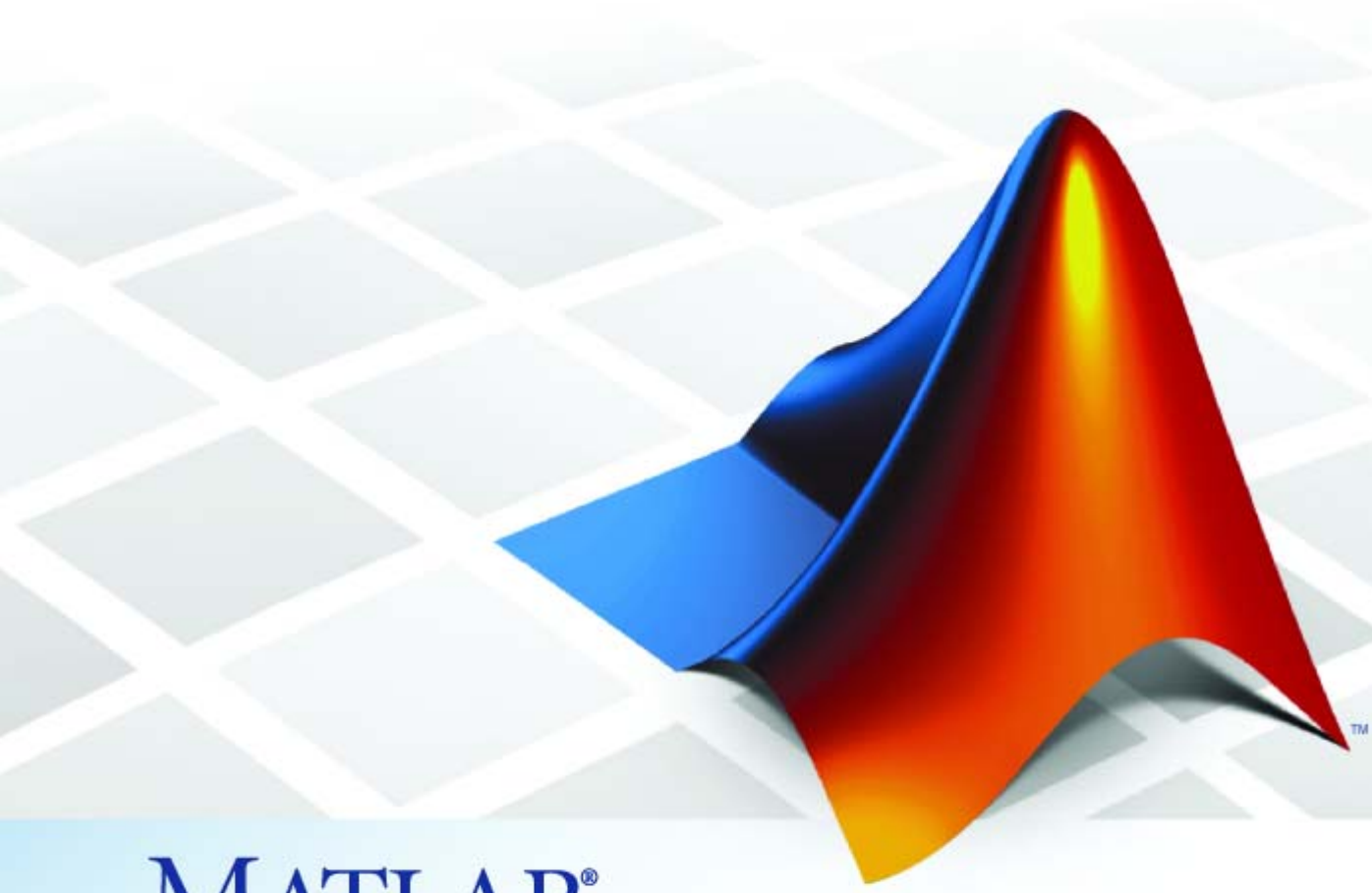
EDA Simulator Link™ DS 1
User's Guide



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.

Revision History

March 2008	Online only	New for Version 1 (Release 2008a)

Contents

1 Getting Started

2 Linking MATLAB® to Discovery™ Simulators

3 Linking Simulink® to Discovery™ Simulators

B TCP/IP Socket Communication

C Race Conditions in HDL Simulators

Index

Getting Started

Product Overview (p. 1-3)	Identifies typical applications and expected users, lists key product features, describes the EDA Simulator Link™ DS cosimulation environment, and provides an overview of how you work with the integrated tool environment
Requirements (p. 1-9)	Describes what you need to know and what other products are required to use the EDA Simulator Link DS software
Setting Up Your Environment for the EDA Simulator Link™ DS Software (p. 1-11)	Explains how to install and set up the EDA Simulator Link DS software
Starting the HDL Simulator (p. 1-15)	Explains and shows how to invoke the HDL simulator so that it will work with EDA Simulator Link DS software
Workflow for Using the EDA Simulator Link™ DS Software with MATLAB® Software (p. 1-17)	Describes very basic steps for creating MATLAB—HDL simulator applications

Workflow for Using the EDA Simulator Link™ DS Software with Simulink® Software (p. 1-18)	Describes very basic steps for creating Simulink—HDL simulator cosimulation sessions
Learning More About the EDA Simulator Link™ DS Software (p. 1-19)	Identifies and explains how to gain access to available documentation online help, demo, and tutorial resources

Product Overview

In this section...

Integration with Other Products

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

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

---

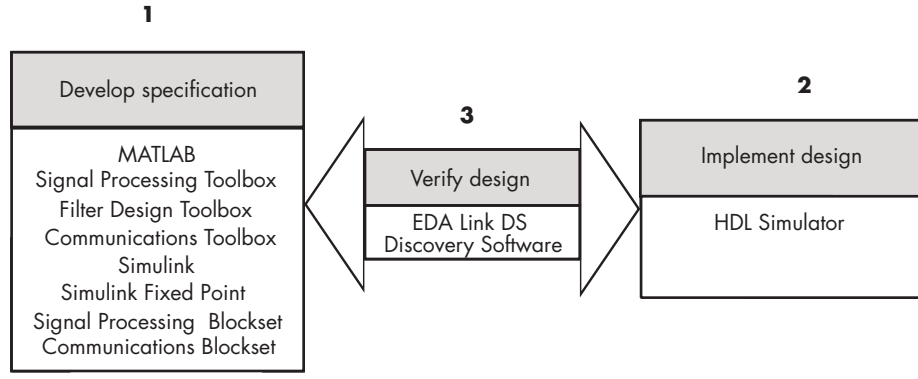
Note Discovery software may also be referred to as "the HDL simulator" throughout this document.

The EDA Simulator Link DS software consists of MATLAB functions and HDL simulator commands for establishing the communication links between the HDL simulator and The MathWorks products. In addition, a library of Simulink blocks is available for including HDL simulator designs in Simulink models for cosimulation.

EDA Simulator Link DS software streamlines FPGA and ASIC development by integrating tools available for

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

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



As the figure shows, EDA Simulator Link DS software connects tools that traditionally have been used discretely to perform specific steps in the design process. By connecting the tools, the link simplifies verification by allowing you to cosimulate the implementation and original specification directly. The end result is 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 use the following:

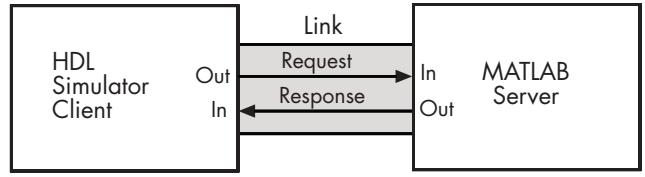
- MATLAB or Simulink to create test signals and software test benches for HDL code
- MATLAB or Simulink to provide a behavioral model for an HDL simulation
- MATLAB analysis and visualization capabilities for real-time insight into an HDL implementation
- Simulink to translate legacy HDL descriptions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

---

Note You can cosimulate a module using SystemVerilog and/or SystemC 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.

