EDA Simulator Link™ DS 2
User's Guide

MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

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™ DS User's Guide

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.

Contents

Linking MATLAB to Synopsys VCS MX Simulators

2

Linking Simulink Software to Synopsys VCS MX Simulators

3

4

MATLAB Function Reference

5

HDL Simulator Command Extensions Reference

6

Simulink Block Reference

A

Examples

E

EDA Simulator Link DS Best Practices

Index

1

Getting Started

Product Overview

Integration with Other Products

The EDA Simulator Link™ DS 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 Synopsys hardware description language (HDL) simulator, Synopsys VCS™ MX, 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:

- Synopsys VCS MX — Hardware modeling in HDL and simulation

- MATLAB — Numerical computing, algorithm development, and visualization

- Simulink — Simulation of system-level designs and complex models

Note Synopsys VCS MX software may also be referred to as *the HDL simulator* throughout this document.

The EDA Simulator Link DS software consists of MATLAB functions and HDL simulator commands that establish communication links between the HDL simulator and The MathWorks products. In addition, the link software

contains a library of Simulink blocks that you may use to include HDL simulator designs in Simulink models for cosimulation.

EDA Simulator Link DS 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.

1

Develop specification

MATLAB
Signal Processing Toolbox
Filter Design Toolbox
Communications Toolbox
Simulink
Simulink Fixed Point
Signal Processing Blockset
Communications Blockset

3

Verify design

EDA Link DS
Discovery Software

2

Implement design
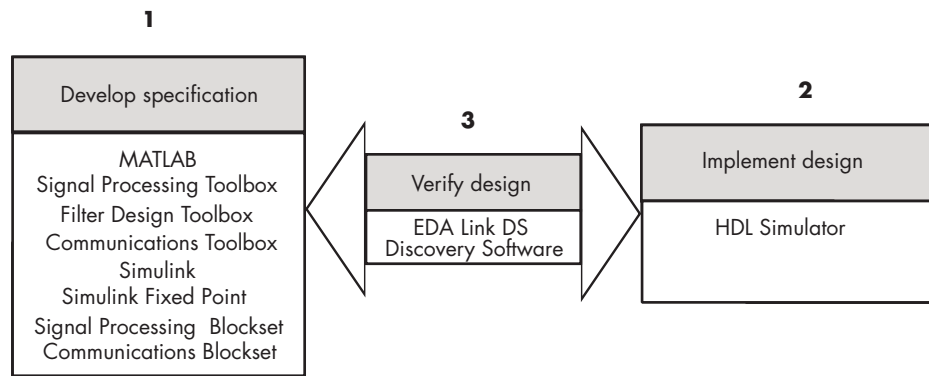
HDL Simulator

As the figure shows, EDA Simulator Link DS 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 DS 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 DS 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 DS MATLAB functions and the HDL Cosimulation block offer the same language-transparent feature set for both Verilog and VHDL models.

EDA Simulator Link DS 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.
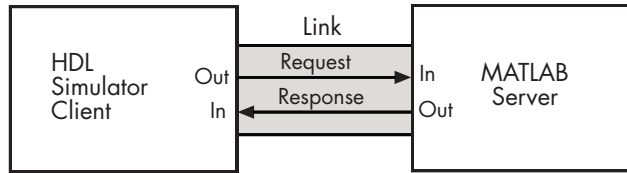
In this scenario, a MATLAB server function waits for service requests that it receives from an HDL simulator session. After receiving a request, the server establishes a communication link and invokes a specified MATLAB function that computes data for, verifies, or visualizes the HDL module (coded in 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 DS function `launchDiscovery`. Required and optional parameters allow you to specify the following:

- Tcl commands that execute as part of startup
- A specific Synopsys VCS MX executable to start
- The name of a VCS 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.
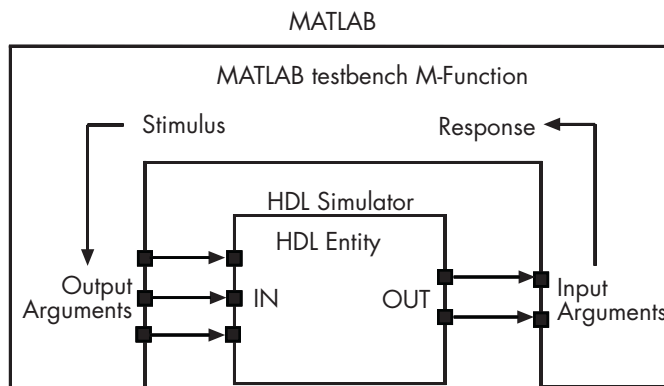
The following figure shows how a MATLAB component function is wrapped around by and communicates with the HDL simulator during a component simulation session.
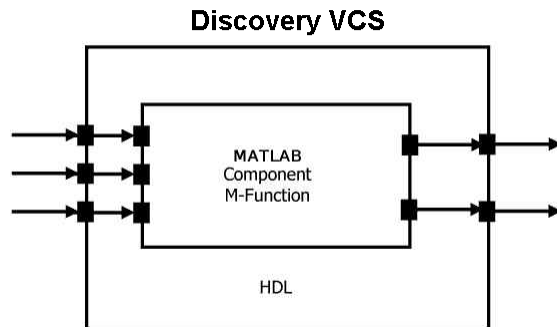
During the configuration process, EDA Simulator Link DS software equips the HDL simulator with a set of customized command extensions you use to perform the following tasks:

- Load the HDL simulator with an instance of an HDL module to be tested with MATLAB

- Begin a MATLAB test bench or component session for that instance

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

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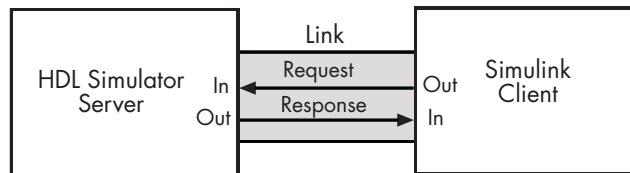
In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For example, you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

EDA Simulator Link DS 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 "TCP/IP Socket Communication".

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 DS software assumes users have a moderate level of prerequisite knowledge in the following subject areas:

- Hardware design and system integration
- VHDL and/or Verilog
- Synopsys VCS MX 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 DS software requires the following:

Platform	Visit the EDA Simulator Link DS requirements page on The MathWorks Web site for specific platforms supported with the current release.
Application software	Requires Synopsys® Discovery™ family of simulators
	Visit the EDA Simulator Link DS requirements page on The MathWorks Web site for specific versions supported with the current release.
	MATLAB
Application software required for cosimulation	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
	Note Many EDA Simulator Link DS demos require one or more of the optional products listed.
Platform-specific software	The EDA Simulator Link DS shared libraries (`liblfdhdls*.so`, `liblfdhdlc*.so`) are built using the GCCs included in the Synopsys 2008.03 VG_GNU_PACKAGE distribution. To ensure compatibility with our product you must use one of these GCCs to compile your HDL.

Setting Up Your Environment for the EDA Simulator Link DS Software

In this section...
"Installing the Link Software" on page 1-11
"Installing Related Application Software" on page 1-11
"Using the EDA Simulator Link DS Libraries" on page 1-11

Installing the Link Software

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

Installing Related Application Software

Based on your configuration decisions and the software required for your EDA Simulator Link DS 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.

Note In addition to making sure Synopsys VCS MX is installed and on the path, you must also download and install one of the three supported GCCs in the Synopsys 2008.03 VG_GNU_PACKAGE release: gcc336 or gcc412. See "Using the EDA Simulator Link DS Libraries" on page 1-11.

Using the EDA Simulator Link DS 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 DS 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 matching your installed version of GCC from the VG_GNU_PACKAGE. The MATLAB`launchDiscovery` command understands a default version of this library.

Library Names

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

```
productdir/arch/lib{product_short_name}{client_server_tag}_{design_language}_{compiler_tag}.
     {libext}
```

where

`productdir`	discovery
`arch`	linux32, linux64, windows32
`product_short_name`	lfd
`client_server_tag`	c or s (MATLAB or Simulink)
`design_language`	mixed = pure vhdl or mixed vhdl/vlog
	vlog = pure vlog
`compiler_tag`	gcc336, gcc412, vs05
`libext`	dll or so

Not all combinations are supported. See "Default Libraries" on page 1-13 for valid combinations.

Default Libraries

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

With the EDA Simulator Link DS software, the default library for each platform is the version compiled using the same compiler that The MathWorks uses to compile MATLAB and Simulink. 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	**liblfdhdlc_vlog_gcc336.so** (for Verilog designs) **liblfdhdlcv_mixed_gcc336.so** (for VHDL or mixed language designs)	**liblfdhdls_vlog_gcc336.so** (for Verilog designs) **liblfdhdls_mixed_gcc336.so** (for VHDL or mixed-language designs)
	liblfdhdlc_vlog_gcc412.so liblfdhdlc_mixed_gcc412.so	liblfdhdls_vlog_gcc412.so liblfdhdls_mixed_gcc412.so
Windows32	**liblfdhdlc_vlog_vs05.dll** (for Verilog designs) **liblfdhdlc_mixed_vs05.dll** (For VHDL or mixed language designs)	**liblfdhdls_vlog_vs05.dll** (for Verilog designs) **liblfdhdls_vs05.dll** (for VHDL or mixed language designs)
	liblfdhdlc_vlog_gcc32.dll liblfdhdlc__mixed_gcc33.dll	liblfdhdls_vlog_gcc32.dll liblfdhdls_mixed_gcc33.dll

Using an Alternative Library

You can use a different HDL-side library by specifying it explicitly using the VG_GNU_PACKAGE properties of the launchDiscovery 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.

Example: EDA Simulator Link DS Alternate Library Using launchDiscovery. This example uses the GCC 4.1.2 VG_GNU_PACKAGE.

```
>> launchDiscovery( ...
        'LinkType',              'MATLAB', ...
        'VerilogFiles',          'mydesign.v', ...
        'TopLevel',              'mydesign', ...
        'AccFile',               'mydesign.acc', ...
        'UseDefaultVgGnuPackage', false, ...
        'VgGnuGccVersion',       'gcc-4.1.2' ...
        );
```

Starting the HDL Simulator

Starting the HDL Simulator from MATLAB

Start the Synopsy VCS MX simulator directly from MATLAB or Simulink by calling the MATLAB function `launchDiscovery('PropertyName','PropertyValue'...)`. This function starts and configures the HDL simulator for use with the EDA Simulator Link DS software. By default, the function starts the first version of the simulator executable (`simv.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.

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

```
>> launchDiscovery('property', 'value', ...)
```

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 `launchDiscovery` with property name/property value pairs. Refer to `launchDiscovery` reference documentation for specific information regarding the property name/property value pairs.

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

See "Example of Starting the Synopsys VCS MX Simulator from MATLAB" on page 1-16 for examples of using `launchDiscovery` with various property/name value pairs and other parameters.

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

Example of Starting the Synopsys VCS MX Simulator from MATLAB

This example compiles and launches a single-file HDL design for cosimulation with Simulink. The code allows the use of Verilog-2000 syntax in the HDL source. This code launches the Synopsys DVE software.

```
>> launchDiscovery( ...
        'LinkType',     'Simulink', ...
        'VerilogFiles', 'myinverter.v', ...
        'VlogAnFlags',  '+v2k', ...
        'TopLevel',     'myinverter', ...
        'AccFile',      'myinverter.acc' ...
    );
```

Starting the Synopsys VCS MX Software from a Shell

You can run the scripts generated with a call to launchDiscovery (or scripts you've created yourself) to start the Synopsys VCS MX software and load the link libraries outside of MATLAB. See launchDiscovery for more information and an example.

Workflow for Using the EDA Simulator Link DS Software with MATLAB Software

The following diagram illustrates the steps necessary to create and run a MATLAB test bench or component session.
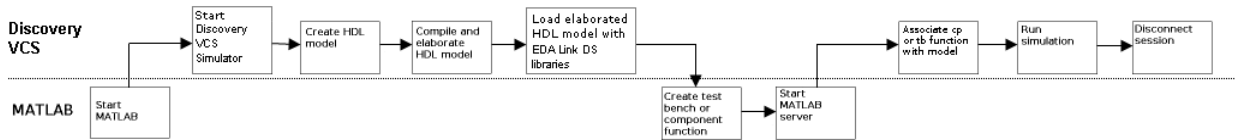
Note This workflow is a recommendation only. You might create, compile, and elaborate your HDL module differently than illustrated before starting MATLAB software. The preceding illustration is simply one possible workflow.

Workflow for Using the EDA Simulator Link DS Software with Simulink Software
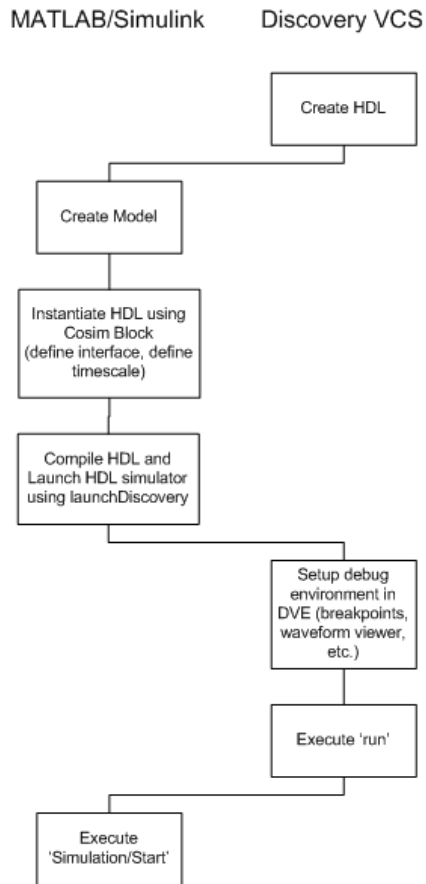
MATLAB/Simulink   Discovery VCS

```
                                    ┌──────────────┐
                                    │  Create HDL  │
                                    └──────────────┘
       ┌──────────────┐
       │ Create Model │
       └──────────────┘

  ┌───────────────────────┐
  │ Instantiate HDL using │
  │      Cosim Block      │
  │ (define interface,    │
  │   define timescale)   │
  └───────────────────────┘

  ┌───────────────────────┐
  │   Compile HDL and     │
  │ Launch HDL simulator  │
  │ using launchDiscovery │
  └───────────────────────┘
                                    ┌──────────────┐
                                    │ Setup debug  │
                                    │ environment  │
                                    │ in DVE       │
                                    │ (breakpoints,│
                                    │ waveform     │
                                    │ viewer, etc.)│
                                    └──────────────┘

                                    ┌──────────────┐
                                    │ Execute 'run'│
                                    └──────────────┘
  ┌──────────────────┐
  │     Execute      │
  │'Simulation/Start'│
  └──────────────────┘
```

Note This workflow is a recommendation only. You might create, compile, and elaborate your HDL module differently than illustrated before starting MATLAB and Simulink software. The preceding illustration is simply one possible workflow.

Learning More About the EDA Simulator Link DS Software

In this section...
"Documentation Overview" on page 1-19
"Online Help" on page 1-20
"Demos and Tutorials" on page 1-21

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, "Linking MATLAB to Synopsys VCS MX Simulators"	Explains how to code HDL models and MATLAB functions for EDA Simulator Link DS 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 and component sessions.
Chapter 3, "Linking Simulink Software to Synopsys VCS MX Simulators"	Explains how to use the HDL simulator and Simulink for cosimulation modeling.
Chapter 4, "MATLAB Function Reference"	Describes EDA Simulator Link DS functions for use with MATLAB.
Chapter 5, "HDL Simulator Command Extensions Reference"	Describes EDA Simulator Link DS commands for use with the HDL simulator.
Chapter 6, "Simulink Block Reference"	Describes EDA Simulator Link DS blocks for use with Simulink.

