding to MATLAB search path 2-21
- coding for HDL verification 2-12
- defining 7-1
- naming 2-28
- programming for HDL verification 2-12
- sample of 2-13
- scheduling invocation of 2-32
- specifying required parameters for 7-1

test bench sessions

- logging changes to signal values during 6-2
- monitoring 2-35
- restarting 2-38
- running 2-35
- stopping 2-39

The HDL simulator is running on this computer parameter

- specifying block communication with 4-34

time 9-15

- callback 7-1
- delta 9-25
- simulation 7-1
 - guidelines for 9-15
 - representation of 9-15
- See also* time values

TIME data type 2-8

- conversion of 9-5
- converting to 9-10

time property

- setting return time type with 2-22

time scale, VCD file 11-27

time values

- specifying as scheduling options 2-32
- specifying with `hdldaemon` function 2-22

Timescales pane

- description of 11-2

timing errors 9-15

Timing mode

- absolute 4-27
- configuring for cosimulation 4-27
- relative 4-27
- `tnext` parameter 7-1
 - controlling callback timing with 2-32
 - specifying as scheduling options 2-32
 - time representations for 2-32
- `tnow` parameter 7-1
- To VCD File block
 - description of 11-24
 - uses of 1-6
- tools, postprocessing 6-2
- `tscale` parameter 7-7
- tutorials 1-23
 - for EDA Simulator Link™ 1-21

U

- unsigned data 9-9
- UNSIGNED data type 9-10
- unsupported data types 2-8
- users
 - for EDA Simulator Link™ software 1-9

V

- value change dump (VCD) files 6-2
 - See also* VCD files
- VCD file name parameter
 - description of 11-24
- VCD files
 - format of 11-27
 - using 6-2
- `vcd2wlf` command 6-2
- vectors
 - converting for MATLAB 9-9
 - converting to 9-10
- verification
 - coding test bench functions for 2-12
- verification sessions

- logging changes to signal values during 6-2
- monitoring 2-35
- restarting 2-38
- running 2-35
- stopping 2-39
- Verilog data types
 - conversion of 9-5
- Verilog modules
 - coding for MATLAB verification 2-7
 - coding for MATLAB visualization 3-7
- VHDL data types
 - conversion of 9-5
- VHDL entities
 - coding for MATLAB verification 2-7
 - coding for MATLAB visualization 3-7
 - sample definition of 2-10
 - verifying port direction modes for 7-7
- VHDL models 1-2
 - compiling 2-10
 - debugging 2-10
 - See also* HDL models
- visualization
 - coding component functions for 3-8
 - coding functions for 2-12

W

- Wave Log Format (WLF) files 6-2
- wave window, HDL simulator 4-15
- waveform files 6-2
- `which` function
 - for component function 3-10
 - for test bench function 2-21
- WLF files 6-2
- workflow
 - Cadence Incisive or NC Simulator simulator
 - with Simulink 4-2
 - HDL simulator with MATLAB 2-4
 - HDL simulator with MATLAB component
 - function 3-2

Z

zero-order hold 9-15