Linking with MATLAB and the HDL Simulator

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



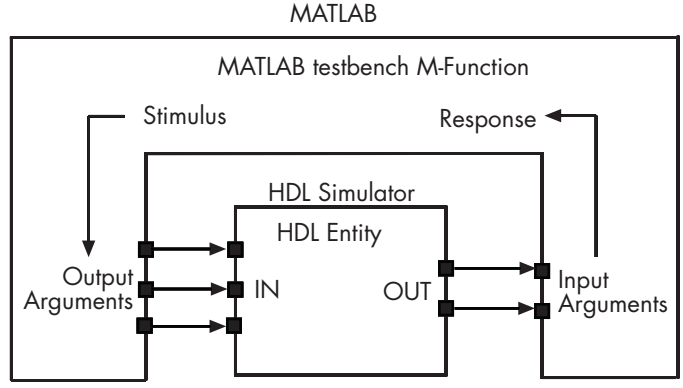
In this scenario, a MATLAB server function waits for service requests that it receives from an HDL simulator session. After receiving a request, the server establishes a communication link, and invokes a specified MATLAB function that computes data for, verifies, or visualizes the HDL module (coded in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

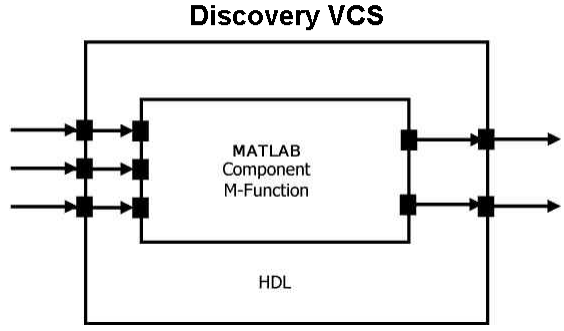
- Tcl commands that execute as part of startup

- A specific Discovery 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:

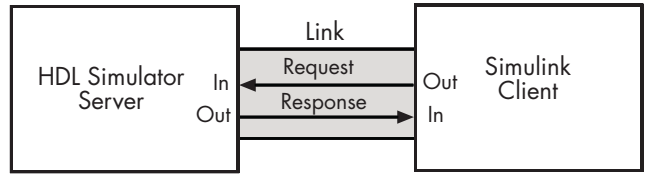
- Load the HDL simulator with an instance of an HDL module to be tested with MATLAB
- Begin a MATLAB test bench or component session for that instance

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

- 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 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

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
- Discovery™ 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 VCSMX® family of simulators Visit the EDA Simulator Link DS requirements page on The MathWorks Web site for specific versions supported with the current release.
Application software required for cosimulation	MATLAB Simulink Simulink Fixed Point Fixed-Point Toolbox
Optional application software	Communications Blockset Signal Processing Blockset Filter Design Toolbox Signal Processing Toolbox Video and Image Processing Blockset

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 compatability 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

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 MathWorks™ products, see the MATLAB installation instructions.

Note In addition to making sure Discovery VCS 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, gcc346, 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. A default library version is understood by the `launchDiscovery` MATLAB command.

Library Names

The EDA Simulator Link DS libraries are named according to the following format:

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

where

`productdir`	discovery
`arch`	linux32, linux64
`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, gcc346, gcc412
`libext`	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)
	
	`liblfdhdlc_vlog_gcc346.so`
	`liblfdhdlc_mixed_gcc346.so`
	
	`liblfdhdlc_vlog_gcc412.so`
	`liblfdhdlc_mixed_gcc412.so`
		`liblfdhdls_vlog_gcc336.so` (for Verilog designs)
		`liblfdhdls_mixed_gcc336.so` (for VHDL or mixed-language designs)
		
		`liblfdhdls_vlog_gcc346.so`
		`liblfdhdls_mixed_gcc346.so`
		
		`liblfdhdls_vlog_gcc412.so`
		`liblfdhdls_mixed_gcc412.so`

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 examples 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

In this section...

Starting Discovery VCS from MATLAB

Start Discovery™ simulators 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 Discovery VCS 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 will get an error if you try to use the function without them.

See "launchDiscovery Examples" 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/Discovery VCS sessions.

launchDiscovery 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.

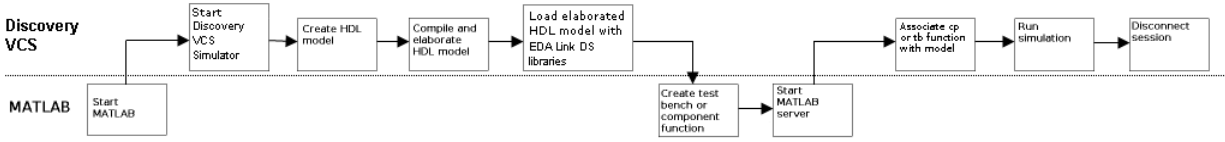
```
>> launchDiscovery( ...
        'LinkType',     'Simulink', ...
        'VerilogFiles', 'myinverter.v', ...
        'VlogAnFlags',  '+v2k', ...
        'TopLevel',     'myinverter', ...
        'AccFile',      'myinverter.acc' ...
    );
```

Starting Discovery VCS from a Shell

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

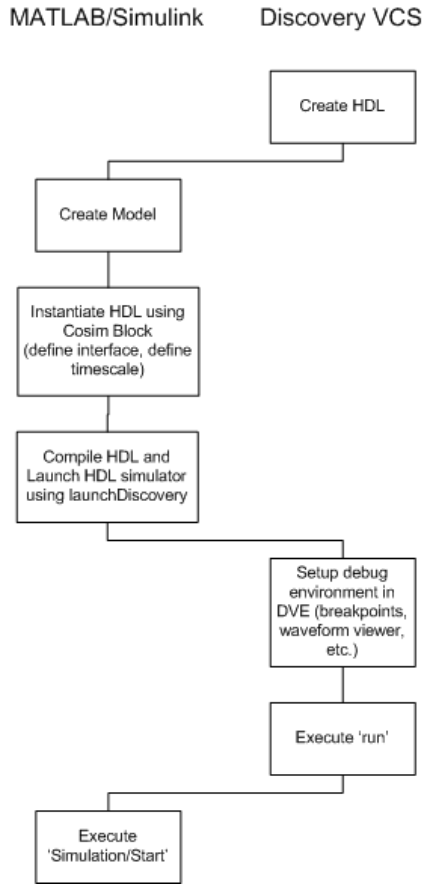
Workflow for Using the EDA Simulator Link™ DS Software with MATLAB® Software

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



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

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 Discovery™ 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® to Discovery™ Simulators"	Explains how to use the HDL simulator and Simulink for cosimulation modeling.
Chapter 4, "EDA Simulator Link™ DS MATLAB® Function Reference"	Describes EDA Simulator Link DS functions for use with MATLAB.

Chapter 5, “EDA Simulator Link™ DS Command Extensions for the HDL Simulator Reference”	Describes EDA Simulator Link DS commands for use with the HDL simulator.
Chapter 6, “EDA Simulator Link™ DS Simulink® Block Reference”	Describes EDA Simulator Link DS blocks for use with Simulink.
Appendix A, “EDA Simulator Link™ DS Machine Configuration Requirements”	Explains the machine configurations permissible when linking the HDL simulator to MATLAB or Simulink
Appendix B, “TCP/IP Socket Communication”	Provides information for choosing TCP/IP socket ports.
Appendix C, “Race Conditions in HDL Simulators”	Describes ways to avoid race conditions in hardware cosimulations with MATLAB and Simulink

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.
- M-help for EDA Simulator Link DS MATLAB functions and HDL simulator commands. This help is accessible with the MATLAB `doc` and `help` commands. For example, enter the command

  ```
  doc hdldaemon
  ```

  or

  ```
  help hdldaemon
  ```

  at the MATLAB command prompt.
- 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 Discovery™ Simulators

MATLAB®-Discovery™ Workflow (p. 2-2)	Provides a high-level view of the steps involved in coding and running MATLAB® functions for use with the EDA Simulator Link™ DS interface.
Coding an EDA Simulator Link™ DS MATLAB® Application (p. 2-4)	Explains how to code HDL modules and MATLAB functions for use with EDA Simulator Link DS software. Provides details on how the EDA Simulator Link DS interface maps HDL data types to MATLAB data types and vice versa.
Associating a MATLAB® Link Function with an HDL Module (p. 2-35)	Describes scheduling and communications options for a MATLAB link session with Discovery VCS.
Running MATLAB® Link Sessions (p. 2-46)	Explains how to start and control Discovery VCS and MATLAB link sessions.