Appendix B, "EDA Simulator Link DS Machine Configuration Requirements"	Explains the machine configurations permissible when linking the HDL simulator to MATLAB or Simulink
Appendix C, "TCP/IP Socket Communication"	Provides information for choosing TCP/IP socket ports.
Appendix D, "Race Conditions in HDL Simulators"	Describes ways to avoid race conditions in hardware cosimulations with MATLAB and Simulink
Appendix E, "EDA Simulator Link DS Best Practices"	Provides suggestions for optimizing your cosimulation performance

Online Help

The following online help is available:

- Online help in the MATLAB Help browser. Click the EDA Simulator Link DS 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 DS MATLAB functions and HDL simulator commands. 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 f_x icon:
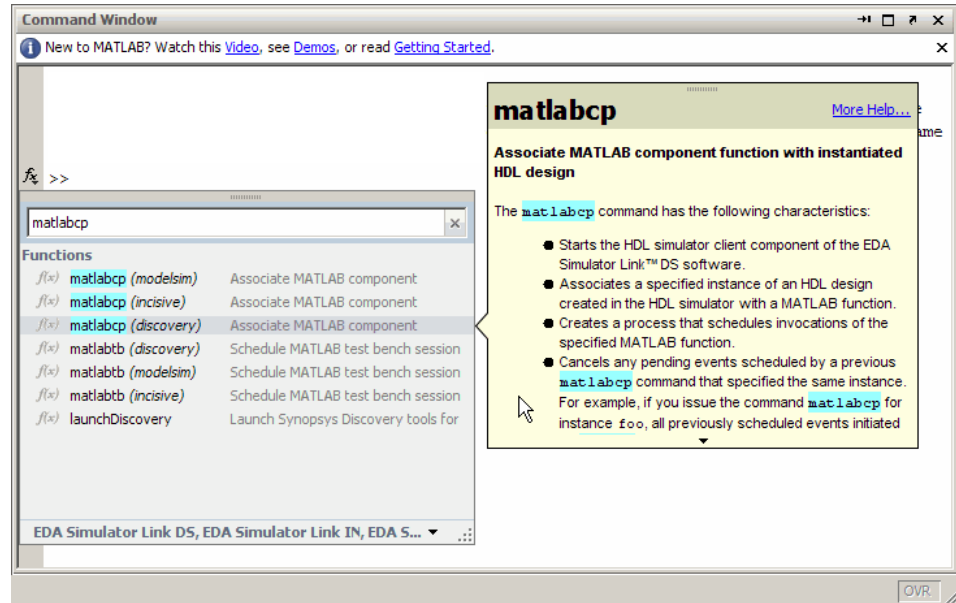
- Block reference pages accessible through the Simulink interface.

Demos and Tutorials

The EDA Simulator Link DS 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 DS demos with the online documentation. To access demos, type at the MATLAB command prompt:

```
>> demos
```

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

2

Linking MATLAB to Synopsys VCS MX Simulators

MATLAB-Synopsys VCS MX Workflow

The following table lists the steps necessary to create and run a MATLAB test bench or component session.

In MATLAB...	In the HDL simulator...
	1 Create the HDL model.
	2 Compile and elaborate the HDL model.

In MATLAB...	In the HDL simulator...
3 Start the MATLAB application, invoke the Synopsys VCS MX simulator, and load elaborated HDL model with EDA Simulator Link DS libraries using `launchDiscovery`, using "MATLAB" for LinkType property. See "Loading an HDL Design for Verification" on page 2-11 and `launchDiscovery` reference.	
4 Create a test bench or component function (see "Coding an EDA Simulator Link DS MATLAB Application" on page 2-4).	
5 Start the server. See "Starting the MATLAB Server" on page 2-47.	
	6 Use `matlabcp`, `matlabtb`, or `matlabtbeval` to associate the function you wrote in step 5 with a module of the loaded model currently in the HDL simulator (see "Associating the HDL Module Component with the MATLAB Link Function" on page 2-36). For additional scheduling and communication options, see "Scheduling Options for a Link Session" on page 2-38. See also the reference pages for `matlabcp`, `matlabtb`, and `matlabtbeval`.
	7 Run the simulation.
	8 Disconnect the session by using `nomatlabtb`.

Coding an EDA Simulator Link DS MATLAB Application

Overview

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

EDA Simulator Link DS software supports two types of MATLAB functions that interface to HDL modules:

- *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 syntax of a MATLAB test bench function is

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

- *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 syntax of a MATLAB component function is

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

These two types of MATLAB functions are referred to collectively as MATLAB *link functions*, and a test bench or component session may be referred to as a MATLAB *link session*.

The programming, interfacing, and scheduling conventions for test bench functions and MATLAB component functions are almost identical. Most of this chapter focuses on test bench functions, but in general all operations can be performed on and with both link functions. The test bench section is followed by a discussion of MATLAB component functions and how to use them.

Refer to Appendix B, "EDA Simulator Link DS Machine Configuration Requirements" for valid machine configurations.

Process for Coding an EDA Simulator Link DS MATLAB Application

This section provides an overview of the steps required to develop an HDL model for use with MATLAB and the EDA Simulator Link DS software. To program the HDL component of an EDA Simulator Link DS application,

1 Code the HDL model for MATLAB verification (see "Coding HDL Modules for MATLAB Verification" on page 2-7).

2 Compile the HDL model (see "Compiling and Debugging the HDL Model" on page 2-11).

3 Code the required MATLAB test bench or MATLAB component functions (see "Coding MATLAB Link Functions" on page 2-12).

4 Place the MATLAB functions on the MATLAB search path

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

MATLAB

MATLAB testbench M-Function

Stimulus   Response

HDL Simulator

HDL Entity

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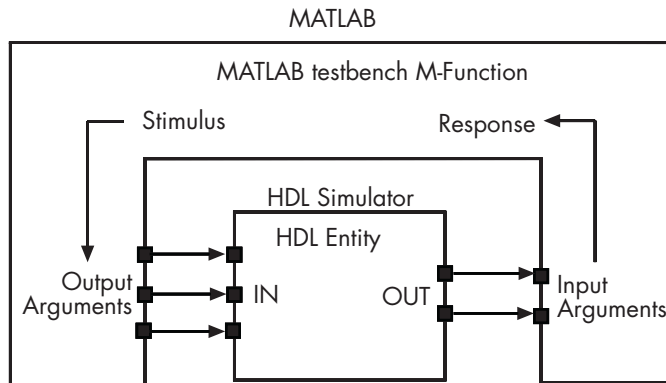
The following figure shows how an HDL simulator wraps around a MATLAB component function and how MATLAB communicates with the HDL simulator during a component simulation session.

Discovery VCS

MATLAB
Component
M-Function

HDL
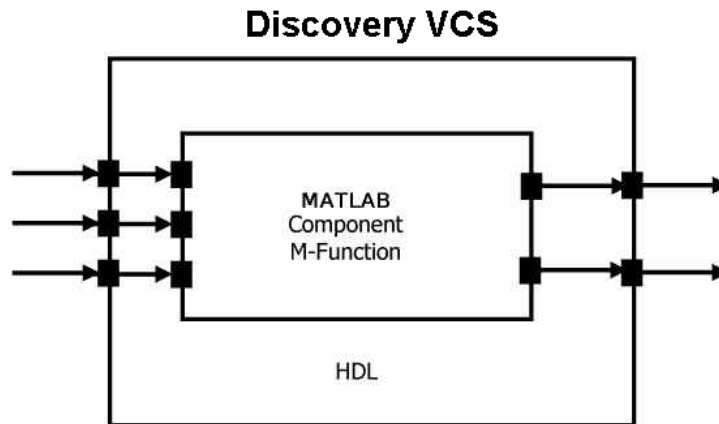
When linked with MATLAB, the HDL simulator functions as the client, with MATLAB as the server. The following figure shows a multiple-client scenario connecting to the server at TCP/IP socket port 4449.
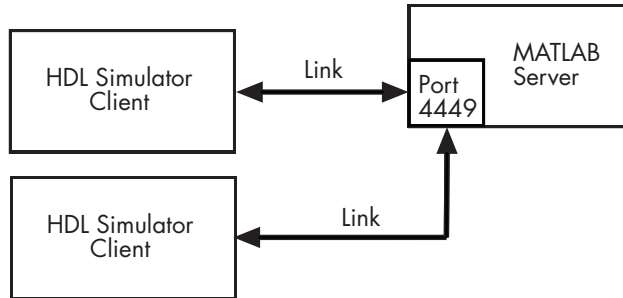
The MATLAB server can service multiple simultaneous HDL simulator sessions and HDL modules. However, you should follow recommended guidelines to ensure the server can track the I/O associated with each module and session. The MATLAB server, which you start with the supplied MATLAB function hdldaemon, 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 Appendix B, "EDA Simulator Link DS Machine Configuration Requirements" for valid machine configurations.

Coding HDL Modules for MATLAB Verification

- "Overview" on page 2-8
- "Choosing an HDL Module Name" on page 2-8
- "Specifying Port Direction Modes" on page 2-8
- "Specifying Port Data Types" on page 2-9
- "Sample VHDL Entity Definition" on page 2-10
- "Compiling and Debugging the HDL Model" on page 2-11
- "Loading an HDL Design for Verification" on page 2-11

Overview

The most basic element of communication in the EDA Simulator Link DS interface is the HDL module. The interface passes all data between the HDL simulator and MATLAB as port data. The EDA Simulator Link DS 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 function name. (Generally, naming rules for VHDL or Verilog and MATLAB are compatible.) By default, EDA Simulator Link DS software assumes that an HDL module and its simulation function share the same name. See "Naming a MATLAB Link Function" on page 2-35.

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 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

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 DS interface converts data types for the MATLAB environment, see "Performing Data Type Conversions" on page 2-18.

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

- `reg`

- `integer`

- `wire`

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

Sample VHDL Entity Definition

The next 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 DS interface converts data of standard logic scalar and array types for use in the MATLAB environment, see "Performing Data Type Conversions" on page 2-18.

```
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.

For more examples, see the EDA Simulator Link DS tutorials. For details on using the Synopsys VCS MX compiler, see the simulator documentation.

Loading an HDL Design for Verification

Start the HDL simulator from MATLAB and load an instance of an HDL module for verification with a call to `launchDiscovery('PropertyType', 'PropertyValue'...)`. At this point, it is assumed that you have coded and compiled your HDL model as explained in "Coding HDL Modules for MATLAB Verification" on page 2-7. For example:

```
launchDiscovery( ...
    'VerilogFiles','osc_top.v', ...
    'TopLevel', 'osc_top', ...
    'RunMode','GUI', ...
    'RunDir',projdir,...
    'LinkType','MATLAB',...
    'PreSimTcl', preSimTclCmds, ...
    'AccFile',tabaccessfile,...
    'VlogAnFlags', '"+v2k"' ...
);
```

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

Coding MATLAB Link Functions

- "Process for Coding MATLAB Link Functions" on page 2-12
- "Defining Link Functions and Link Function Parameters" on page 2-13
- "Performing Data Type Conversions" on page 2-18
- "Sample MATLAB Test Bench Function" on page 2-27

Process for Coding MATLAB Link Functions

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

1 Learn the syntax for a MATLAB link function (see "Defining Link Functions and Link Function Parameters" on page 2-13).

2 Understand how EDA Simulator Link DS software converts HDL modules data for use in the MATLAB environment (see "Performing Data Type Conversions" on page 2-18).

3 Choose a name for the MATLAB function (see "Choosing an HDL Module Name" on page 2-8).

4 Define expected parameters in the function definition line (see "Defining Link Functions and Link Function Parameters" on page 2-13).

5 Determine the types of port data being passed into the function (see "Defining Link Functions and Link Function Parameters" on page 2-13).

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 2-16).

7 Convert data for manipulation in the MATLAB environment, as necessary (see "Converting HDL Data to Send to MATLAB" on page 2-18).

8 Convert data that needs to be returned to the HDL simulator (see "Converting Data for Return to the HDL Simulator" on page 2-24).

Defining Link Functions and Link Function Parameters

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.

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

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

For more information on using tnext and tnow for simulation scheduling, see "Scheduling Options for a Link Session" on page 2-38.

The following table describes each of the link function parameters and the roles they play in each of the functions.

Parameter	Test Bench	Component
iport	*Output* Structure that forces (by deposit) values onto signals connected to input ports of the associated HDL module.	*Input* Structure that receives signal values from the input ports defined for the associated HDL module at the time specified by tnow.

Parameter	Test Bench	Component
tnext	*Output, optional* Specifies the time at which the HDL simulator schedules the next callback to MATLAB. tnext should be initialized to an empty value ([]). If tnext is not later updated, no new entries are added to the simulation schedule.	*Output, optional* Same as test bench.
oport	*Input* Structure that receives signal values from the output ports defined for the associated HDL module at the time specified by tnow.	*Output* Structure that forces (by deposit) values onto signals connected to output ports of the associated HDL module.
tnow	*Input* Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see "Scheduling Options for a Link Session" on page 2-38.	Same as test bench.
portinfo	*Input* For the first call to the function only (at the start of the simulation) , portinfo receives a structure whose fields describe the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the port's type, direction, and size.	Same as test bench.

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 2-16.

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

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

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

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

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

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

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

    portinfo.*field1*.*field2*.*field3*

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

HDL Port Information

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

HDL Port Information (Continued)

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

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

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

Performing Data Type Conversions

To successfully use the EDA Simulator Link DS software with the HDL simulator and MATLAB or Simulink, you need to understand the data type conversions that the EDA Simulator Link DS software performs to transmit and receive data between HDL modules and the MATLAB environment.

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 DS 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 DS software converts supported VHDL data types to MATLAB types based on whether the type is scalar or array.

VHDL-to-MATLAB Data Type Conversions

VHDL Types...	As Scalar Converts to...	As Array Converts to...
STD_LOGIC, STD_ULOGIC, and BIT	A character that matches the character literal for the desired logic state.	
STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED		A column vector of characters (as defined in VHDL Conversions for the HDL Simulator on page 2-24) with one bit per character.

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
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.

VHDL-to-MATLAB Data Type Conversions (Continued)

VHDL Types...	As Scalar Converts to...	As Array Converts to...
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 "Starting the MATLAB Server" on page 2-47).	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 characters as indicated for the types STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, and UNSIGNED.

The following table summarizes how the EDA Simulator Link DS 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 double	Use the double function to convert the data to type double before performing the computation. For example: `datas(inc+1) = double(idata);`

Required Data Conversions (Continued)

If You Need the Function to...	Then...
Convert a standard logic or bit vector to an unsigned integer or positive decimal	Use the mvl2dec function to convert the data to an unsigned decimal value. For example: `uval = mvl2dec(oport.val')` This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent. The mvl2dec function converts the binary data that the MATLAB function receives from the entity's osc_in port to unsigned decimal values that MATLAB can compute. See mvl2dec for more information on this function.
Convert a standard logic or bit vector to a negative decimal	Use the following application of the mvl2dec function to convert the data to a signed decimal value. For example: `suval = mvl2dec(oport.val'), true);` This example assumes the standard logic or bit vector is composed of the character literals '1' and '0' only. These are the only two values that can be converted to an integer equivalent.

Examples

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

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

2-23

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 DS 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 = dec2mvl(2)
```

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

```
oport.signal = dec2mvl(2, N)
```

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

VHDL Conversions for the HDL Simulator

To Return Data to an IN Port of Type...	Then...

`STD_LOGIC`, `STD_ULOGIC`, or `BIT`	Declare the data as a character that matches the character literal for the desired logic state. For `STD_LOGIC` and `STD_ULOGIC`, the character can be `'U'`, `'X'`, `'0'`, `'1'`, `'Z'`, `'W'`, `'L'`, `'H'`, or `'-'`. For `BIT`, the character can be `'0'` or `'1'`. For example:
	
	`iport.s1 = 'X'; %STD_LOGIC`
	`iport.bit = '1'; %BIT`

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...

STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	Declare the data as a column vector or row vector of characters (as defined above) with one bit per character. For example:
	
	```iport.s1v = 'X10ZZ'; %STD_LOGIC_VECTOR```
	```iport.bitv = '10100'; %BIT_VECTOR```
	```iport.uns = dec2mvl(10,8); %UNSIGNED, 8 bits```

Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See "Array Indexing Differences Between MATLAB and HDL" on page 2-21.
INTEGER or NATURAL	Declare the data as an array of type int32 with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type int32 with the MATLAB int32 function before returning it. Be sure to limit the data to values with the range of the VHDL type. If necessary, check the right and left fields of the portinfo structure. For example: ```iport.int = int32(1:10)';```
REAL	Declare the data as an array of type double with a size that is equivalent to the VHDL port size. For example: ```iport.dbl = ones(2,2);```

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...

TIME	Declare a VHDL TIME value as time in seconds, using type double, or as an integer of simulator time increments, using type int64. You can use the two formats interchangeably and what you specify does not depend on the hdldaemon 'time' option (see "Starting the MATLAB Server" on page 2-47), which applies to IN ports only. Declare an array of TIME values by using a MATLAB array of identical size and shape. All elements of a given port are restricted to time in seconds (type double) or simulator increments (type int64), but otherwise you can mix the formats. For example:
	
	 ```iport.t1 = int64(1:10)'; %Simulator time```
	 ``` %increments```
	 ```iport.t2 = 1e-9; %1 nsec```

Enumerated types	Declare the data as a 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 2-16). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as in the first example shown here. .
	