MATLAB®-Discovery™ Workflow

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

In MATLAB...	In Discovery VCS...
	1 Create the HDL model. **2** Compile and elaborate the HDL model.

In MATLAB...	In Discovery VCS...
3 Start the MATLAB application, invoke the Discovery™ 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 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

In this section...

Overview

The EDA Simulator Link™ DS software provides a means for verifying and visualizing Discovery™ HDL modules within the MATLAB® environment. You do this 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.

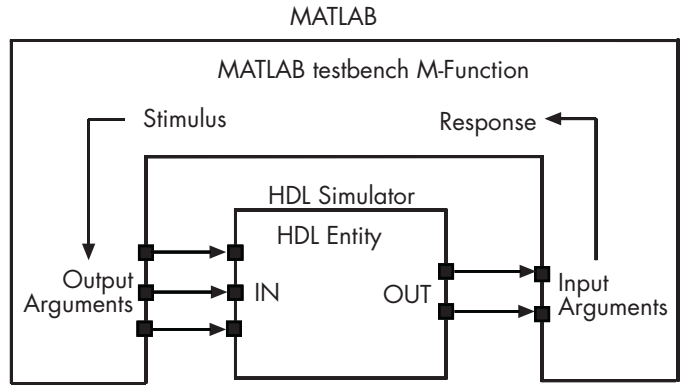
Refer to Appendix A, "EDA Simulator Link™ DS Machine Configuration Requirements" for valid machine configurations.

Process for Coding an EDA Simulator Link™ DS MATLAB Application

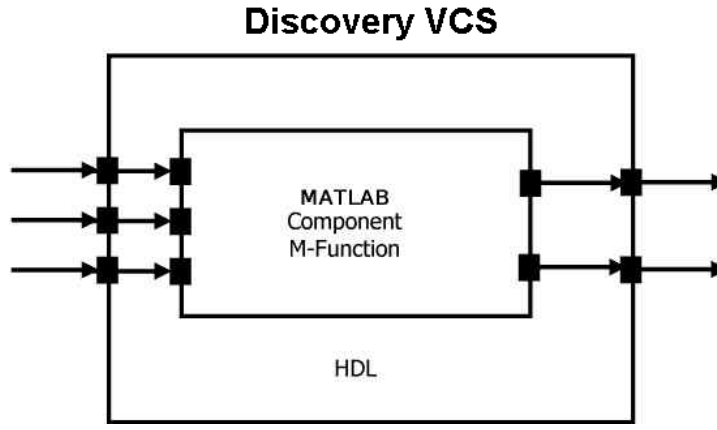
This section provides an overview of the steps required to develop an HDL model for use with MATLAB and the EDA Simulator Link DS software. To program the HDL component of an EDA Simulator Link DS application,

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

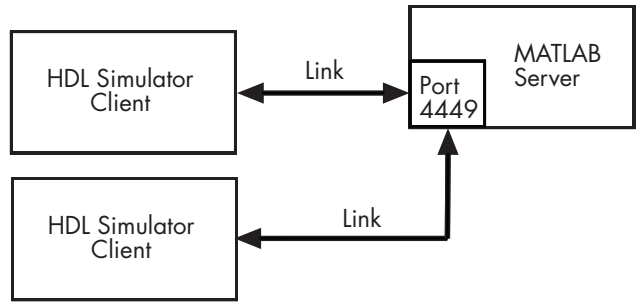
The following figure shows how a MATLAB function wraps around and communicates with the HDL simulator during a test bench simulation session.



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



When linked with MATLAB, the HDL simulator functions as the client, 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 adhere to 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 Discovery VCS 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 A, “EDA Simulator Link™ DS Machine Configuration Requirements” for valid machine configurations.

Coding HDL Modules for MATLAB Verification

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 coding 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 the 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 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

Basic identifiers for VHDL are supported.

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. Simple identifiers for Verilog are supported.

Sample VHDL Entity Definition

The sample VHDL code fragment below 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 Discovery compiler, see the Discovery VCS 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

When coding a MATLAB function that is to verify or visualize an HDL module or component, you must adhere to specific coding conventions, 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 or).

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)
```

Note that 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.

Parameter	Test Bench	Component
`tnow`	*Input* Receives the simulation time at which the MATLAB function is called. By default, time is represented in seconds. For more information see "Scheduling 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. 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.	Same as test bench

Note When importing VHDL signals, signal names in iport, oport, and portinfo are returned in all capitals.

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

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

Note that 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 below.

`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. The information passed in the `portinfo` structure is useful for validating the entity or module under simulation. The information is supplied in three fields, as indicated below. The content of these fields depends on the type of ports defined for the VHDL entity or Verilog module.

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

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

HDL Port Information

Field...	Can Contain...	Which...	And Applies to...
field1	`in`	Indicates the port is an input port	All port types
	`out`	Indicates the port is an output port	All port types
	`inout`	Indicates the port is a bidirectional port	All port types
	`tscale`	Indicates the simulator resolution limit in seconds as specified in the HDL simulator	All types
field2	*portname*	Is the name of the port	All port types
field3	`type`	Identifies the port type For VHDL: `integer`, `real`, `time`, or `enum` For Verilog: `'verilog_logic'` identifies port types `reg`, `wire`, `integer`	All port types
	`right` *(VHDL only)*	The VHDL `RIGHT` attribute	VHDL `integer`, `natural`, or `positive` port types
	`left` *(VHDL only)*	The VHDL `LEFT` attribute	VHDL `integer`, `natural`, or `positive` port types

HDL Port Information (Continued)

Field...	Can Contain...	Which...	And Applies to...
	`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 Discovery VCS application needs to send HDL data to a MATLAB function, it may be necessary for you to first 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 section below).

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.
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, 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. Only scalar data types are supported 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). Be careful when using `matlabtb` and `matlabcp` functions 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, the function may need to convert data to a different type before manipulating it. The following table lists circumstances under which such conversions are required.

Required Data Conversions

If the Function Needs 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 the Function Needs to...	Then...
Convert a standard logic or bit vector to an unsigned integer	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 signed integer	Use the following application of the `mvl2dec` function to convert the data to a signed decimal value. For example: `suval = mvl2dec(oport.val')-2^length(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.

Examples

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

```
InDelayLine(1) = InputScale * mvl2dec(iport.osc_in')/2^(Nbits-1);
```

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, it may be necessary for you to first 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.

VHDL Conversions for the HDL Simulator

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

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

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

Array of STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, BIT_VECTOR, SIGNED, or UNSIGNED	Declare the data as an array of type character with a size that is equivalent to the VHDL port size. See “Array Indexing Differences Between MATLAB and HDL” on page 2-21.

VHDL Conversions for the HDL Simulator (Continued)

To Return Data to an IN Port of Type...	Then...
`INTEGER` or `NATURAL`	Declare the data as an array of type `int32` with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type `int32` with the MATLAB `int32` function before returning it. Be sure to limit the data to values with the range of the VHDL type. If 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);`

`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`

VHDL Conversions for the HDL Simulator (Continued)

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

Enumerated types	Declare the data as a string for scalar ports or a cell array of strings for array ports with each element equal to a label for the defined enumerated type. The `'label'` field of the `portinfo` structure lists all valid labels (see "Gaining Access to and Applying Port Information" on page 2-16). Except for character literals, labels are not case sensitive. In general, you should specify character literals completely, including the single quotes, as shown in the first example below.
	`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. The full text of the code used in this sample can be seen in the section M-Function Example: manchester_decoder.m on page 2-31.

The first step to coding a MATLAB test bench function is to 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.

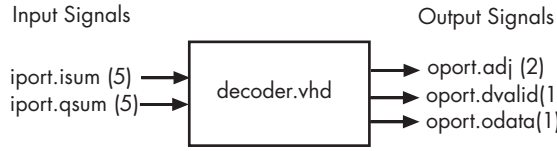
- Defines required argument and return parameters. A MATLAB test bench function *must* return two parameters, `iport` and `tnext`, and pass three arguments, `oport`, `tnow`, and `portinfo`, and *must* appear in the order shown. See "Defining Link Functions and Link Function Parameters" on page 2-13.

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

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

  Recommended practice is to initialize the function outputs at the beginning of the function.

  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.
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.
%
%      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

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

```
    iport.isum = '00000';
    iport.qsum = '00000';
    return;
end

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

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

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

```
        testisdone = 1;
        return;
    end
    iport.isum = dec2bin(isum,5);
    iport.qsum = dec2bin(qsum,5);
else
    iport.isum = dec2bin(isum,5);
end
```

Associating a MATLAB® Link Function with an HDL Module

In this section...