	 ```iport.char = {'''A''', '''B'''}; %Character```
	 ``` %literal```
	 ```iport.udef = 'mylabel'; %User-defined label```

Character array for standard logic or bit representation	Use the dec2mvl function to convert the integer. For example: ```oport.slva =dec2mvl([23 99],8)';``` This example converts two integers to a 2-element array of standard logic vectors consisting of 8 bits.

Verilog Conversions for the HDL Simulator

To Return Data to an input **Port of Type...**	Then...
`reg`, `wire`	Declare the data as a character or a column vector of characters that matches the character literal for the desired logic state (`'0'` or `'1'`). For example: `    iport.bit = '1';`
`integer`	Declare the data as a 32-element column vector of characters (as defined above) with one bit per character.

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 DS software. You can see the full text of the code used in this sample in the section M-Function Example: manchester_decoder.m on page 2-31.

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 "Defining Link Functions and Link Function Parameters" on page 2-13.

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` HDL simulator command. See "Naming a MATLAB Link Function" on page 2-35.

- 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 "Defining Link Functions and Link Function Parameters" on page 2-13.

  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.
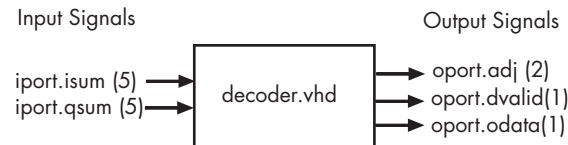
  For more information on the required MATLAB link function parameters, see "Defining Link Functions and Link Function Parameters" on page 2-13.

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

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

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

VHDL Example Port Definitions

Port	Direction	Type...	Converts to/Requires Conversion to...
isum	IN	STD_LOGIC_VECTOR(4 DOWNTO 0)	A 5-bit column or row vector of characters where each bit maps to standard logic character 0 or 1.
qsum	IN	STD_LOGIC_VECTOR(4 DOWNTO 0)	A 5-bit column or row vector of characters where each bit maps to standard logic character 0 or 1.
adj	OUT	STD_LOGIC_VECTOR(1 DOWNTO 0)	A 2-element column vector of characters. Each character matches a corresponding character literal that represents a logic state and maps to a single bit.

VHDL Example Port Definitions (Continued)

Port	Direction	Type...	Converts to/Requires Conversion to...
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 2-18.

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) = mvl2dec(oport.adj');
data(isum) = mvl2dec([oport.dvalid oport.odata]);
.
.
.
```

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

"Performing Data Type Conversions" on page 2-18 provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the HDL simulator.

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

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

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

"Converting Data for Return to the HDL Simulator" on page 2-24 provides a summary of the types of data conversions to consider when returning data to the HDL simulator.

M-Function Example: manchester_decoder.m

```
function [iport,tnext] = manchester_decoder(oport,tnow,portinfo)
% MANCHESTER_DECODER  Test bench for VHDL 'decoder'
%  [IPORT,TNEXT]=MANCHESTER_DECODER(OPORT,TNOW,PORTINFO) -
%    Implements a test of the VHDL decoder entity which is part
%    of the Manchester receiver demo.  This test bench plots
%    the IQ mapping produced by the decoder.
%
```

2-31

```
%       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 2003-2004 The MathWorks, Inc.
%   $Revision: 1.1.4.1 $  $Date: 2007/08/31 00:55:58 $

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
```

```matlab
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,
```

2-33

```
                   if data(k+1) < 2,  % Invalid
                       line(k,qsum,'color','r','Marker','X');
                   else
                       if data(k+1) == 2,  %Valid and 0!
                           line(k,qsum,'color','g','Marker','o');
                       else
                           line(k,qsum,'color','k','Marker','.');
                       end
                   end
             end

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

Associating a MATLAB Link Function with an HDL Module

Overview

This section describes establishing a relationship between the link function and the HDL model in the Synopsys VCS MX simulator by naming the link function (either implicitly or explicitly) and using scheduling options (action based on a specific time or event and registering callbacks) for the MATLAB link session.

Naming a MATLAB Link Function

You can name and specify a MATLAB link function however you like, so long as you follow MATLAB function and file naming guidelines. By default, the EDA Simulator Link DS software assumes the name for a MATLAB function matches the name of the HDL module that the function verifies or visualizes. For example, if you name the HDL module `mystdlogic`, the EDA Simulator Link DS software assumes the corresponding MATLAB function is `mystdlogic` and resides in the file `mystdlogic.m`.

Should you name the m-function or m-file something different than the HDL instance, you must specify the `-mfunc` parameter of one of the link functions and provide the m-function name.

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

Associating the HDL Module Component with the MATLAB Link Function

By default, the EDA Simulator Link DS software assumes the name for a MATLAB function matches the name of the HDL module that the function verifies or visualizes. See "Naming a MATLAB Link Function" on page 2-35.

In the Oscillator demo, the HDL model instantiates an HDL entity as the component u_osc_filter (see osc_top.v). After the HDL simulator compiles and loads the HDL model, an association must be formed between the u_osc_filter component and the MATLAB component function oscfilter. To do so, the software invokes the HDL simulator command matlabcp when the simulation is set up.

```
matlabcp u_osc_filter -mfunc oscfilter
```

The matlabcp command instructs the HDL simulator to call back the oscfilter function when u_osc_filter executes in the simulation.

Specifying HDL Signal/Port and Module Paths for MATLAB Link Sessions

The rules stated in this section are for signal/port and module path specifications for MATLAB link sessions. Other specifications may work but are not guaranteed to work in this or future releases.

In the following example:

```
matlabcp u_osc_filter -mfunc oscfilter
```

u_osc_filter is the top level component. However, if you are specifying a subcomponent, you must follow valid module path specifications for MATLAB link sessions.

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

Path Specifications for MATLAB Link Sessions with Verilog Top Level

These path specifications rules must be followed:

- The path specification must start with a top-level module name.
- The 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
:sub:port_or_sig
:
:sub
```

Path Specifications for MATLAB Link Sessions with VHDL Top Level

These path specifications rules must be followed:

- The path specification may include the top-level module name, but it is not required.
- The 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 are valid signal and module path specification examples:

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

The following examples show invalid signal and module path specifications:

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

Specifying TCP/IP Values

When providing TCP/IP information for a MATLAB link function, you can choose a TCP/IP port number or TCP/IP port alias or service name.

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 Appendix C, "TCP/IP Socket Communication" for more detail in specifying TCP/IP values.

For example,

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

A remote connection might look like this:

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

or

```
>matlabcp u_osc_filter -mfunc oscfilter -socket 4449@computer23
```

Scheduling Options for a Link Session

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

- Using the arguments to the `matlabcp` or `matlabtb` functions ("Scheduling Link Functions Using Link Function Parameters" on page 2-39)

- Inside the MATLAB m-function using the `tnext` parameter ("Scheduling Link Functions Using the tnext Parameter of an M-Function" on page 2-43)

You can schedule a MATLAB simulation function to execute under any of the following conditions:

- At a time that the MATLAB function passes to the HDL simulator with the `tnext` parameter

- Based on a time specification that can include discrete time values, repeat intervals, and a stop time

- When a specified signal experiences a rising edge—changes from `'0'` to `'1'`

- When a specified signal experiences a falling edge—changes from `'1'` to `'0'`

- Based on a sensitivity list—when a specified signal changes state

Decide on a combination of options that best meet your test bench application requirements. For details on using the `tnext` parameter and information on setting other scheduling parameters, see "Scheduling Link Functions Using the tnext Parameter of an M-Function" on page 2-43.

Scheduling Link Functions Using Link Function Parameters

By default, the EDA Simulator Link DS software invokes a MATLAB test bench function once (when time equals 0). If you want to apply more control and execute the MATLAB function more than once, decide on scheduling options that specify when and how often the EDA Simulator Link DS software is to invoke the relevant MATLAB function. Depending on your choices, you may need to modify the function or specify specific arguments when you begin a MATLAB test bench session with the `matlabtb` function.

In addition, the `matlabtb` function can include parameters that control when the MATLAB function executes.

You must specify at least one instance of a VHDL entity or Verilog module in your HDL model. By default, the command establishes a shared memory communication link and associates the specified instance with a MATLAB

function that has the same name as the instance. See "Associating the HDL Module Component with the MATLAB Link Function" on page 2-36.

The matlabtbeval function executes the MATLAB function immediately, while matlabtb provides several options for scheduling MATLAB function execution. The following table lists the various scheduling options.

Simulation Scheduling Options

To Specify MATLAB Function Execution...	Include...	Where...
At explicit times	time[, ...]	time represents one of n time values, past time 0, at which the MATLAB function executes.
		For example:
		```\n10 ns, 10 ms, 10 sec\n```
		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.

Simulation Scheduling Options (Continued)

To Specify MATLAB Function Execution...	Include...	Where...
At a combination of explicit times and repeatedly at an interval	time[, ...] -repeat n	time represents one of n time values at which the MATLAB function executes and the n specified with -repeat represents an interval between MATLAB function executions. The interface applies the union of the two options. For example: `5 ns -repeat 10 ns` The MATLAB function executes at time equals 0 ns, 5 ns, 15 ns, 25 ns, and so on.

Simulation Scheduling Options (Continued)

To Specify MATLAB Function Execution...	Include...	Where...
When a specific signal experiences a rising or falling edge	`-rising signal[, ...]` `-falling signal[, ...]`	`signal` represents a path name of a signal defined as a logic type—`STD_LOGIC`, `BIT`, `X01`, and so on.
On change of signal values (sensitivity list)	`-sensitivity signal[, ...]`	`signal` represents a path name of a signal defined as any type. If the value of one or more signals in the specified list changes, the interface invokes the MATLAB function.
		Note Use of this option for `INOUT` ports can result in double calls.
		If you specify the option with no signals, the interface is sensitive to value changes for all signals.
		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.

Consider the following `matlabtb` command:

```
dve> matlabtb test -rising /test/clk
-socket 4449
```

This command links an instance of the entity `test` to function `test.m`, which executes within the context of MATLAB based on specified timing parameters. In this case, the MATLAB function is called when the signal `/test/clk` experiences a rising edge.

Arguments in the command line specify the following conditions:

`test`	That an instance of the entity `test` be linked with the MATLAB function `test`.
`-rising /test/clk`	That the MATLAB function `test` be called when the signal `/test/clk` changes from `'0'` to `'1'`.
`-socket 4449`	That TCP/IP socket port 4449 be used to establish a communication link with MATLAB.

To verify that the `matlabtb` or `matlabtbeval` command established a connection, change your input focus to MATLAB and call the function `hdldaemon` with the `'status'` option as follows:

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

If a connection exists, the function returns the message

```
HDLDaemon socket server is running on port 4449 with 1 connection
```

Scheduling Link Functions Using the tnext Parameter of an M-Function

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, which gets added to the MATLAB function's simulation schedule. If the function returns a null value ([]) , no new entries are added to the schedule.

You can set the value of tnext to a value of type double or int64. The following table explains how the interface converts each type of data for use in the HDL simulator environment.

Time Representations for tnext Parameter

If You Specify a...	The Interface...
double value	Converts the value to seconds. For example, the following value converts to the simulation time nearest to 1 nanosecond as a multiple of the current HDL simulator time resolution. tnext = 1e-9
int64 value	Converts to an integer multiple of the current HDL simulator time resolution limit. For example, the following value converts to 100 ticks of the current time resolution. tnext=int64(100)

Note The tnext parameter represents time from the start of the simulation. Therefore, tnext should always be greater than tnow. If it is less, nothing is scheduled.

Example. In the Oscillator demo, the oscfilter function calculates a time interval at which callbacks should be executed. This interval is calculated on the first call to oscfilter and is stored in the variable fastestrate. The variable fastestrate represents the sample period of the fastest oversampling rate supported by the filter, derived from a base sampling period of 80 ns.

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

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

A new value for `tnext` is returned each time the function is called.

Running MATLAB Link Sessions

Overview

The EDA Simulator Link DS software offers flexibility in how you start and control an HDL model test bench or component session with MATLAB software. A MATLAB link session is the application of a `matlabtb`, `matlabtbeval`, or `matlabcp` function.

Process for Running MATLAB Link Sessions

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

1 Place MATLAB link function on the MATLAB search path.

2 Check the MATLAB server's link status.

3 Start the MATLAB server.

4 Launch the HDL simulator for use with MATLAB.

5 Load an HDL model in the HDL simulator for simulation and verification with MATLAB.

6 Decide on how you want to schedule invocations of the MATLAB test bench function.

7 Register callbacks for the MATLAB link session.

8 Apply test bench stimuli.

9 Run and monitor the test bench session.

10 Restart simulator during a test bench session.

11 Stop a test bench session.

Placing a MATLAB Test Bench or Component Function on the MATLAB Search Path

The MATLAB function associated with an HDL component must be on the MATLAB search path or reside in the current working directory (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
D:\work\discovery\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.

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 directory to a desired location with the `cd` command.

Starting the MATLAB Server

Start the MATLAB server as follows:

1 Start MATLAB.

2 In the MATLAB Command Window, call the `hdldaemon` function with property name/property value pairs that specify whether the EDA Simulator Link DS 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

Use the following syntax:

```
hdldaemon('PropertyName', PropertyValue...)
```

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')
```

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

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 `matlabtbeval` HDL simulator command. In addition, if you specify TCP/IP socket mode, the socket port that you specify with this function and the HDL simulator command must match. For more information on modes of communication, see "Choosing TCP/IP Socket Ports" on page C-2.
For more information on establishing the HDL simulator end of the communication link, see "Associating the HDL Module Component with the MATLAB Link Function" on page 2-36.

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 DS 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.

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:

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

If the server is not running, the message reads

```
HDLDaemon is NOT running
```

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

Starting the HDL Simulator for Use with MATLAB

Start the HDL simulator directly from MATLAB by calling the MATLAB function `launchDiscovery`. See "Starting the HDL Simulator" on page 1-15 for instructions on using `launchDiscovery`.

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.
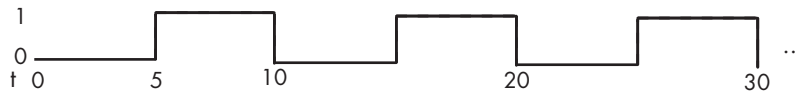
For example,

```
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.

Running a Link Session

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

1 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. 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
```

2 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.

3 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.

4 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.

5 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.

Restarting a Link Session

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 Call the `restart` command. `restart` also sources (runs) the pre-TCL commands specified in `launchdiscovery`. Therefore, if `matlabtb` was included in the pre-TCL commands, there is no need to call `matlabtb` again.

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

Stopping a Link Session

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.

3

Linking Simulink Software to Synopsys VCS MX Simulators

Simulink Software-Synopsys VCS MX Workflow

The following table lists the steps necessary to cosimulate an HDL design using Simulink software.

In MATLAB...	In the Synopsys VCS MX simulator...	In Simulink...
	1 Create the HDL model. **2** Compile and elaborate the HDL model.	
3 Start the MATLAB application, invoke the Synopsys VCS MX simulator , and load elaborated HDL model with EDA Simulator Link DS libraries using `launchDiscovery`, using "Simulink" for LinkType property. See "Loading an HDL Design for Verification" on page 2-11 and `launchDiscovery` reference.		

In MATLAB...	In the Synopsys VCS MX simulator...	In Simulink...
		4 Create a new Simulink model.
		5 Add an HDL Cosimulation block (see "Incorporating Hardware Designs into a Simulink Model" on page 3-29).
		6 Define the block interface (see "Defining the Block Interface" on page 3-34).
		7 Add other Simulink blocks to complete the Simulink model.

In MATLAB...	In the Synopsys VCS MX simulator...	In Simulink...
	8 (Optional) Set breakpoints for interactive HDL debug.	

		9 Run the simulation.
		
		10 Verify that the revised model runs as expected. If it does not, then:
		
		  a Modify the VHDL or Verilog code, and simulate it in the HDL simulator.
		
		  b Determine whether you need to reconfigure the HDL Cosimulation block. If you do, repeat steps 7 and 10.
		
		11 Consider using a To VCD File block to verify cosimulation results.

Introduction to Cosimulation

In this section...

"Creating a Hardware Model Design for Use in Simulink Applications" on page 3-5

"The EDA Simulator Link DS HDL Cosimulation Block" on page 3-7

"Communication Between the HDL Simulator and Simulink Software" on page 3-10

Creating a Hardware Model Design for Use in Simulink Applications

After you decide to include Simulink software as part of your EDA flow, think about how you will use it:

- Will you start by developing an HDL application using Synopsys VCS MX simulators, and possibly MATLAB software, and then test the results at a system level in Simulink?

- Will you start with a system-level model in Simulink with "black box hardware components" and, after the model runs as expected, replace the black boxes with HDL Cosimulation blocks?