Overview

This section describes establishing a relationship between the link function and the HDL model in the Discovery™ 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 adhere to 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 this, the HDL simulator command `matlabcp` is invoked 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. This is not a file name hierarchy.

Path Specifications for MATLAB Link Sessions with Verilog Top Level

These path specifications rules must be followed:

- 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 are valid signal and module path specification examples:

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

The following are invalid signal and module path specification examples:

```
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:

- 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 are valid signal and module path specification examples:

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

The following are invalid signal and module path specification examples:

```
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 hostname or Internet address in addition to the socket port. See Appendix B, "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: `10 ns, 10 ms, 10 sec` The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero. **Note** For time-based parameters, you can specify any standard time units (`ns`, `us`, and so on). If you do not specify units, the command treats the time value as a value of HDL simulation ticks.

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 Discovery VCS 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:

`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` is 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

In this section...

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 Discovery VCS for use with MATLAB.

5 Load an HDL model in Discovery VCS 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 B-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 Discovery VCS for Use with MATLAB

Start Discovery VCS 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 are ready to 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 Discovery VCS main window.

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

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

A typical sequence for running a simulation interactively from the main HDL simulator window is shown below:

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. The following list demonstrates some ways you can set breakpoints in the MATLAB environment:

- Click next to an executable statement in the breakpoint alley of the Editor/Debugger
- Click the Set/Clear Breakpoint button on the toolbar
- Select **Set/Clear Breakpoint** on the **Breakpoints** menu
- Select **Set/Clear Breakpoint** on the context menu
- Call the `dbstop` function

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. The following list demonstrates some ways you can step through code in the MATLAB environment.

- Click the Step, Step In, or Step Out toolbar button
- Select the **Step**, **Step In**, or **Step Out** option on the **Debug** menu
- Select the **Go Until Cursor** menu option
- Call the `dbstep` function

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. The following list demonstrates ways you can resume a simulation in the MATLAB environment.

- Click the Continue toolbar button

- Select the **Continue**, **Run**, or **Save and Run** option on the **Debug** menu
- Call the `dbcont` function

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 preTCL commands specified in `launchdiscovery`. Therefore, if `matlabtb` was included in the preTCL 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® to Discovery™ Simulators

Simulink®-Discovery™ Workflow

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

In MATLAB...	In Discovery VCS...	In Simulink...
	1 Create the HDL model. **2** Compile and elaborate the HDL model.	
3 Start the MATLAB® application, invoke the Discovery™ 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 Discovery VCS...	In Simulink...
		4 Create a new Simulink model. **5** Add an HDL Cosimulation block (see “Incorporating Hardware Designs into a Simulink® Model” on page 3-25). **6** Define the block interface (see “Defining the Block Interface” on page 3-28). **7** Add other Simulink blocks to complete the Simulink model.

In MATLAB...	In Discovery VCS...	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...

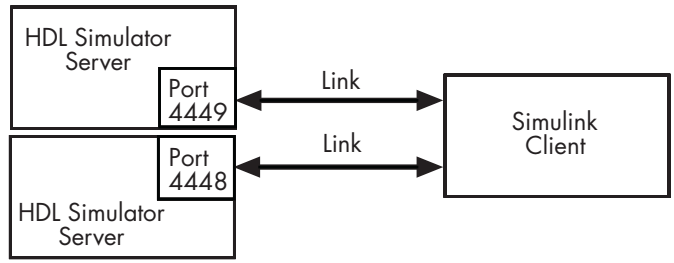
Creating a Hardware Model Design for Use in Simulink® Applications

After you decide to include Simulink® software as part of your EDA flow, think about its role:

- Will you start by developing an HDL application using Discovery™ 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?

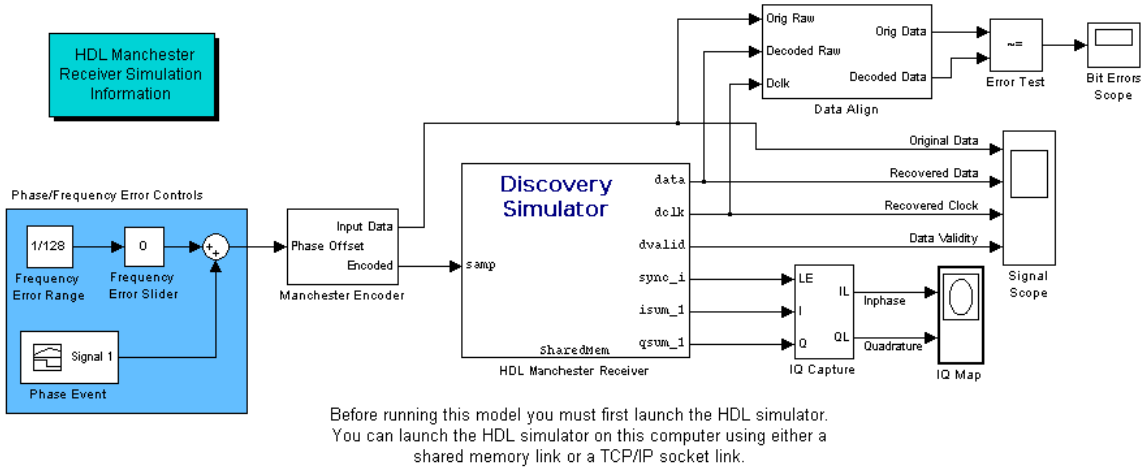
After you answer these questions, use Simulink to build your simulation environment.

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 linked with Simulink, the HDL 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, see the Simulink online help or documentation.

The EDA Simulator Link™ DS HDL Cosimulation Block

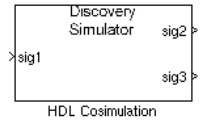
The EDA Simulator Link DS HDL Cosimulation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

Two potential use cases follow:

- A single HDL Cosimulation block fits into the framework of a larger system-oriented Simulink model.
- The Simulink model is a collection of HDL Cosimulation blocks, each representing a specific hardware component.

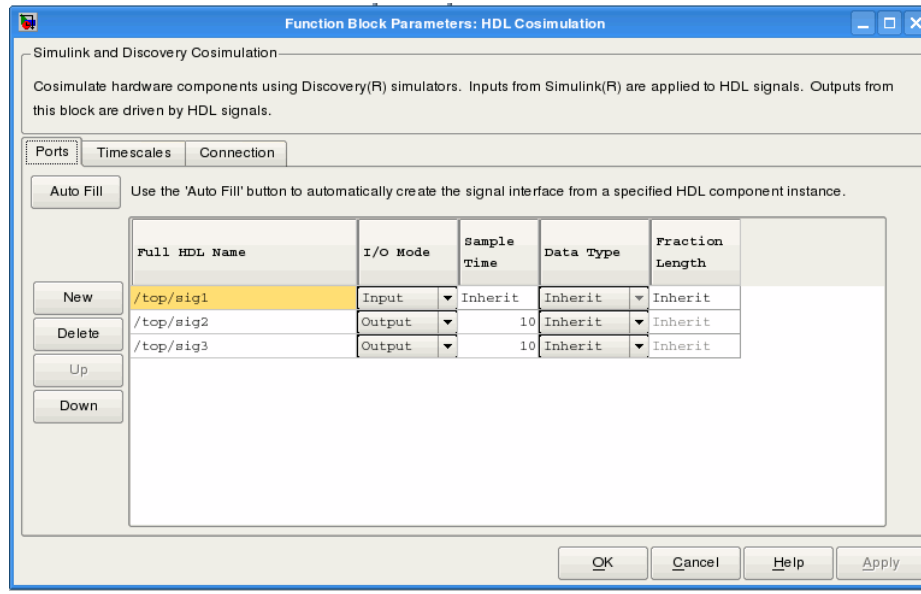
The block mask contains panels for entering port and signal information, setting communication modes, and defining the timing relationship.

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 below.

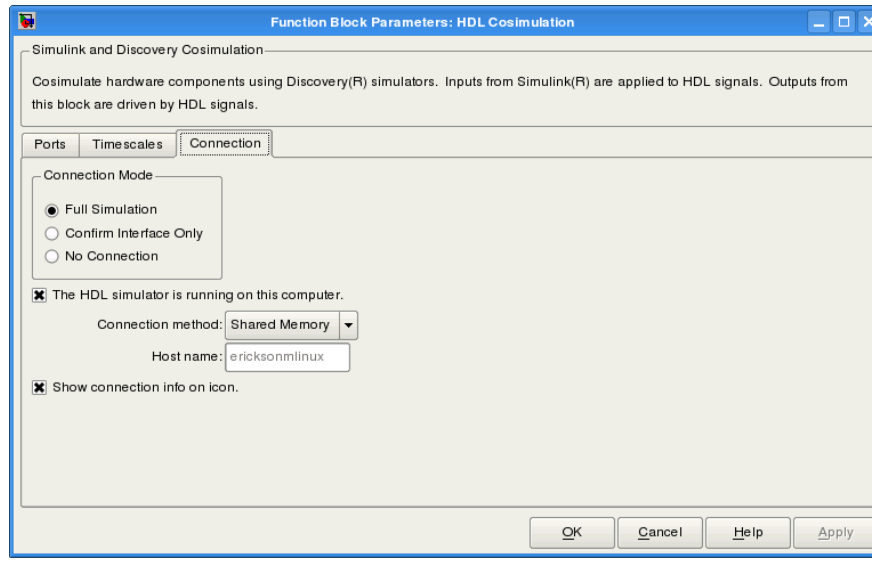


You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog. The HDL Cosimulation block parameters dialog consists of tabbed panes that specify the following:

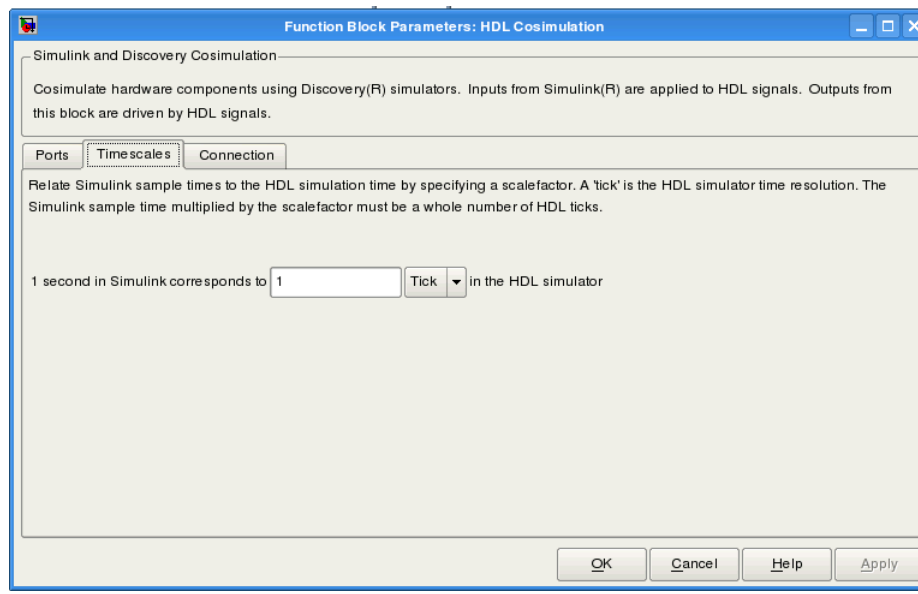
- **Ports Pane**: Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time. See “Ports Pane” on page 6-3 in the Chapter 6, “EDA Simulator Link™ DS Simulink® Block Reference”.



- **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, “EDA Simulator Link™ DS Simulink® Block Reference”.

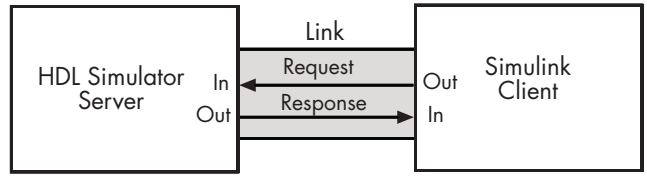


- **Timescales Pane**: Timing relationship between Simulink and the HDL simulator. See “Timescales Pane” on page 6-13 in the Chapter 6, “EDA Simulator Link™ DS Simulink® Block Reference”.



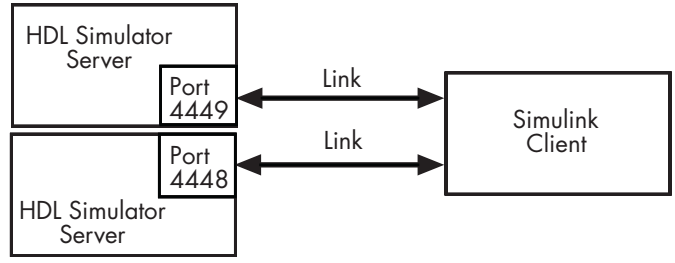
Communicating Between the HDL Simulator and Simulink® Software

When linked with a Simulink application, 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 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

In this section...

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 Discovery™ 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 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 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".

Representation of Simulation Time

The representation of simulation time differs significantly between the HDL simulator and Simulink.

In Discovery VCS, 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.

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

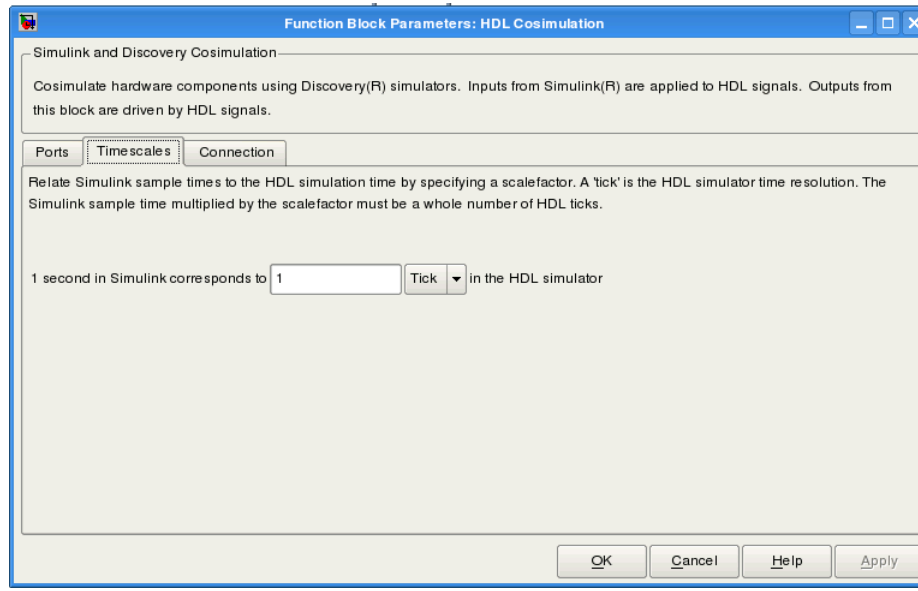
During a simulation run, Simulink communicates the current simulation time to the HDL simulator at each intermediate step. (An intermediate step corresponds to a Simulink sample time hit. Upon each intermediate step, new values are applied at input ports, or output ports are modified.) To bring the HDL simulator up-to-date with Simulink during cosimulation, sampled Simulink time must be converted to HDL simulator time (ticks) and the HDL simulator must run for the computed number of ticks.

Caution If you specify a Simulink sample time that cannot be expressed as a whole number of HDL ticks, you will get an error.

The EDA Simulator Link DS cosimulation interface provides controls that let you configure the timing relationship between the HDL simulator and Simulink and avoid timing errors caused by differences in timing representation.

Defining the Simulink and HDL Simulator Timing Relationship

The **Timescales** pane of the HDL Cosimulation block parameters dialog lets you choose an optimal timing relationship between Simulink and the HDL simulator. The figure below shows the default settings of the **Timescales** pane.



The **Timescales** pane 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*. Relative timing mode is the default.
- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*.

The following sections discuss these two timing modes.