- What other Simulink blocksets might apply to your application? Blocksets of particular interest for EDA applications include the Communications Blockset, Signal Processing Blockset, and Simulink Fixed Point software.

- Will you set up HDL Cosimulation blocks as a subsystem in your model?

- What sample times will be used in the model? Will any sample times need to be scaled?

- Will you generate a Value Change Dump (VCD) file?

After you answer these questions, use Simulink to build your simulation environment.

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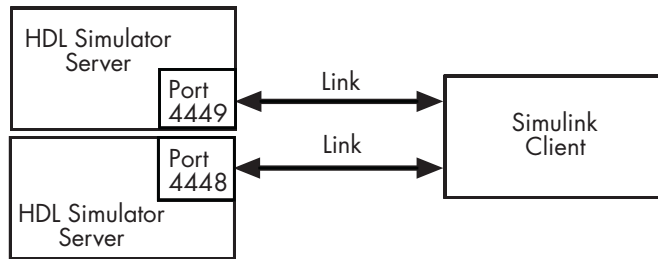
When you link the HDL simulator with a Simulink application, the simulator functions as the server. Using the EDA Simulator Link DS communications interface, an HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation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

This figure shows a sample Simulink model that includes an HDL Cosimulation block.
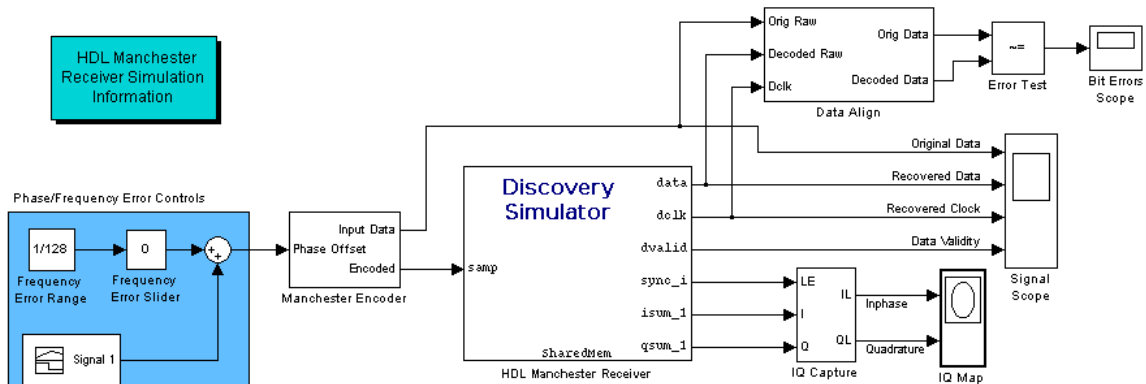
The HDL Cosimulation block models a Manchester receiver that is coded in HDL. Other blocks and subsystems in the model include the following:

- 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.

The EDA Simulator Link DS HDL Cosimulation Block

The EDA Simulator Link DS 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, 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 DS 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 6-3 in the Chapter 6, "Simulink Block Reference".
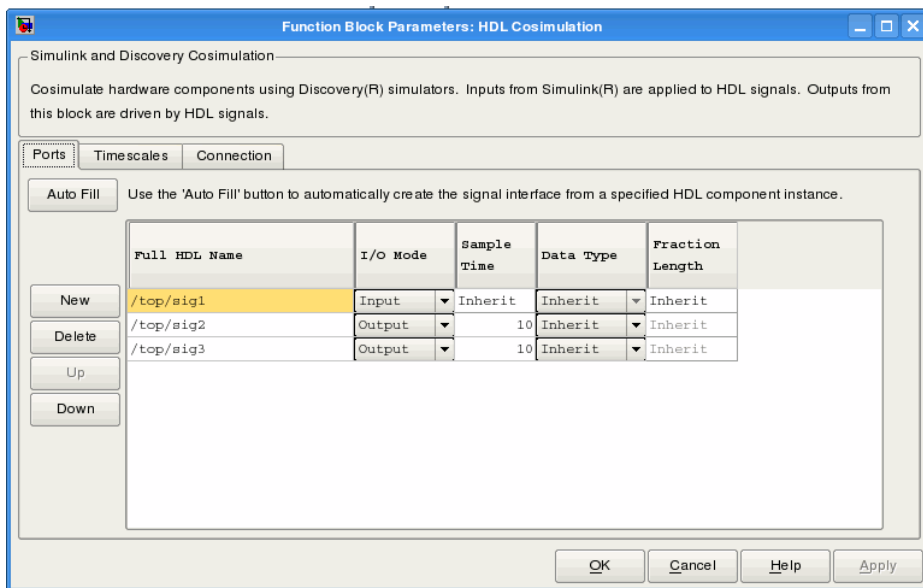
- **Connection Pane**: Type of communication and communication settings to be used for exchanging data between simulators. See "Connection Pane" on page 6-9 in the Chapter 6, "Simulink Block Reference".
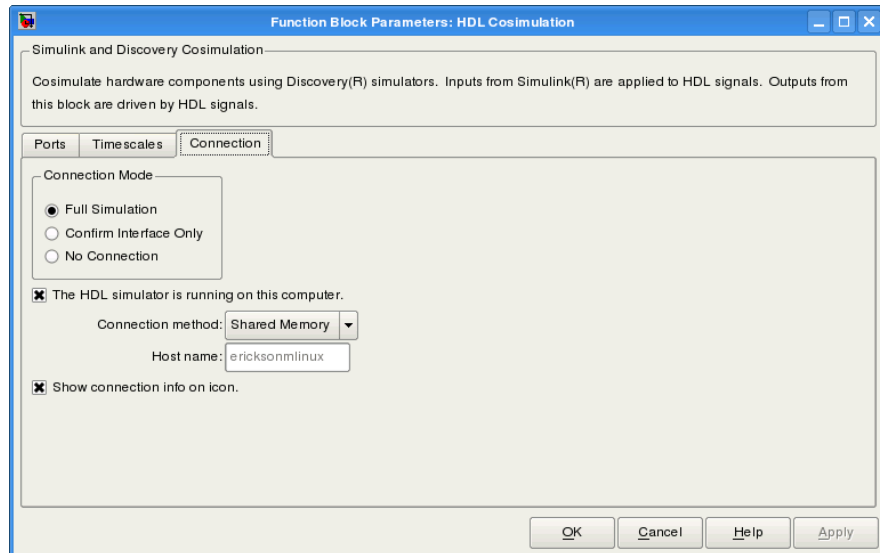
- **Timescales Pane**: The timing relationship between Simulink software and the HDL simulator. See "Timescales Pane" on page 6-13 in the Chapter 6, "Simulink Block Reference".
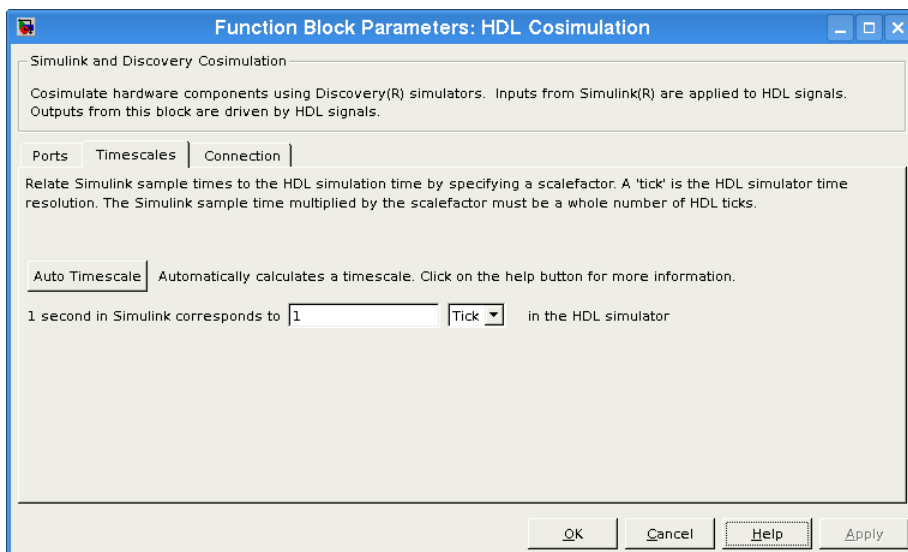
Communication Between the HDL Simulator and Simulink Software

When you link the HDL simulator with a Simulink application, the simulator functions as the server, as shown in the following figure.
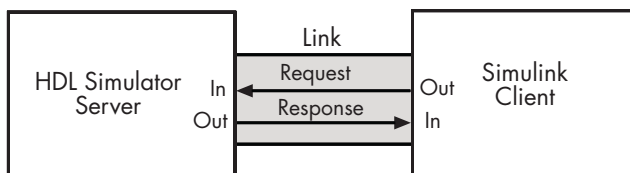
In this case, the HDL simulator responds to simulation requests it receives from cosimulation blocks in a Simulink model. You begin a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results. For

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.
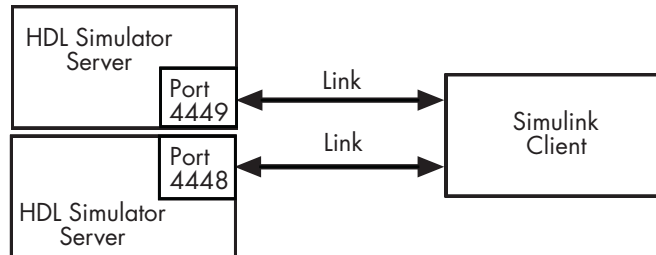
Preparing for Cosimulation

Overview

The EDA Simulator Link DS HDL Cosimulation block serves as a bridge between the Simulink and the HDL simulator domains. The block represents an HDL component model within the Simulink software. Using the block, Simulink software writes (drives) signals to and reads signals from the HDL model under simulation in the Synopsys VCS MX simulator. Signal exchange between the two domains occurs at regularly scheduled time steps defined by the Simulink sample time.

As you develop an EDA Simulator Link DS cosimulation application, you should be familiar with how signal values are handled across the simulation domains with respect to the following cases:

- How Simulink drives cosimulation signals

- Representation of simulation time

- Handling multirate signals

- Handling frame-based signals

- Avoiding race conditions

- Block simulation latency

- Interfacing with continuous time signals

- Setting Simulink configuration parameters

- Setting the communication link

- Starting the HDL simulator

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

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 or the model.

See the HDL simulator documentation for how to determine the current HDL simulator resolution limit

Simulink maintains simulation time as a double-precision value scaled to seconds. This representation accommodates modeling of both continuous and discrete systems.

The relationship between Simulink and the HDL simulator timing affects the following aspects of simulation:

- Total simulation time

- Input port sample times

- Output port sample times

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 DS 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 DS 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 DS

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 3-16.

- 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 3-18.

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.
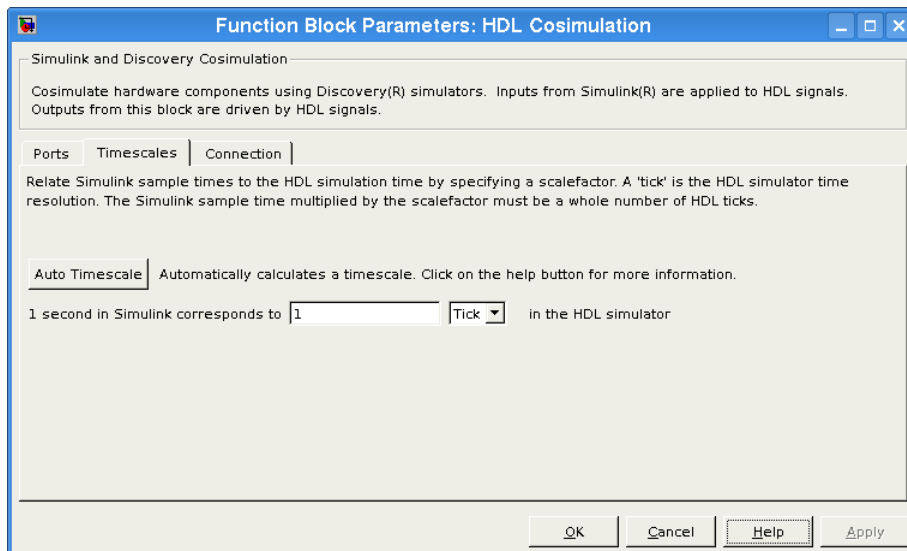
For instructions on setting the timing mode either automatically or manually, see "Timescales Pane" on page 6-13 in the HDL Cosimulation block reference.

Relative Timing Mode

Relative timing mode defines the following one-to-one correspondence between simulation time in Simulink and the HDL simulator:

One second in Simulink corresponds to *N ticks* in the HDL simulator, where N is a scale factor.

This correspondence holds regardless of the HDL simulator timing resolution.

The following pseudocode shows how Simulink time units are converted to HDL simulator ticks:

```
InTicks = N * tInSecs
```

where `InTicks` is the HDL simulator time in ticks, `tInSecs` is the Simulink time in seconds, and `N` is a scale factor.

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

- If the total simulation time in Simulink is specified as N seconds, then the HDL simulation will run for exactly N ticks (i.e., N ns at the default resolution limit).

- Similarly, if Simulink computes the sample time of an HDL Cosimulation block input port as *Tsi* seconds, new values will be deposited on the HDL input port at exact multiples of *Tsi* ticks. If an output port has an explicitly specified sample time of *Tso* seconds, values will be read from the HDL simulator at multiples of *Tso* ticks.

Relative Timing Mode Example
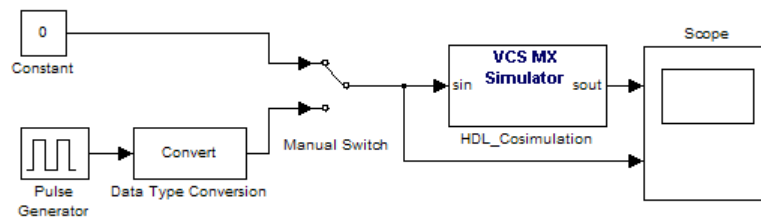
The model contains an HDL Cosimulation block (labeled `HDL_Cosimulation1`) simulating an 8-bit inverter that is enabled by an explicit clock. The inverter has a single input and a single output. The following code excerpt lists the Verilog code for the inverter:

```
module inverter_clock_vl(sin, sout,clk);

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


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

A cosimulation of this model might have the following settings:

- Simulation parameters in Simulink:
  - **Timescales** parameters: 1 Simulink second = 10 HDL simulator ticks
  - Total simulation time: `30 s`
  - Input port (`inverter_clock_vl.sin`) sample time: `N/A`
  - Output port (`inverter_clock_vl.sout`) sample time: `1 s`
- HDL simulator resolution limit: `1 ns`

Absolute Timing Mode

Absolute timing mode lets you define the timing relationship between Simulink and the HDL simulator in terms of absolute time units and a scale factor:

One second in Simulink corresponds to *(N * Tu) seconds* in the HDL simulator, where *Tu* is an absolute time unit (e.g., 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 3-17. Suppose that the model is reconfigured as follows:

- Simulation parameters in Simulink:
  - **Timescale** parameters: `1 s` of Simulink time corresponds to `1 s` of HDL simulator time.
  - Total simulation time: `60e-9 s` (60ns)
  - Input port (`/inverter/inport`) sample time: `24e-9 s` (24 ns)

- Output port (`/inverter/outport`) sample time: `12e-9 s (12 ns)`
  - Clock (`inverter/clk`) period: `10e-9 s (10 ns)`
- HDL simulator resolution limit: `1 ns`

Given these simulation parameters, the Simulink software will cosimulate with the HDL simulator for 60 ns, during which Simulink will sample inputs at a intervals of 24 ns, update outputs at intervals of 12 ns, and drive clocks at intervals of 10 ns.

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 3-20
- "Non-Integer Time Periods" on page 3-20

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 333.33333333333... ms, which is not a valid tick period for the HDL simulator. The HDL simulator will truncate such numbers. But Simulink does not make the same decision; thus, for cosimulation where you are trying to keep two independent simulators in synchronization, you should not assume anything. Instead you have to decide whether it is convenient to truncate or round the number.

Therefore, the solution is to "snap" either the Simulink sample time or the HDL sample time (via the timescale) to valid numbers. There are infinite possibilities, but here are some possible ways to perform a snap:

- Change Simulink sample times from 1/3 sec to 0.33333 sec 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. 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. 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 `senv timePrecision` 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 DS 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.

Handling Frame-Based Signals

This section discusses how to improve the performance of your cosimulation by using frame-based signals. An example is provided.

- "Overview" on page 3-23
- "Using Frame-Based Processing" on page 3-23

Overview

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.

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.

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 3-24. 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 scalars values, not vectors or arrays (with the exception of bit vectors, as VHDL and Verilog bit vectors are converted to the appropriately sized fixed-point scalar data type by the HDL Cosimulation block).

Avoiding Race Conditions in HDL Simulation

In the Synopsy VCS MX simulator, you cannot control the order in which clock signals (rising-edge or falling-edge) defined with `launchDiscovery` are applied, relative to the data inputs driven by these clocks. If you are careful to

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

For more on race conditions in hardware simulators, see Appendix D, "Race Conditions in HDL Simulators".

Block Simulation Latency

Simulink and the EDA Simulator Link DS 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 DS 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 ($t-\delta$) .
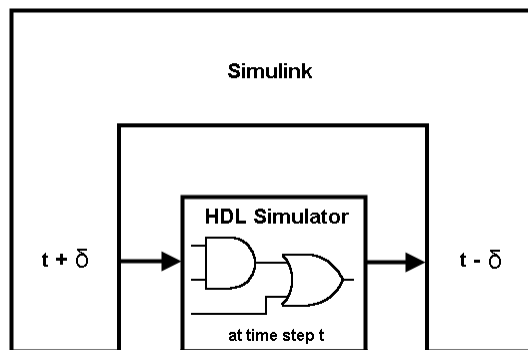
Regardless of whether you specify your HDL code with latency, the cosimulation block has a minimum latency that is equivalent to the cosimulation block's output sample time. For large sample times, the delay can appear to be quite long, but this length occurs as an artifact of the cosimulation block, 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.

Setting Simulink Software Configuration Parameters

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

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 DS Cosimulation blocks.

Simulink and HDL Simulator Communication Options

Select shared memory or socket communication. See "Communicating with MATLAB or Simulink and the HDL Simulator" on page 1-7.

Starting the HDL Simulator

See "Starting the HDL Simulator" on page 1-15.

Incorporating Hardware Designs into a Simulink Model

In this section...

Overview

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 DS 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 "Adding a Value Change Dump (VCD) File" on page 3-49. |
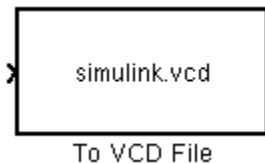
4 Copy the HDL Cosimulation block icon from the Library Browser to your model. Simulink creates a link to the block at the point where you drop the block icon.

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.

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 for Simulink Cosimulation Sessions with Verilog Top Level

Path specifications must follow these rules:

- 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
:sub:port_or_sig
:
:sub
```

Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level

Path specifications must follow these rules:

- 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
:sub:port_or_sig
:
:sub
```

Driving Clocks, Resets, and Enables

You can creating 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:

- Add signals to your Simulink model using Simulink blocks.

  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

Manchester Receiver Subsystem
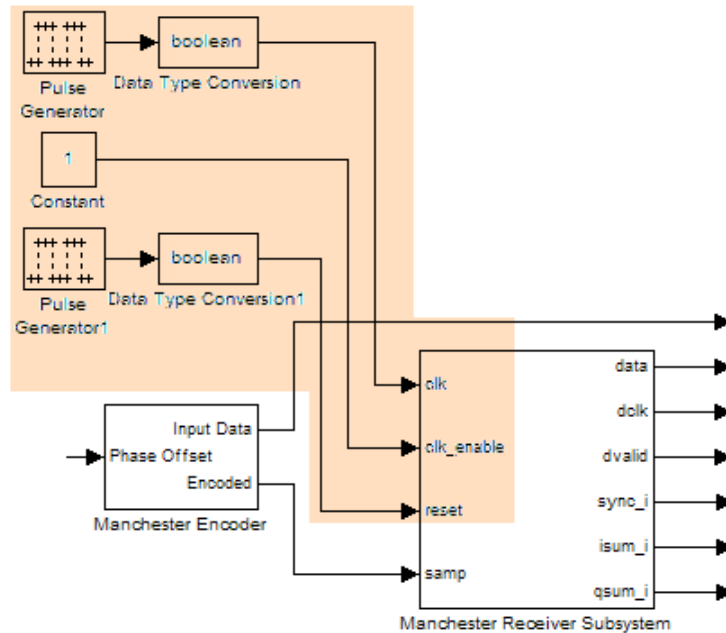
- Specify signals with the `launchDiscovery` command.

  For example (note code in bold):

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

- Implement clocks, resets, and enable signals 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.

Defining the 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.
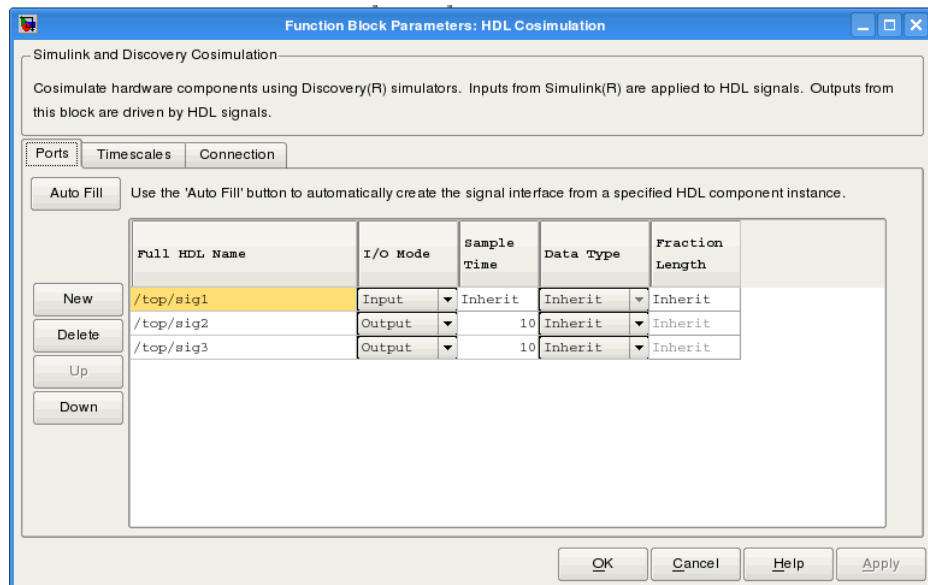
Mapping HDL Signals to Block Ports

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

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

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

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog box (see "Entering Signal Information Manually" on page 3-40). This approach can be more efficient when you want to connect a small number of signals from your HDL model to Simulink.

- Use the **Auto Fill** button to obtain signal information automatically by transmitting a query to the HDL simulator. This approach can save significant effort when you want to cosimulate an HDL model that has many signals that you want to connect to your Simulink model. However, in some cases, you will need to edit the signal data returned by the query. See "Obtaining Signal Information Automatically from the HDL Simulator" on page 3-35 for details.

Note Verify that signals used in cosimulation have read/write access. A tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

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. The following example describes the required steps.

The example is based on a modified copy of the Manchester Receiver model (see "Creating a Hardware Model Design for Use in Simulink Applications" on page 3-5), in which all signals were first deleted from the **Ports** pane.

1 Open the block parameters dialog box for the HDL Cosimulation block. Click the **Ports** tab. The **Ports** pane opens.
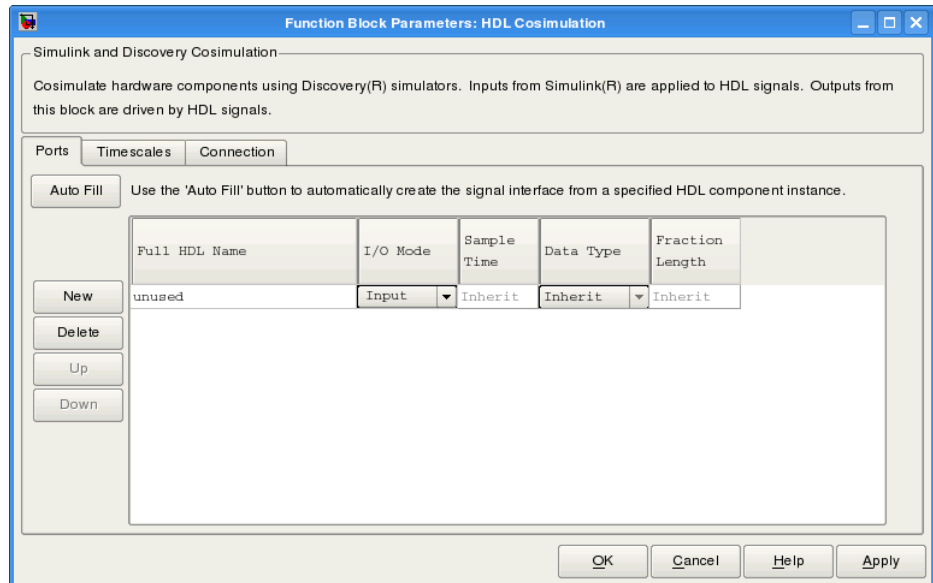
Tip Delete all ports before performing **Auto Fill** to ensure that no unused signal remains in the Ports list at any time.

2 Click the **Auto Fill** button. The **Auto Fill** dialog box opens.
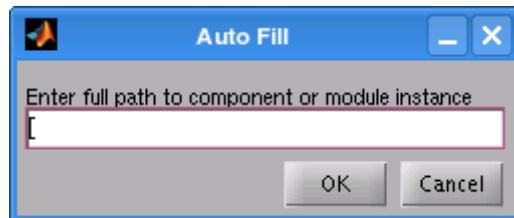
This modal dialog box requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit

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`.
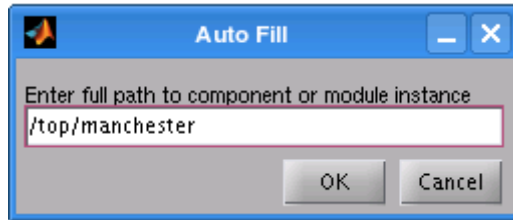
4 Click **OK** to dismiss the dialog box and the query is transmitted.

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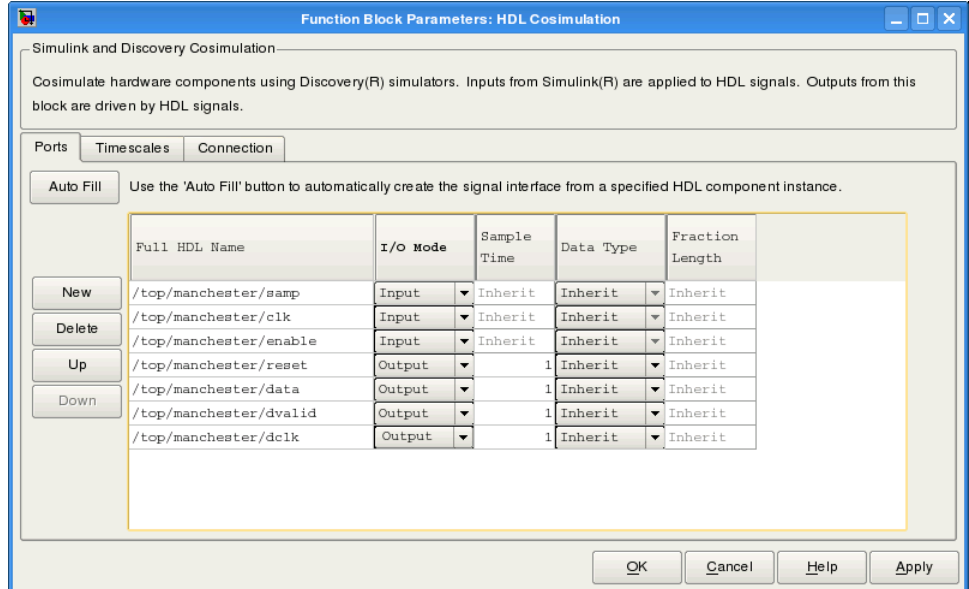
6 Click **Apply** to commit the port additions.

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 via PreSimTcl property of the
`launchDiscovery` function, as shown here:

```
PreSimTcl',    ...
                    { 'force manchester.clk 1 0, 0 5 -repeat 10', ...
                      'force manchester.enable 1 0', ...
                      'force manchester.reset 1 0, 0 1000' }, ...
```

The `enable` and `reset` signals are deleted from the **Ports** pane, as the
figures show.
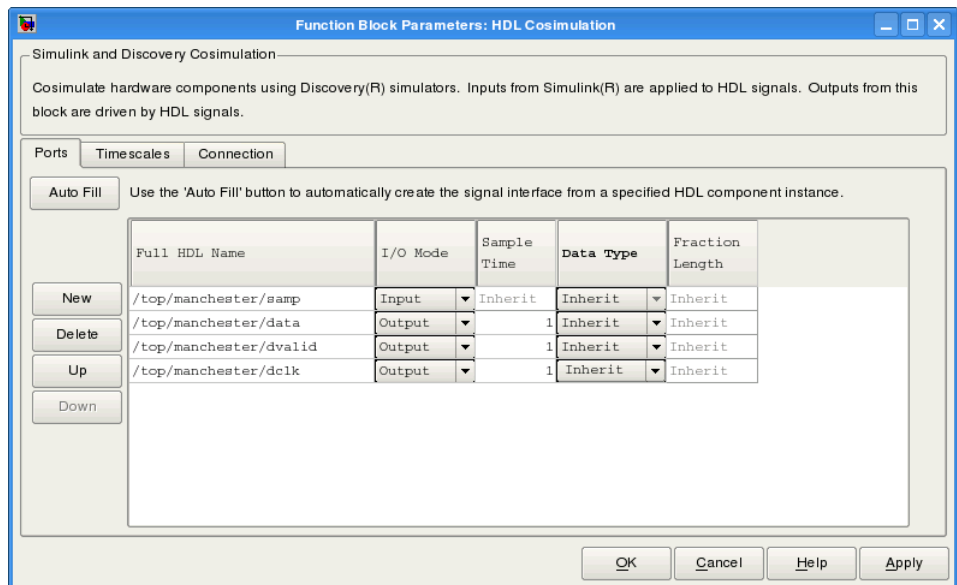
7 **Auto Fill** returns default values for output ports:

- **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 3-43.

8 Before closing the HDL Cosimulation block parameters dialog box, click **Apply** to commit any edits you have made.
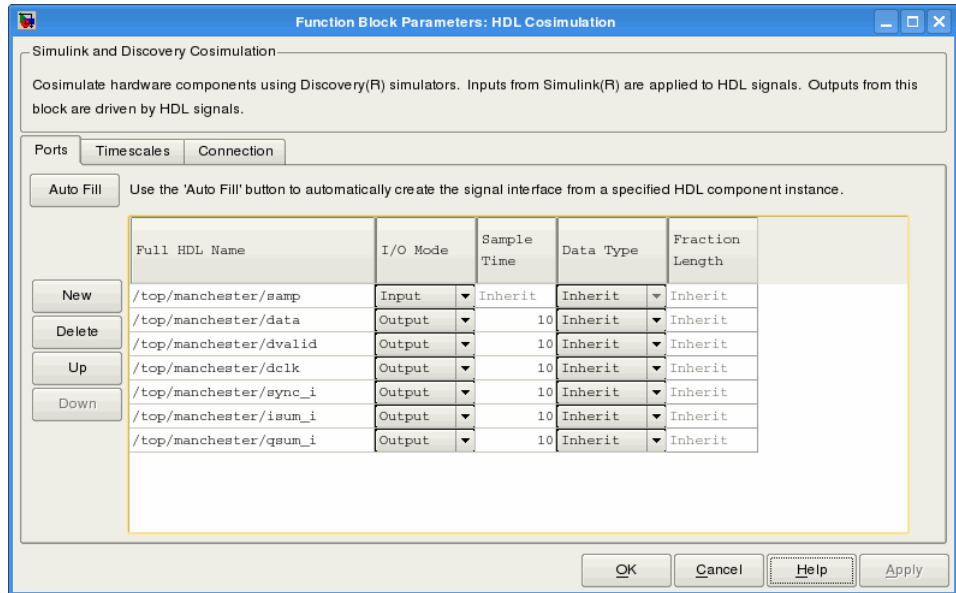
Observe that **Auto Fill** returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You may delete any such entries from the list in the **Ports** pane if 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 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.
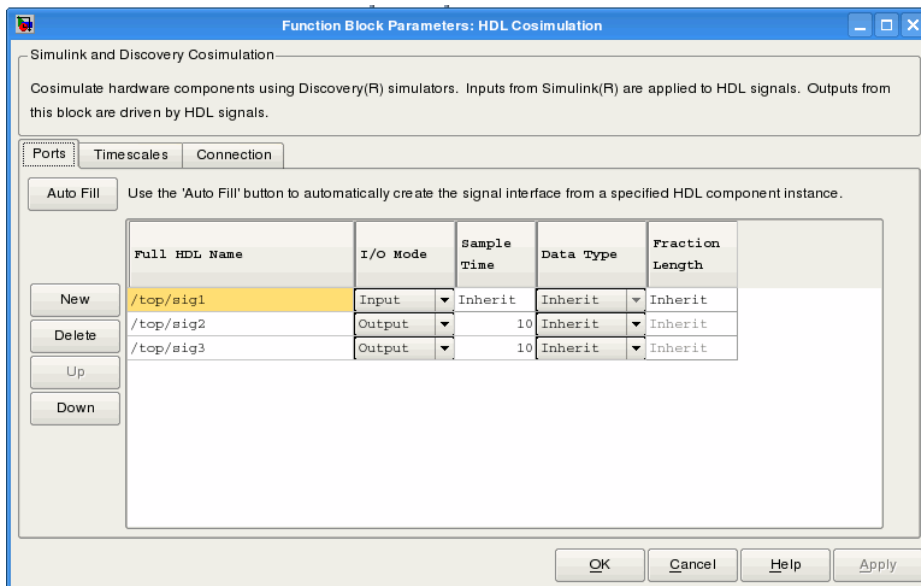
In this pane, you define the HDL signals of your design that you want to include in your Simulink block and set a sample time and data type for output ports. The parameters that you should specify on the **Ports** pane depend on the type of device the block is modeling as follows:

- 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 3-30).

- 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.

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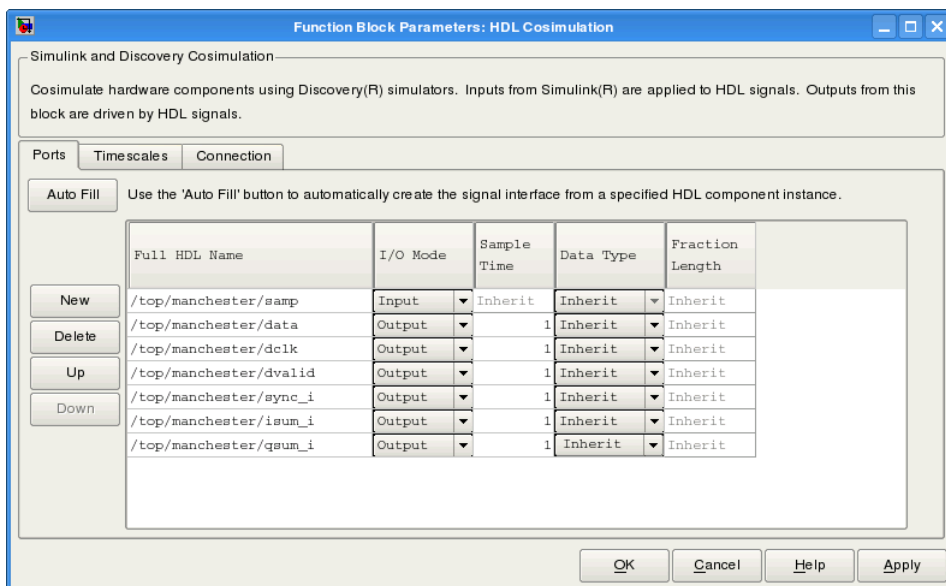
Note When you define an input port, make sure that only one source is set up to force input to that port. If multiple sources drive a signal, your Simulink model may produce unpredictable results.

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 DS cosimulation environment, see "Representation of Simulation Time" on page 3-13.

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, as follows:

- 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. The **Fraction Length** property is enabled when the signal **Data Type** property is not set to `Inherit`.

The **Data Type** and **Fraction Length** properties apply only to the following signals:

- VHDL signals of `STD_LOGIC` or `STD_LOGIC_VECTOR` type

- Verilog signals of `wire` or `reg` type

Output port data types are determined by the signal width and by the **Data Type** and **Fraction Length** properties of the signal. 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.

  Inherit is the default setting. When you select Inherit, the **Fraction Length** edit field is disabled.

  Simulink attempts to compute the data type of the signal connected to the output port by backward propagation. For example, if a Signal Specification block is connected 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 data type STD_LOGIC_VECTOR for a VHDL 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 is enabled. The port is assigned 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 is enabled. The HDL Cosimulation block assigns the port a fixed point type ufixN_EnF, where N is the signal width and F is the **Fraction Length** value.

  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.

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 "Representation

of Simulation Time" on page 3-13 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 6-13 of the HDL Cosimulation block reference.

Configuring the Communication Link in the HDL Cosimulation Block

Configure a block's communication link with the **Connection** pane of the block parameters dialog box.
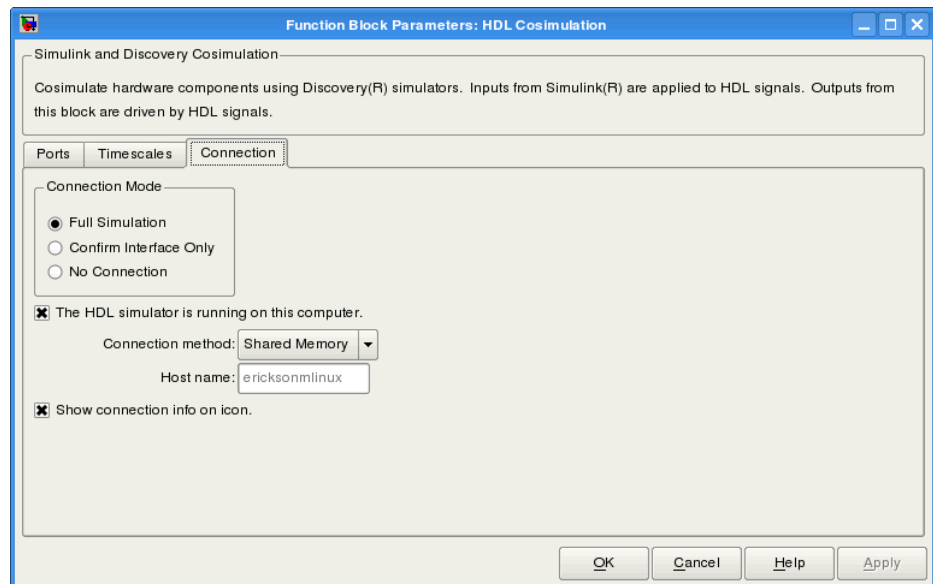
The following steps guide you through the communication configuration:

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 C-2. 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 C-2.

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 DS 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.
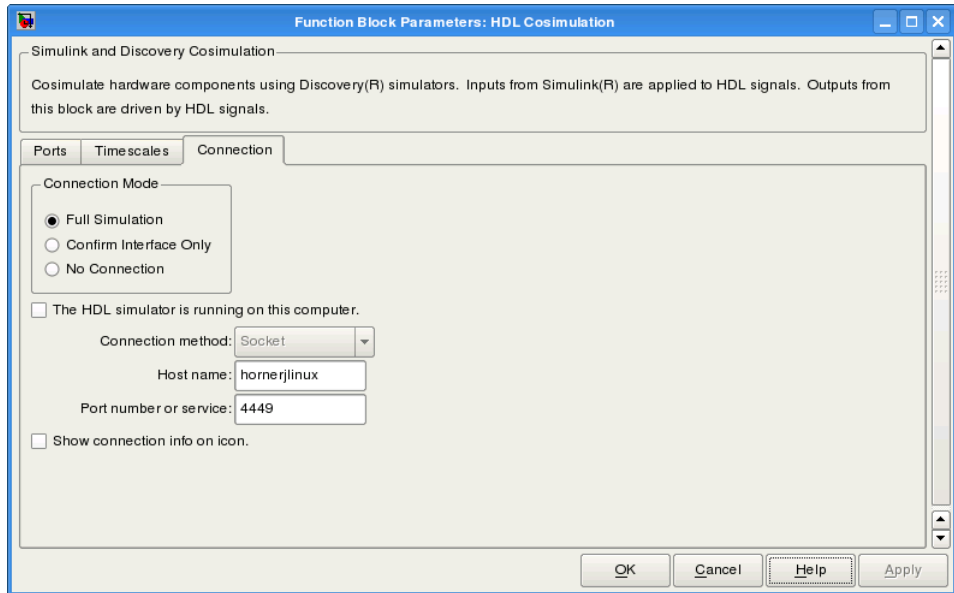
Programmatically Controlling the Block Parameters

One way to control block parameters is through the HDL Cosimulation block graphical dialog box. However, you can also control blocks by programmatically controlling the mask parameter values and the running of simulations. Parameter values can be read using the Simulink `get_param` function and written using the Simulink `set_param` function. All block parameters have attributes that indicate whether they are:

- Tunable — The attributes can change during the simulation run.

- Evaluated — The parameter string value undergoes an evaluation to determine its actual value used by the S-Function.

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

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

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

Some examples of using MATLAB to control simulations and mask parameter values follow. Usually, the commands are put into 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, e.g., **View > Model Explorer > Simulink Root > *model_name* > Model Workspace > Data Source is M-Code**.
- In a model callback, e.g., **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 DS 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**).

Examples
The following example shows the one way of programmatically controlling the HDL Cosimulation block parameters.

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

```
ttcp_exec = [matlabroot '/toolbox/shared/hdllink/scripts/ttcp_glnx'];
 [status, results] = system([ttcp_exec ' -a']);
 if ~s
```

```
   parsed_result = strread(results,'%s');
   avail_port = parsed_result{2};
  else
   error(results);
  end

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

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 EDA Simulator Link DS 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 DS block library.
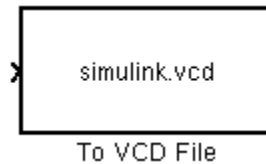
To VCD File

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.
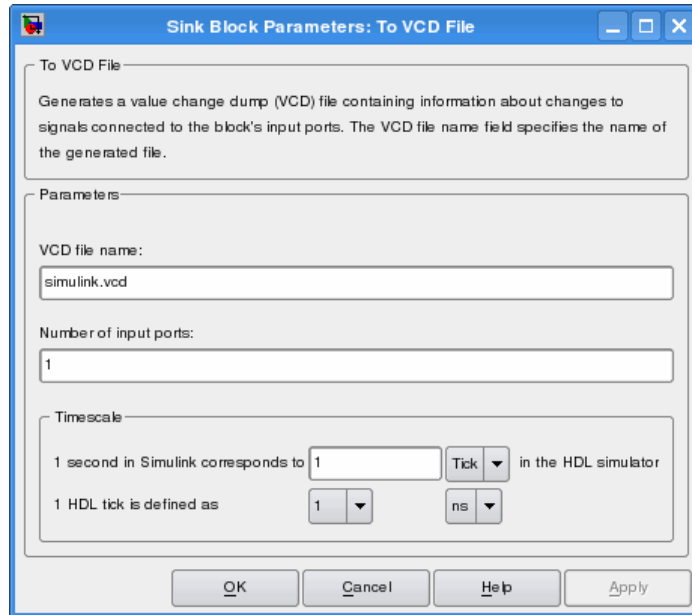
b Specify a file name for the generated VCD file in the **VCD file name** text box.

- If you specify a file name only, Simulink places the file in your current MATLAB directory.

- 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 6-21.

Running Cosimulation Sessions

In this section...
"Starting the HDL Simulator for Use with Simulink" on page 3-53
"Determining an Available Socket Port Number" on page 3-53
"Checking the Connection Status" on page 3-53
"Managing a Simulink Cosimulation Session" on page 3-54

Starting the HDL Simulator for Use with Simulink

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 `launchDiscovery`. Alternatively, you can start the HDL simulator manually and load the EDA Simulator Link DS libraries yourself. Either way, see "Starting the HDL Simulator" on page 1-15.

Determining an Available Socket Port Number

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

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.

Managing a Simulink Cosimulation Session

To run and test a cosimulation model 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.

If you change any part of the Simulink model, including the HDL
Cosimulation block parameters, rerun the simulation or click the Update

diagram button [◈] or select **Edit > Update Diagram** so that the diagram
reflects those changes.

MATLAB Function Reference

breakHDLSim

Purpose	Execute stop command in HDL simulator
Syntax	breakHdlSim() breakHdlSim('portNumber') breakHdlSim('portNumber','hostName')
Description	breakHDLSim executes a stop command on the HDL simulator. breakHdlSim() connects to the HDL simulator using a shared connection. breakHdlSim('portNumber') connects to the local host on port *portNumber*. breakHdlSim('portNumber','hostName') connects to the host *hostName* on port *portNumber*.
Examples	>> breakHdlSim >> breakHdlSim('1234') >> breakHdlSim('1234', 'mylinux')
See Also	pingHdlSim

dec2mvl

Purpose
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

hdldaemon

Purpose   Start MATLAB server component of EDA Simulator Link DS software

Syntax
```
hdldaemon
hdldaemon('PropertyName', 'PropertyValue'...)
hdldaemon('status')
hdldaemon('kill')
```

Description   Use `hdldaemon` to activate servers, check link status, and shutdown servers.

Server Activation

`hdldaemon` starts the MATLAB server component of the EDA Simulator Link DS software with the following default settings:

- Shared memory communication enabled

- Time resolution for the MATLAB simulation function output ports set to scaled (type `double`)

You can use TCP/IP on a single system (one that runs both MATLAB and the HDL simulator). However, consider using shared memory communication when your application configuration consists of a single system. This approach can result in increased performance.

Only one `hdldaemon` per MATLAB session can be running at any given time.

Matching Communication Modes and Socket Ports

The communication mode that you specify (shared memory or TCP/IP sockets) must match what you specify for the communication mode when you issue the `matlabcp`, `matlabtb`, or `matlabtbeval` command in the HDL simulator.

In addition, 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.

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 "Associating a MATLAB Link Function with an HDL Module" on page 2-35.

hdldaemon('PropertyName', 'PropertyValue'...) starts the EDA Simulator Link DS MATLAB server component with property-value pair settings. These settings specify the communication mode for the link between MATLAB and the HDL simulator and the resolution of tnow (the current time argument passed by the associated m-function). See hdldaemon "Property Name/Property Value Pairs" on page 4-6 for details.

Link Status

hdldaemon('status') returns the following message indicating that a link (connection) exists between MATLAB and the HDL simulator:

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

You can also use this function to check on the communication mode being used, the number of existing connections, and the interprocess communication identifier (ipc_id) being used for a link by assigning the return value of hdldaemon to a variable. The ipc_id identifies a port number for TCP/IP socket links or the file system name for a shared memory communication channel. For example:

```
x=hdldaemon('status')
x =
           comm: 'sockets'
    connections: 0
         ipc_id: '4449'
```

This function call indicates that the server is using TCP/IP socket communication with socket port 4449 and is running with no active HDL simulator clients. If you are using a shared memory link, set the value of comm to 'shared memory' and make the value of ipc_id a

file system name for the shared memory communication channel. For example:

```
x=hdldaemon('status')
HDLDaemon shared memory server is running with 0 connections
x =
            comm: 'shared memory'
     connections: 0
          ipc_id: [1x45 char]
```

Server Shutdown

`hdldaemon('kill')` shuts down the MATLAB server without shutting down MATLAB.

Property Name/ Property Value Pairs

`hdldaemon` accepts the following valid property name/property value pairs:

`'socket'`, `tcp_spec`

Specifies the TCP/IP socket mode of communication for the link between MATLAB and the HDL simulator. If you omit this argument, the server uses the shared memory mode of communication.

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

The `tcp_spec` 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 C-5 for some valid `tcp_spec` examples.

For more information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page C-2.

> **Note** If you specify the operating system option ('O' or 0), use hdldaemon('status') to acquire the assigned socket port number. You must specify this port number when you issue a link request with the matlabtb or matlabtbeval command in the HDL simulator.

'time', 'sec' | 'time', 'int64'
  Specifies the time resolution for MATLAB function ports and simulation times (tnow).

Specify...	For...
'time' 'sec' (default)	A double value that is scaled to seconds based on the current HDL simulation resolution
'time' 'int64'	64-bit integer representing the number of simulation steps

  If you omit this argument, the server uses scaled resolution time.

'quiet', 'true'
  Suppresses printing messages to the standard queue. Errors still appear.