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 quantized to HDL simulator ticks:

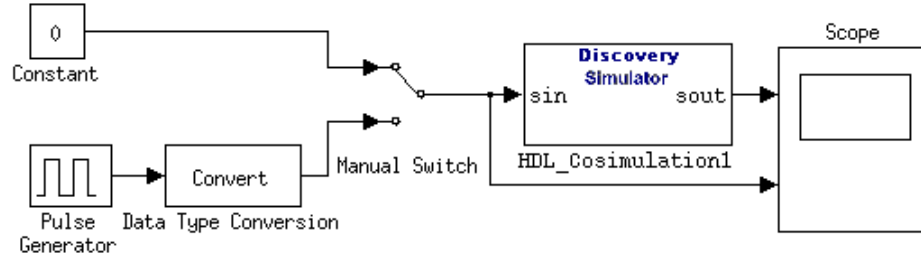
```
InTicks = N * tInSecs
```

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. By default, the HDL Cosimulation block is configured for relative 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 Discovery VCS 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 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.

To configure the **Timescales** parameters for absolute timing mode, you select a unit of absolute time, rather than `Tick`.

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 understand the operation of absolute timing mode, we will again consider the example model discussed in "Operation of Relative Timing Mode" on page 3-15. 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, Simulink will cosimulate with the HDL simulator for 60 ns. Inputs are sampled at a intervals of 24 ns and outputs are updated at intervals of 12 ns. Clocks are driven at intervals of 10 ns.

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.

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. This 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

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 the input port(s) of the HDL Cosimulation block. All such signals must meet the requirements described in “Frame-Based Processing Requirements and Restrictions” on page 3-20. The HDL Cosimulation block automatically configures its output(s) for frame-based operation at the appropriate frame size.

Note that 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. Samples acquired from the HDL simulator are assembled into frames as required by Simulink. Conversely, output data framed by Simulink is transmitted 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:

- VHDL signals should be specified as scalars, 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

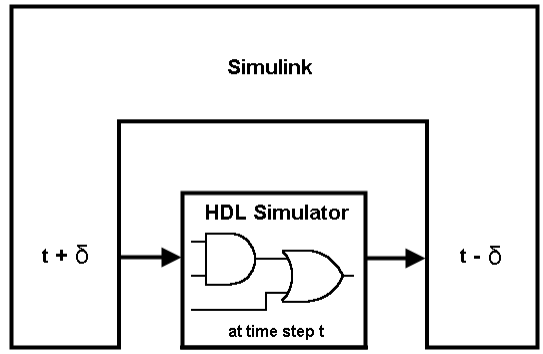
In Discovery VCS, it is not possible to guarantee the order in which clock signals (rising-edge or falling-edge) defined with `launchDiscovery` are applied, relative to the data inputs driven by these clocks. Therefore, if care is not taken to ensure the relationship between the data and active edges of the clock, race conditions could create non-deterministic cosimulation results.

For more on race conditions in hardware simulators, see Appendix C, “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+δ`) and block output reflects signal values just before the current HDL simulator step time (`t-δ`) .



Regardless of whether your HDL code is specified 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 is an artifact of the cosimulation block, which exchanges data with the HDL simulator at the block's output sample time only. This 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 sample frequency for output ports. 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.runn

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 take the approach of designing 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 1-8.

Starting the HDL Simulator

See “Starting the HDL Simulator” on page 1-15.

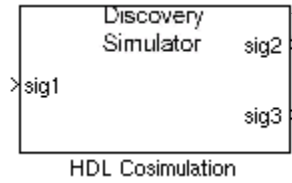
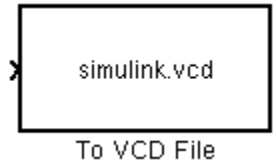
Incorporating Hardware Designs into a Simulink® Model

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 block icons shown below.

	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 3-44.

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 is not a file name hierarchy.

Path Specifications for Simulink Cosimulation Sessions with Verilog Top Level

Path specifications must adhere to the following 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 are valid signal and module path specification examples:

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

The following are invalid signal and module path specification examples:

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

Path Specifications for Simulink Cosimulation Sessions with VHDL Top Level

Path specifications must adhere to the following 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 are valid signal and module path specification examples:

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

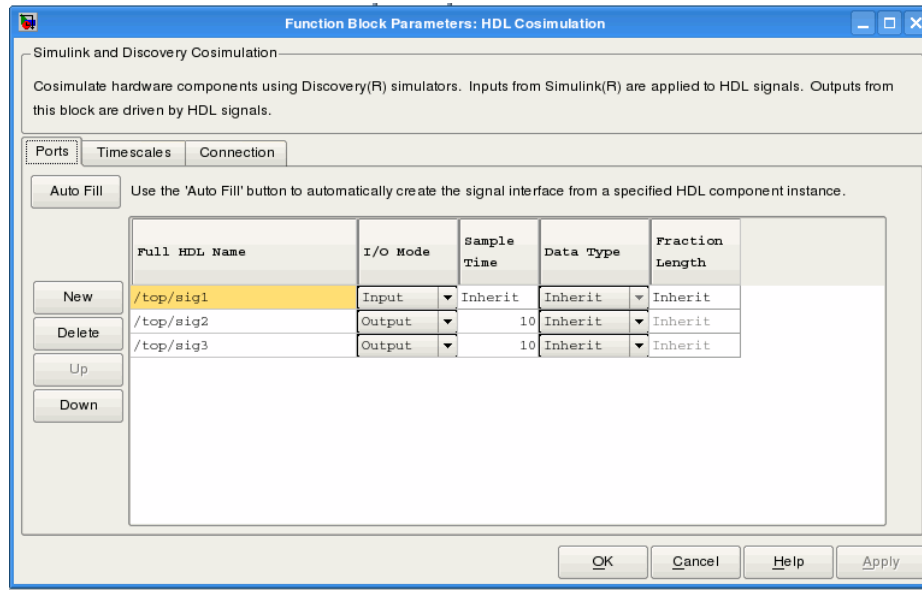
The following are invalid signal and module path specification examples:

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

Defining the Block Interface

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

Simulink displays the following Block Parameters dialog.



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 (see "Entering Signal Information Manually" on page 3-34). 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. Note, however, that 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 3-29 for details.

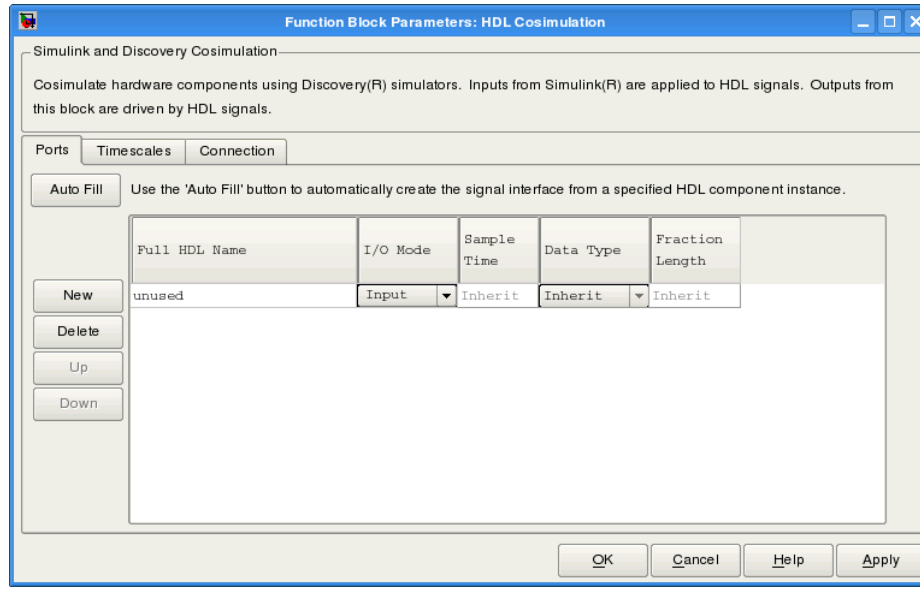
Note Make sure that signals being used in cosimulation have read/write access. A tab file is included in the simulation via the required `launchDiscovery` property "AccFile".

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 required steps are outlined in the example below.

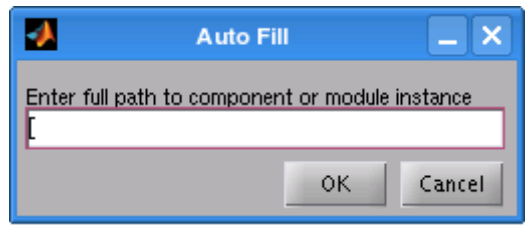
The example is based on a modified copy of the Manchester Receiver model (see "Creating a Hardware Model Design for Use in Simulink® Applications" on page 3-5), in which all signals were first deleted from the **Ports** pane.

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



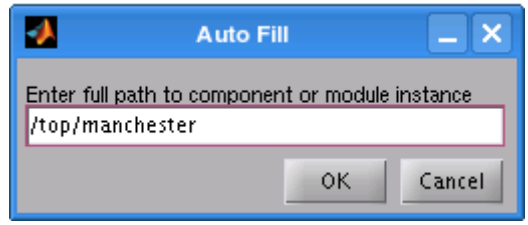
Tip Delete all ports before performing **Auto Fill**. This ensures that no unused signal is present in the Ports list at any time.

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



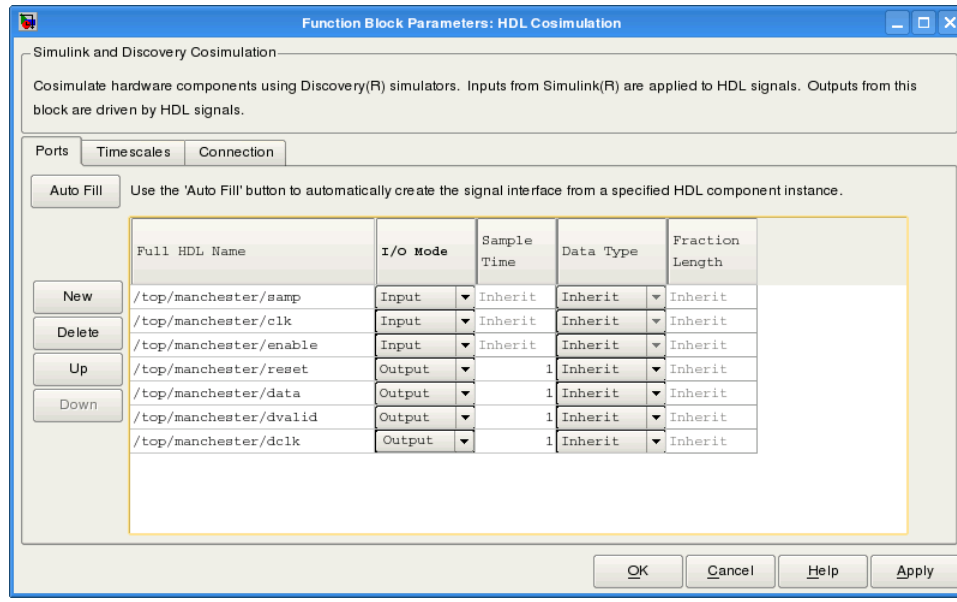
This modal dialog requests an instance path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field. Note this is not a file path and has nothing to do with the source files.

3 In this example, we will obtain port data for a VHDL component called `manchester`. The HDL path is specified as `/top/manchester`.



4 Click **OK**. The dialog is dismissed and the query is transmitted.

5 Port data is returned and entered into the **Ports** pane, as shown in the figure below.



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

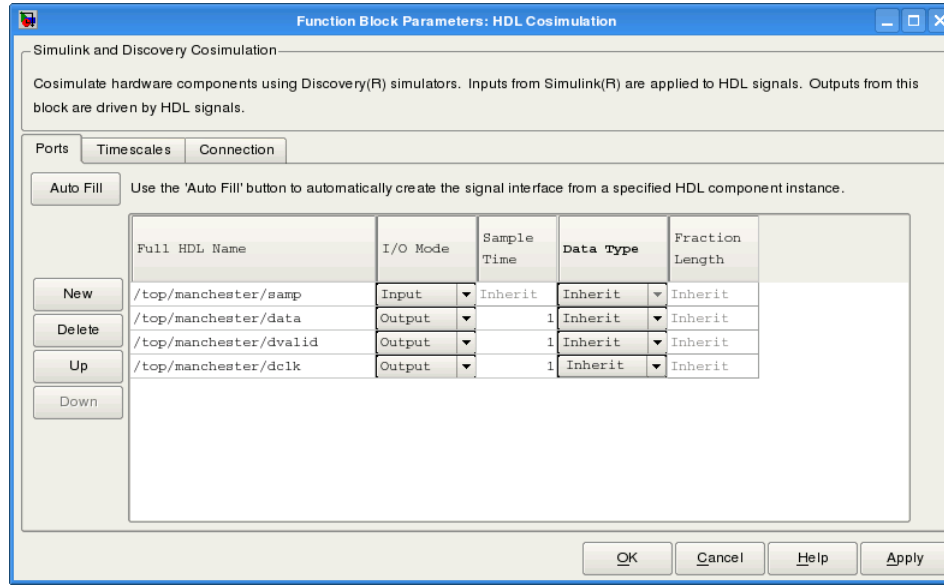
7 Observe that **Auto Fill** has returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the

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.

The figure above 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 shown in the figures below.



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

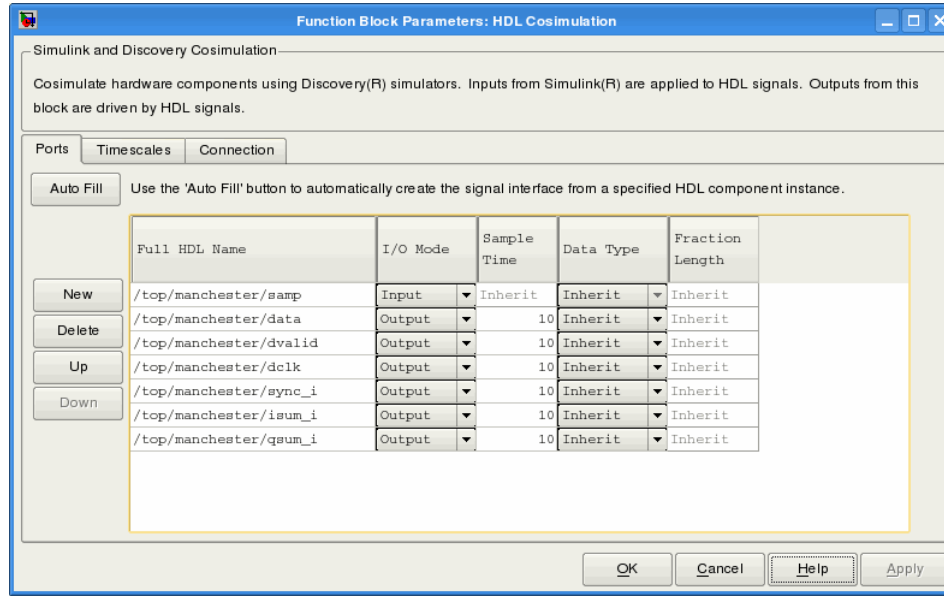
- **Sample time**: `1`
- **Data type**: `Inherit`

- **Fraction length**: `Inherit`

You may need to change these values as required by your model. In this example, the **Sample time** should be set to `10` for all outputs. See also "Specifying the Signal Datatypes" on page 3-36.

9 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 below.

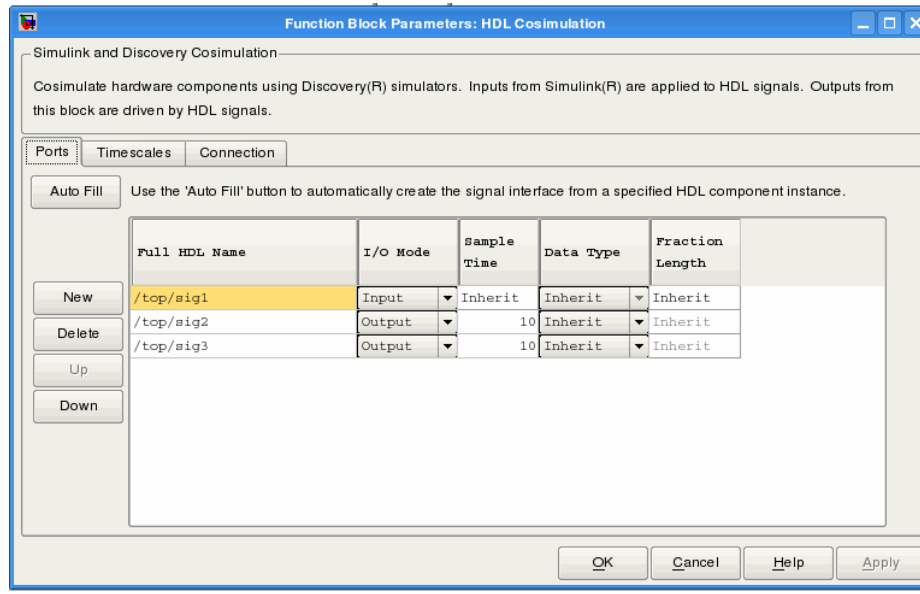
10 Before closing the HDL Cosimulation block parameters dialog, click **Apply** to commit any edits you have made.