The following table provides guidelines on when and how to specify property name/property value pairs.

hdldaemon

If Your Application Is to...	Do the Following...
Operate in shared memory mode	Omit the `'socket'`, *tcp_spec* property name/property value pair. The interface operates in shared memory mode by default. You should use shared memory mode if your application configuration consists of a single system and uses a single communication channel.
Operate in TCP/IP socket mode, using a specific TCP/IP socket port	Specify the `'socket'`, *tcp_spec* property name and value pair. The *tcp_spec* can be a socket port number or service name. Examples of valid port specifications include `'4449'`, `4449`, and `MATLAB Service`. For information on choosing a TCP/IP socket port, see "Choosing TCP/IP Socket Ports" on page C-2.
Operate in TCP/IP socket mode, using a TCP/IP socket that the operating system identifies as available	Specify `'socket'`, `0` or `'socket'`, `'0'`.
Return time values in seconds (type `double`)	Specify `'time'`, `'sec'` or omit the parameter. This is the default time value resolution.
Return 64-bit time values (type `int64`)	Specify `'time'`, `'int64'`.
Suppress server shutdown message when using `hdldaemon` to get an unused socket number (message can appear confusing)	Specify `'quiet'`, `'true'`.

Examples

The following function call starts the MATLAB server with shared memory communication enabled and a 64-bit time resolution format for the MATLAB function's tnow parameter:

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

The following function call starts the MATLAB server with TCP/IP socket communication enabled on socket port 4449. Although you do not need to use TCP/IP socket communication on a single-computer application, you can use that mode of communication locally. Because the function does not specify a time resolution, the HDL simulator applies the default, scaled simulation time resolution to the MATLAB function's output ports:

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

The following function call starts the MATLAB server with TCP/IP socket communication enabled on port 4449. This call also specifies a 64-bit time resolution format:

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

You also can start the server from a script. Consider the following function call sequence:

```
dstat = hdldaemon('socket', 0)
portnum = dstat.ipc_id
```

The first call to hdldaemon specifies that the server use TCP/IP communication with a port number that the operating system identifies and returns connection status information, including the assigned port number, to dstat. The statement on the second line assigns the socket port number to portnum for future reference.

launchDiscovery

Purpose	Launch Synopsys Discovery tools for use with Simulink and MATLAB using EDA Simulator Link DS software
Syntax	pv = launchDiscovery('PropertyName', 'PropertyValue'...) pv = launchDiscovery(PropertyValueStruct)
Description	pv = launchDiscovery('PropertyName', 'PropertyValue'...) generates HDL compile scripts and HDL simulator launch scripts and executes them. These scripts set up an appropriate GCC environment and load the correct EDA Simulator Link DS library into the Synopsys VCS MX simulator. The function returns a structure of properties and their values.

For custom scripting requirements, you can use launchDiscovery to generate template "sh" scripts that you can modify and invoke from MATLAB using a "system" command.

You must use a property name/property value pair with launchDiscovery('PropertyName', 'PropertyValue'...).

pv = launchDiscovery(PropertyValueStruct) both passes and returns a structure of properties and their values.

In batch run modes, the function returns only after the HDL simulator starts and the HDL simulation begins. In interactive run modes, the function returns without waiting for the user to start the HDL simulation.

Property Name/ Property Value Pairs

Required Properties

'LinkType' , 'appname'
    Specifies either Simulink or MATLAB. A Simulink link session includes using the HDL Cosimulation block in a Simulink model for cosimulation with the HDL simulator. A MATLAB link session includes using matlabtb, matlabcp, and matlabtbeval to employ MATLAB functions as callbacks for HDL simulator events.

'VerilogFiles', 'pathname'
> Specifies the full or relative (to "RunDir") path to Verilog files.
> Specify as single string in double quotes or as cell array of
> filenames.

'VhdlFiles', 'pathname'
> Specifies the full or relative (to "RunDir") path to VHDL files.
> Specify as single string in double quotes or as cell array of
> filenames.

'TopLevel', 'modulename'
> Specifies the name of the top-level HDL module.

'AccFile', 'filename'
> Specifies the name of the signal access file that gives cosimulated
> signals read/write/force access to the EDA Simulator Link DS
> application. See the Synopsys VCS MX documentation (search for
> "PLI table") on how to create this file.

Common Optional Properties

'RunDir', 'dirname'
> Specifies the directory from which to execute the compilation and
> launch scripts. This property defaults to an automatically created
> temporary directory.

'RunMode', 'modetype'
> Specifies how to start the HDL simulator. This property accepts
> the following valid values:
>
> - 'Batch': Start the HDL simulator in the background with no
>   window.
>
> - 'Batch with Xterm': Start the HDL simulator in the
>   background but show session in an Xterm.
>
> - 'CLI': Start the HDL simulator in an interactive shell.
>
> - 'GUI': Start the HDL simulator in the Synopsys DVE GUI.
>
> This value defaults to 'GUI'.

'VlogAnFlags', 'flagnames'
> Specifies 'vlogan' flags.

'VhdlAnFlags', 'flagnames'
> Specifies 'vhdlan' flags.

'UumCompFlags', 'flagnames'
> Sets UUM-compatible compilation flags to 'vcs'.

'UumRunFlags', 'flagnames'
> Sets UUM-compatible runtime flags to 'simv'.

'PreSimTcl', 'command'
> Specifies Tcl commands to execute before starting the HDL simulation. Use this property for simple waveform generation statements for signals such as clocks, resets, and enables.

'PingTimeout', 'seconds'
> For Simulink link sessions only. Specifies the number of seconds to wait for the HDL simulator to launch before reporting back an error. To avoid waiting for the simulator to start, use the value of 0. This property defaults to 10 for 'Batch' and 'Batch with Xterm' run modes, and 0 for 'CLI' and 'GUI' run modes.

Advanced Optional Properties

'CosimBlockList', 'blocklist'
> For Simulink links only. Specifies a cell array of HDL Cosimulation block instances that are bound to the HDL simulator about to be built and launched. This value defaults to all cosimulation blocks in the current model. Correct syntax is:
>
>   'CosimBlockList', { 'block1', block2', ... }

'HostComm','commtype'
> For Simulink link sessions only. Specifies the communications mechanism between Simulink and a local HDL simulator. This property accepts the following valid values:

- `'AutoGenSocketPort'`: Find an available TCP port on the current host and program the CosimBlockList with that port. HostCommdefaults to this value.

- `'SharedPipe'`: Program the CosimBlockList to use a shared pipe connection.

- `'GetFromCosimBlock'`: Use whichever communication parameters appear in any existing cosimulation block masks.

- `'<portnumber>'`: Program the CosimBlockList with a numeric socket port value, `'<portnumber>'`, specified as a string.

- `'<servicename>'`: Program the CosimBlockList with an OS TCP/IP service name, `'<servicename>'`, specified as a string.

Note `launchDiscovery` currently does not directly support remote host execution; see "Examples" on page 4-15 section for help in setting up remote connections.

`'HostName', 'name'`
Specifies a remote host name for cross-machine simulations.

`'UserEnv,' 'arrayname'`
Specifies a cell array of VAR=value environment variables for use by the compilation and launch scripts. Correct syntax is:

    'UserEnv', { 'VAR1=val1', 'VAR2=val2', ... }

`'SkipScriptGeneration', true|false`
When true, instructs HDL simulator *not* to write the compilation and launch scripts. This value defaults to false.

`'SkipCompilation', true|false`
When true, instructs the HDL simulator *not* to execute the compilation script. This value defaults to false.

'SkipLaunch', true|false
> When true, instructs the HDL simulator *not* to execute the launch script. This value defaults to false.

The default GCC compiler used is the default VG_GNU_PACKAGE from a standard installation in the VCS tree. If you want to compile using a different version of GCC, you must specify the following properties.

VG_GNU_PACKAGE Properties

'UseDefaultVgGnuPackage', true|false
> Specifies using the default VG_GNU_PACKAGE in the VCS installation tree. See Synopsys documentation for the installation instructions. When you set the UseDefaultVgGnuPackage property to True, the function ignoresVgGnuPackage and VgGnuGccVersion. To guarantee inter-operability of the link application with the Synopsys VCS MX software, keep this property set to True. This value defaults to True.

'VgGnuPackage', 'dirpath'
> Specifies the full directory path to a nondefault installation (an installation outside of the VCS tree) of VG_GNU_PACKAGE. This value defaults to ''. If you do not specify a value for dirpath (if you leave it an empty string), the default value is $VCS_HOME/gnu/{arch} (giving dirpath a dynamic default value, in effect).

'VgGnuGccVersion', 'version'
> Specifies the version of GCC to use. For the 2008.03 VG_GNU_PACKAGE downloads, versions include:
>
> - gcc-3.3.6
> - gcc-4.1.2

Typical use cases for these properties include:

- Use default GCC version in the default VG_GNU_PACKAGE installation location. You specify nothing. Synopsys and the VG_GNU_PACKAGE distribution determine these defaults.

- Use default GCC version in a nondefault VG_GNU_PACKAGE installation location. For this property, you must specify:

  - `'UseDefaultVgGnuPackage', false`

  - `'VgGnuPackage', '/path/to/vg/gnu/installation'`

- Use nondefault GCC version in the default VG_GNU_PACKAGE installation location. For this property, you must specify:

  - `'UseDefaultVgGnuPackage', false`

  - `'VgGnuGccVersion', 'gcc-4.1.2'` (for example)

- Use nondefault GCC version in a nondefault VG_GNU_PACKAGE installation location. For this property, you must specify:

  - `'UseDefaultVgGnuPackage', false`

  - `'VgGnuPackage', '/path/to/vg/gnu/installation'`

  - `'VgGnuGccVersion', 'gcc-3.3.6'` (for example)

Examples   This example compiles and launches a single-file HDL design for cosimulation with Simulink. The code allows the use of Verilog-2000 syntax in the HDL source. This code launches the Synopsys DVE software.

```
>> launchDiscovery( ...
        'LinkType',     'Simulink', ...
        'VerilogFiles', 'myinverter.v', ...
        'VlogAnFlags',  '+v2k', ...
        'TopLevel',     'myinverter', ...
        'AccFile',      'myinverter.acc' ...
    );
```

This next example compiles and launches an HDL design in batch mode. In batch mode, the HDL simulator exits after the simulation

completes, thus the example relaunches the simulation by calling launchDiscovery again with the previously returned property/value structure.

To run cosimulation after HDL simulator has exited:

```
>> pv = launchDiscovery( ...
        'LinkType',  'Simulink', ...
        'VhdlFiles', '"mymultiplier.vhd mymultiplier_tb.vhd"', ...
        'TopLevel',  'mymultiplier_tb', ...
        'AccFile',   'mymultiplier.acc', ...
        'RunMode',   'Batch', ...
      );
```

To rerun cosimulation:

```
>> pv.SkipScriptGeneration = true;
>> pv.SkipCompilation = true;
>> pv = launchDiscovery(pv);         % relaunch the simulator
```

This next example generates scripts for customizing the environment of a specific project (USER_ENV includes some custom environment). Some common reasons to customize the resultant script include:

- You want to run the scripts on a different platform.

- You want to run the scripts on the same platform but on a remote machine.

- The build and run for the HDL simulator is part of a larger environment involving Perl scripts, makefiles, or LSF.

- You want to run in 32-bit mode on a 64-bit machine

- You want to use some other GCC besides the default VG_GNU_PACKAGE.

```
>> srcDir = '/path/to/src';
>> launchDiscovery( ...
        'LinkType',        'MATLAB', ...
```

```
           'VhdlFiles',        {[srcDir '/top.vhd'], [srcDir '/dut.vhd']}, ...
           'TopLevel',         'top', ...
           'AccFile',          'top.acc', ...
           'RunDir',           '/testruns/myrun', ...
           'UserEnv',          {'LM_LICENSE_FILE=/path/to/license.dat'}, ...
           'SkipCompilation',  true, ...
           'SkipLaunch',       true ...
         );
```

On remote machine, for example, you might use:

```
sh> cd /testruns/myrun
sh> (edit scripts as needed)
sh> . tmwESLDS.compile.sh
sh> . tmwESLDS.launch.sh
```

After you finalize the scripts, you can execute them from MATLAB:

```
>> system('rsh linux100 cd /testruns/myrun ; sh tmwESLDS.compile.sh ; sh tmwESLDS.launch.sh');
```

This example shows the generated compilation script:

```
# AUTO-GENERATED SH SCRIPT FOR Simulink COSIMULATION
#--- EDA Link Environment ---

LAUNCHER_NAME=tmwESLDS
NUM_BITS=64
LINK_LIB_DIR=/matlab/toolbox/discovery/linux64
LINK_SL_FILE=liblfdhdls_vlog_gcc336.so
LINK_ML_FILE=liblfdhdlc_vlog_gcc336.so
BITS_FLAG=-full64
export VG_GNU_PACKAGE=${VCS_HOME}/gnu/linux
COMPILE_SETUP_CMDS=".  ${VG_GNU_PACKAGE}/source_me_${NUM_BITS}.sh"
export LD_LIBRARY_PATH=${VG_GNU_PACKAGE}/gcc-${NUM_BITS}/slib64:${LINK_LIB_DIR}:
      ${LD_LIBRARY_PATH}
LOAD_SL_LIB="-load ${LINK_SL_FILE}:simlinkserver"
LOAD_ML_LIB="-load ${LINK_ML_FILE}:matlabclient"
VHPI_SL_LIB="-vhpi ${LINK_SL_FILE}:simlinkserver"
```

```
VHPI_ML_LIB="-vhpi ${LINK_ML_FILE}:matlabclient"
export SL_LIB_SOCKET=37592
VLOG_FILES=/matlab/toolbox/discovery/discoverydemos/Filter/lowpass_filter.v
VHDL_FILES=
TOP_LEVEL=lowpass_filter
ACC_FILE=/matlab/toolbox/discovery/discoverydemos/Filter/lowpass_filter.pli_acc.tab
VHDLAN_FLAGS=
VLOGAN_FLAGS="+v2k"
UUMCOMP_FLAGS=
UUMRUN_FLAGS=
COMP_DEBUG_FLAGS=-debug_all
LAUNCH_DEBUG_FLAGS="-gui -i tmwESLDS.presim.tcl"

#--- User Environment     ---


eval ${COMPILE_SETUP_CMDS}
vlogan ${BITS_FLAG} ${VLOGAN_FLAGS} ${VLOG_FILES}
vcs ${COMP_DEBUG_FLAGS} ${UUMCOMP_FLAGS} +vpi +applylearn+${ACC_FILE} ${BITS_FLAG}
     ${TOP_LEVEL} ${LOAD_ML_LIB}
```

This example shows the generated launch script:

```
# AUTO-GENERATED SH SCRIPT FOR Simulink COSIMULATION
#--- EDA Link Environment ---

LAUNCHER_NAME=tmwESLDS
NUM_BITS=64
LINK_LIB_DIR=/matlab/toolbox/discovery/linux64
LINK_SL_FILE=liblfdhdls_vlog_gcc336.so
LINK_ML_FILE=liblfdhdlc_vlog_gcc336.so
BITS_FLAG=-full64
export VG_GNU_PACKAGE=${VCS_HOME}/gnu/linux
COMPILE_SETUP_CMDS=".  ${VG_GNU_PACKAGE}/source_me_${NUM_BITS}.sh"
LAUNCH_SETUP_CMDS=".  ${VG_GNU_PACKAGE}/source_me_${NUM_BITS}.sh"
export LD_LIBRARY_PATH=${VG_GNU_PACKAGE}/gcc-${NUM_BITS}/slib64:${LINK_LIB_DIR}:
    ${LD_LIBRARY_PATH}
```

```
LOAD_SL_LIB="-load ${LINK_SL_FILE}:simlinkserver"
LOAD_ML_LIB="-load ${LINK_ML_FILE}:matlabclient"
VHPI_SL_LIB="-vhpi ${LINK_SL_FILE}:simlinkserver"
VHPI_ML_LIB="-vhpi ${LINK_ML_FILE}:matlabclient"
export SL_LIB_SOCKET=37592
VLOG_FILES=/matlab/toolbox/discovery/discoverydemos/Filter/lowpass_filter.v
VHDL_FILES=
TOP_LEVEL=lowpass_filter
ACC_FILE=/matlab/toolbox/discovery/discoverydemos/Filter/lowpass_filter.pli_acc.tab
VHDLAN_FLAGS=
VLOGAN_FLAGS="+v2k"
UUMCOMP_FLAGS=
UUMRUN_FLAGS=
COMP_DEBUG_FLAGS=-debug_all
LAUNCH_DEBUG_FLAGS="-gui -i tmwESLDS.presim.tcl"

#--- User Environment     ---


eval ${LAUNCH_SETUP_CMDS}
simv ${LAUNCH_DEBUG_FLAGS} ${UUMRUN_FLAGS}
```

This example uses the GCC 4.1.2 VG_GNU_PACKAGE.

```
>> launchDiscovery( ...
    'LinkType',                'MATLAB', ...
    'VerilogFiles',            'mydesign.v', ...
    'TopLevel',                'mydesign', ...
    'AccFile',                 'mydesign.acc', ...
    'UseDefaultVgGnuPackage',  false, ...
    'VgGnuGccVersion',         'gcc-4.1.2' ...
    );
```

mvl2dec

Purpose   Convert multivalued logic to decimal

Syntax   mvl2dec('*mv_logic_string*')
                mvl2dec('*mv_logic_string*', *signed*)

Description mvl2dec('*mv_logic_string*') converts a multivalued logic string to a
                positive decimal. If *mv_logic_string* contains any character other than
                '0' or '1', NaN is returned. *mv_logic_string* must be a vector.

                mvl2dec('*mv_logic_string*', *signed*) converts a multivalued logic
                string to a positive or a negative decimal. If *signed* is true, this function
                assumes the first character *mv_logic_string*(1) to be a signed bit of a 2s
                complement number. If *signed* is missing or false, the multivalued logic
                string becomes a positive decimal.

Examples   The following function call returns the decimal value 23:

                    mvl2dec('010111')

                The following function call returns NaN:

                    mvl2dec('xxxxxx')

                The following function call returns the decimal value -9:

                    mvl2dec('10111',true)

See Also   dec2mvl

Purpose   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')
```

pingHdlSim

HDL Simulator Command Extensions Reference

matlabcp

Purpose 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]
```

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 link 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. The HDL simulator always calls the MATLAB function once at the start of the simulation, even if you do not specify a time. Separate multiple time values by a space, for example: `matlabcp vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns`
`-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 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. For TCP/IP socket communication on a single computer, the <tcp_spec> argument 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 that is running the MATLAB server (hdldaemon). See "Specifying TCP/IP Values" on page C-5 for some valid tcp_spec examples.

    For more information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page C-2.

    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 "Starting the MATLAB Server" on page 2-47.

-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. For determining signal transition in:

- Verilog: Z and X are read as 0

- VHDL: Z and X will not create a rate transition

Rate transitions are only from 0 -> 1 and 1-> 0. Specify -rising with the path names of one or more signals defined as a logic type.

-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.

-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.

-mfunc <name>

The name of the MATLAB function that is associated with the HDL module instance you specify for instance. If you omit this argument, matlabcp associates the HDL module instance to a MATLAB function that has the same name as the HDL instance. For example, if the HDL module instance is myfirfilter, matlabcp associates the HDL module instance with the MATLAB function myfirfilter. 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 DS 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
tnext	Write only	Used to schedule a callback during the set time value. This field is equivalent to old tnext. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` will schedule a callback at time equals 5 nanoseconds from tnow.
userdata	Read/Write	Stores state variables of the current `matlabcp` instance. You can retrieve the variables the next time the callback of this instance is scheduled.

Field	Read/Write Access	Description

simstatus	Read only	Stores the status of the HDL simulator. The EDA Simulator Link DS software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. simstatus is a read-only property.
		
		`>> hdl_instance_obj.simstatus`
		
		`ans=`
		`        Init`

instance	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		
		In the HDL simulator:
		
		`hdlsim> matlabcp osc_top -mfunc oscfilter use_instance_obj`
		
		In MATLAB:
		
		`>> hdl_instance_obj.instance`
		
		`ans=`
		`  osc_top`

Field	Read/Write Access	Description

argument	Read only	Stores the argument set by the -argument option of `matlabcp`. For example:
		
		`matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo`
		
		The link software supports the -argument option only when it is used with -use_instance_obj, otherwise the argument is ignored. argument is a read-only property.
		
		`>> hdl_instance_obj.argument`
		
		`ans=`
		`        foo`

portinfo	Read only	Stores information about the VHDL and Verilog ports associated with this instance. portinfo is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information". `hdl_instance_obj.portinfo.field1.field2.field3` **Note** When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.

Field	Read/Write Access	Description

tscale	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. tscale is a read-only property.
		
		`>> hdl_instance_obj.tscale`
		
		`ans=`
		` 1.0000e-009`
		
		Note When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.

tnow	Read only	Stores the current time. tnow is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

matlabcp

Field	Read/Write Access	Description
portvalues	Read/Write	Stores the current values of and sets new values for the output and input ports for a matlabcp instance. For example: ```\n>> hdl_instance_obj.portvalues\n\nans =\nRead Only Input ports:\n clk_enable: []\n clk: []\n reset: []\nRead/Write Output ports:\n sine_out: [22x1 char]\n```
linkmode	Read only	Stores the status of the callback. The EDA Simulator Link DS software sets this field to 'testbench' if the callback is associated with matlabtb and 'component' if the callback is associated with matlabcp. linkmode is a read-only property. ```\n>> hdl_instance_obj.linkmode\n\nans=\n component\n```

-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.

Description   The matlabcp command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator Link DS 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.

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 "Coding an EDA Simulator Link DS MATLAB Application" on page 2-4.

Note For the HDL simulator to establish a communication link with MATLAB, the MATLAB server, `hdldaemon`, must be running with the same communication mode and, if appropriate, the same TCP/IP socket port as you specify with the `matlabcp` command.

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 vlogmatlabc using the -mfunc option. The '-socket' option specifies using socket communication on port 4449.

```
matlabcp vlogtestbench_top.u_matlab_component -mfunc vlogmatlabc -socket 4449
```

matlabcp

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.

```
hldsim> matlabcp vlogtestbench_top  1 2 3 4 5 6 7 -rising outclk3 -falling u_matlab_component/inoutcl
```

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.

```
hdlsim> matlabcp u1_filter1x -mfunc osc_filter -use_instance_obj -argument  oversample=1
hdlsim> matlabcp u1_filter8x -mfunc osc_filter -use_instance_obj -argument  oversample=8
hdlsim> 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

...
```

matlabtb

Purpose	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]
```

Arguments

<instance>

Specifies the instance of an HDL module that the EDA Simulator Link DS 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 link 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 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns`
`-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 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. 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 that is running the MATLAB server (hdldaemon). See "Specifying TCP/IP Values" on page C-5 for some valid tcp_spec examples.

For more information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page C-2.

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 "Starting the
> MATLAB Server" on page 2-47.

`-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.

`-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.

`-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.

`-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.
    For example, if the HDL module instance is `myfirfilter`,
    `matlabtb` associates the HDL module instance with the MATLAB
    function `myfirfilter`. 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 DS 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
tnext	Write only	Used to schedule a callback during the set time value. This field is equivalent to old tnext. For example: `hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9` will schedule a callback at time equals 5 nanoseconds from tnow.
userdata	Read/Write	Stores state variables of the current matlabtb instance. You can retrieve the variables the next time the callback of this instance is scheduled.

Field	Read/Write Access	Description

simstatus	Read only	Stores the status of the HDL simulator. The EDA Simulator Link DS software sets this parameter to 'Init' during the first callback for this particular instance and to 'Running' thereafter. simstatus is a read-only property.
		
		`>> hdl_instance_obj.simstatus`
		
		`ans=`
		`        Init`

instance	Read only	Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:
		
		In the HDL simulator:
		
		`hdlsim> matlabtb osc_top -mfunc oscfilter use_instance_obj`
		
		In MATLAB:
		
		`>> hdl_instance_obj.instance`
		
		`ans=`
		`  osc_top`

Field	Read/Write Access	Description
argument	Read only	Stores the argument set by the -argument option of matlabtb. For example:
| | | ```
matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo
``` |
| | | The link software supports the -argument option only when it is used with -use_instance_obj, otherwise the argument is ignored. argument is a read-only property. |
| | | ```
>> hdl_instance_obj.argument

ans=
     foo
``` |
| portinfo | Read only | Stores information about the VHDL and Verilog ports associated with this instance. portinfo is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the portinfo structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information". |
| | | ```
hdl_instance_obj.portinfo.field1.field2.field3
``` |
| | | **Note** When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo. |

Field	Read/Write Access	Description

tscale	Read only	Stores the resolution limit (tick) in seconds of the HDL simulator. tscale is a read-only property.
		
		`>> hdl_instance_obj.tscale`
		
		`ans=`
		`  1.0000e-009`
		
		--- 	
		Note When you use use_instance_obj, you access tscale through the HDL instance object. If you do not use use_instance_obj, you can still access tscale through portinfo.
		--- 	

tnow	Read only	Stores the current time. tnow is a read-only property. `hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;`

Field	Read/Write Access	Description
portvalues	Read/Write	Stores the current values of and sets new values for the output and input ports for a matlabtb instance. For example: ``` >> hdl_instance_obj.portvalues ans = Read/Write Input ports: clk_enable: [] clk: [] reset: [] Read Only Output ports: sine_out: [22x1 char] ``` For example, you can set the reset port to 1 by calling hdl_instance_obj.portvalues.reset = '1'.
linkmode	Read only	Stores the status of the callback. The EDA Simulator Link DS 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. ``` >> hdl_instance_obj.linkmode ans= testbench ```

-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.

Description   The matlabtb command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator Link DS 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.

MATLAB test bench functions mimic stimuli passed to entities in the HDL model. You force stimulus from MATLAB or HDL scheduled with matlabtb.

Note For the HDL simulator to establish a communication link with MATLAB, the MATLAB server, hldaemon, must be running with the same communication mode and, if appropriate, the same TCP/IP socket port as you specify with the matlabtb command.

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 DS, 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 DS, 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 DS 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
```

Purpose   Call specified MATLAB function once and immediately on behalf of instantiated HDL module

Syntax   matlabtbeval <instance> [-socket <tcp_spec>]
                [-mfunc <name>]

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 C-5 for some valid tcp_spec examples.

For more information on choosing TCP/IP socket ports, see "Choosing TCP/IP Socket Ports" on page C-2.

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 "Starting the MATLAB Server" on page
> 2-47.

`-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. For example, if the HDL module instance is
`myfirfilter`, `matlabtbeval` associates the HDL module instance
with the MATLAB function `myfirfilter` . If you omit this
argument and `matlabtbeval` does not find a MATLAB function
with the same name, the command displays an error message.

Description   The `matlabtbeval` command has the following characteristics:

- Starts the HDL simulator client component of the EDA Simulator
  Link DS 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.

> **Note** The `matlabtbeval` command executes the MATLAB function
> immediately, while `matlabtb` provides several options for scheduling
> MATLAB function execution.

> **Note** For the HDL simulator to establish a communication link with
> MATLAB, the MATLAB `hdldaemon` must be running with the same
> communication mode and, if appropriate, the same TCP/IP socket port
> as you specify with the `matlabtbeval` command.

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:
```

nomatlabtb

Purpose   End active MATLAB test bench and MATLAB component sessions

Syntax   `nomatlabtb`

Description   The `nomatlabtb` command ends all active MATLAB test bench and
                   MATLAB component sessions that were previously initiated by
                   matlabtb or matlabcp commands.

> **Note** This command should be called before shutting down `hdldaemon`
> or `hdldaemon` will block shutdown until the call occurs.

Examples   The following command ends all MATLAB test bench and MATLAB
                   component sessions:

```
> nomatlabtb
```

See Also   `matlabtb`, `matlabcp`

Simulink Block Reference

HDL Cosimulation

Purpose   Cosimulate hardware component by communicating with HDL module
instance executing in HDL simulator

Library   EDA Simulator Link DS

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.

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.

Note You must make sure that signals being used in cosimulation have read/write access. A tab file is included in the simulation via the required `launchDiscovery` property "AccFile". This rule applies to all signals on the **Ports** pane.

Dialog Box

The Block Parameters dialog box consists of the following tabbed panes of configuration options:

- "Ports Pane" on page 6-3
- "Connection Pane" on page 6-9
- "Timescales Pane" on page 6-13

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 sfix8_En5. 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 asSTD_LOGIC or STD_LOGIC_VECTOR
- Verilog signals of wire or reg type

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:

- **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 3-35 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 DS 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 3-30.

I/O Mode

Select either Input or Output.

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 Date 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 6-13). See also "Representation of Simulation Time" on page 3-13.

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 selectSigned, the **Fraction Length** edit field becomes available. EDA Simulator Link DS 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 selectUnsigned, the **Fraction Length** edit field becomes available. EDA Simulator Link DS 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.

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 DS 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 C-2.

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.

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 DS defaults to relative timing mode.

- Using *absolute timing mode*

HDL Cosimulation

For more information on calculating relative and absolute timing modes, see "Defining the Simulink and HDL Simulator Timing Relationship" on page 3-14.

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 "Representation of Simulation Time" on page 3-13.

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 DS software calculate the timing relationship for you by performing the following steps:

1 Verify that the HDL simulator is running. EDA Simulator Link DS 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.

Auto Timescale	Automatically calculates a timescale. Click on the help button for more information.

1 second in Simulink corresponds to `0.2` `Tick ▼` in the HDL simulator

EDA Simulator Link DS software analyzes all the clock and port signal rates from the HDL Cosimulation block when it calculates the scale factor.

Note EDA Simulator Link DS 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 DS 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 DS does not support Auto Timescale
calculated from frame-based signals.

For more on the timing relationship between the HDL simulator and
Simulink, see "Representation of Simulation Time" on page 3-13.

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 `10`   `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.

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 | 1 | | s | ▼ | in the HDL simulator

4 Click **Apply** to commit your changes.

To VCD File

Purpose   Generate value change dump (VCD) file

Library   EDA Simulator Link DS

```
        ┌────────────────────┐
    ╲   │                    │
    ╳   │   simulink.vcd     │
    ╱   │                    │
        └────────────────────┘
            To VCD File
```

Description

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 6-21.

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 directory. Specify a complete path name to place the generated file in a different location. If you specify the same name

To VCD File

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
| ```
$date
23-Sep-2003 14:38:11
$end
``` | Data and time the file was generated. |
| ```
$version EDA Simulator Link DS
version 1.0 $ end
``` | Version of the VCD block that generated the file. |
| ```
$timescale 1 ns $ end
``` | The time scale that was used during the simulation. |
| ```
$scope module manchestermodel $end
``` | The scope of the module being dumped. |

Generated VCD File Format (Continued)

File Content	Description

`$var wire 1 ! Original Data [0] $end`
`$var wire 1 " Recovered Clock [0] $end`
`$var wire 1 # Recovered Data [0] $end`
`$var wire 1 $ Data Validity [0] $end`
	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.

`$upscope $end`	Marks a change to the next higher level in the HDL design hierarchy.
`$enddefinitions $end`	Marks the end of the header and definitions section.
`#0`	Simulation start time.

`$dumpvars`
`  0!`
`  0"`
`  0#`
`  0$`
`$end`
	Lists the values of all defined variables at time equals 0.

Generated VCD File Format (Continued)

File Content	Description
| ```
#630
 1!
``` | The starting point of logged value changes from checks of variable values made at each simulation time increment.<br><br>This entry indicates that at 63 nanoseconds, the value of signal Original Data changed from 0 to 1. |
| ```
.
.
.
#1160
 1#
 1$
``` | At 116 nanoseconds the values of signals Recovered Data and Data Validity changed from 0 to 1. |
| ```
$dumpoff
 x!
 x"
 x#
 x$
$end
``` | Marks the end of the file by dumping the values of all variables as the value x. |

To VCD File

Examples

Use this list to find examples in the documentation.

Starting the HDL Simulator Using the MATLAB Function launchDiscovery

matlabcp Examples

matlabtb Examples

EDA Simulator Link DS Machine Configuration Requirements

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

The following list provides samples of valid configurations for using the Synopsys VCS MX HDL simulator and the EDA Simulator Link DS 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 DS 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 DS Software with Simulink Software

The following list provides samples of valid configurations for using the Synopsys VCS MX HDL simulator and the EDA Simulator Link DS 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.

TCP/IP Socket Communication

- "Choosing TCP/IP Socket Ports" on page C-2
- "Specifying TCP/IP Values" on page C-5
- "TCP/IP Services" on page C-6

Choosing TCP/IP 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 DS 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 DS 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 DS 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.

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 DS 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 DS interface to fail to make a connection.

  In such cases the error messages displayed by the EDA Simulator Link DS 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, 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 DS 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 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. 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`

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.

D

Race Conditions in HDL Simulators

Overview

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

For Simulink value changes scheduled to occur at a specific simulation time, the HDL simulator does not make any guarantees as to the order that value change occurs versus some other blocking signal assignment. Thus, if the Simulink values are driven/sampled at the same time as an active clock edge in the HDL, there is a race condition.

For cases where your active HDL clock edge and your intrinsic Simulink active clock edges are at the same frequency, you can 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:

```
Discovery> force top.clk 0 0, 1 2, 0 12 -repeat 20
```

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 Simulink 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

EDA Simulator Link DS
Best Practices

Improving Simulation Speed

HDL simulator users have expressed a desire to increase the speed of their simulation when using EDA Simulator Link DS. However, many factors impact performance of simulation and it is very difficult to generalize performance profiles.

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

Index