Note When importing VHDL signals, signal names are returned 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 for your HDL Cosimulation block, if it is not already open.
3. Select the **Ports** tab of the Block Parameters dialog. Simulink displays the dialog as shown below.



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:

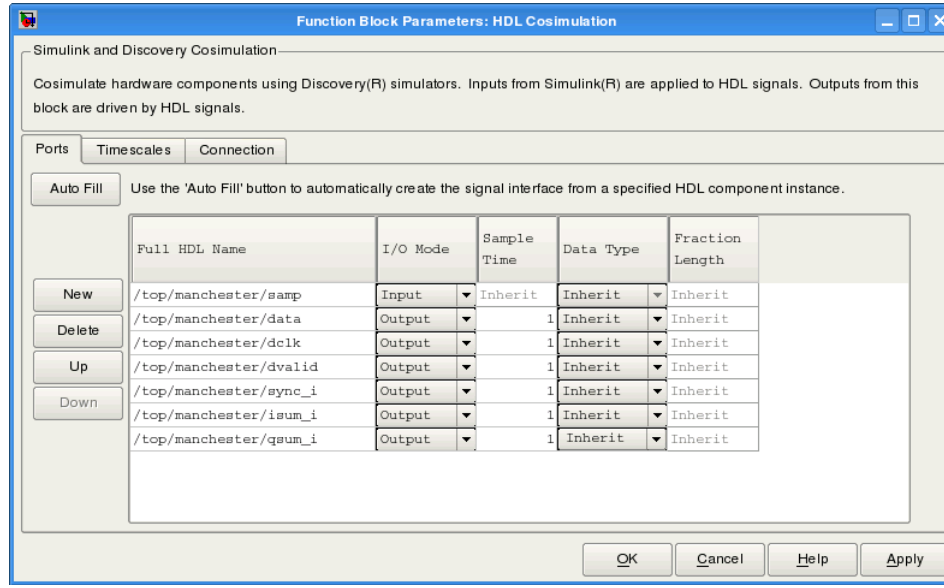
- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types. For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the

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 3-26). 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 desired, set the **Sample Time**, **Data Type**, and **Fraction Length** parameters for signals explicitly, as discussed below.

The following dialog shows port definitions for an HDL Cosimulation block. Note 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 3-12.

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, an 8-bit signal with `Signed` data type and a **Fraction Length** of `5` is assigned the data type `sfix8_En5`. An `Unsigned` 16-bit signal with no fractional part (a **Fraction Length** of `0`) is assigned the data type `ufix16`.

7 Before closing the dialog, click **Apply** to register your edits.

Specifying the Signal Datatypes

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 will apply only to the following:

- 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 `Inherit` is selected, 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 `Signed` is selected, 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 `Unsigned` is selected, the **Fraction Length** edit field is enabled. The port is assigned 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 Discovery VCS Timing Relationship

You configure the timing relationship between Simulink and the HDL simulator by using the **Timescales** pane of the block parameters dialog. Before setting the **Timescales** parameters, you should read "Representation of Simulation Time" on page 3-12 to understand the supported timing modes and the issues that will determine your choice of timing mode.

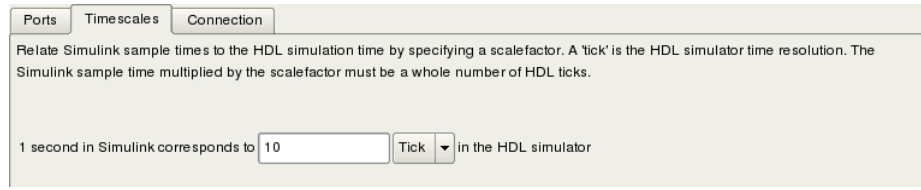
You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator, as described in the sections below.

Specifying a Relative Timing Relationship

To configure relative timing mode for a cosimulation, perform the following steps:

1. Select the **Timescales** tab of the HDL Cosimulation block parameters dialog.
2. Select `Tick` from the list on the right. (This is the default.)
3. Enter a scale factor in the text box on the left. The default scale factor is `1`.

For example, in the figure below, the **Timescales** pane is configured for a relative timing correspondence of `10` HDL simulator ticks to `1` Simulink second.



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

Specifying an Absolute Timing Relationship

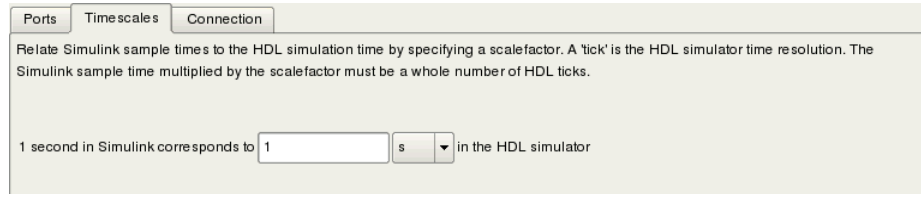
To configure absolute timing mode for a cosimulation, perform the following steps:

1 Select the **Timescales** tab of the HDL Cosimulation block parameters dialog.

2 Select a unit of absolute time from the list on the right. Available units are `fs`, `ps`, `ns`, `us`, `ms`, and `s`.

3 Enter a scale factor in the text box on the left. The default scale factor is `1`.

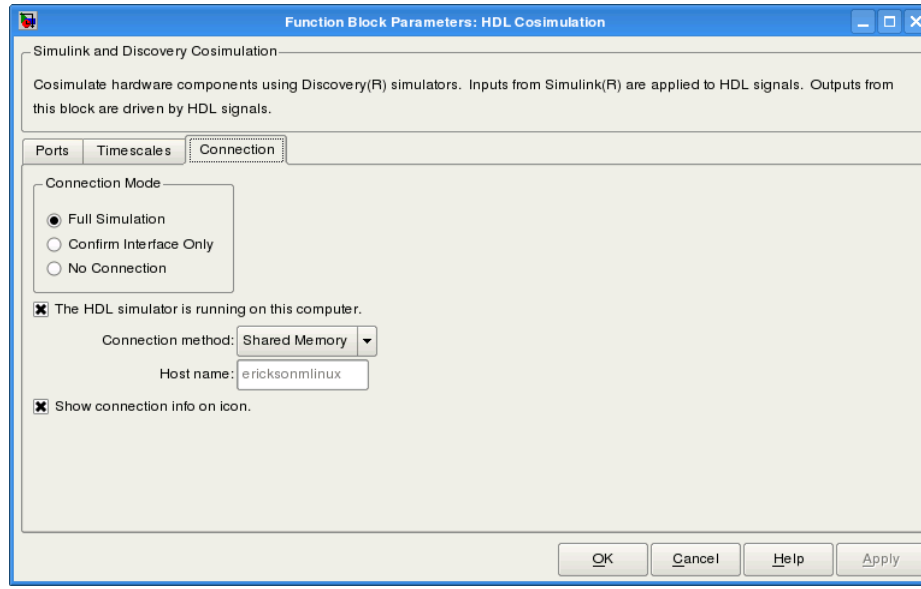
For example, in the figure below, the **Timescales** pane is configured for an absolute timing correspondence of `1` HDL simulator second to `1` Simulink second.



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

Configuring the Communication Link in the HDL Cosimulation Block

Configure a block's communication link with the **Connection** pane of the block parameters dialog.



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 is selected by default.) Note that since Simulink and the HDL simulator are running on different computers, **Connection method** is automatically set to `Socket`.

3. Enter the hostname 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 B-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 1-8.

   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 B-2.

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

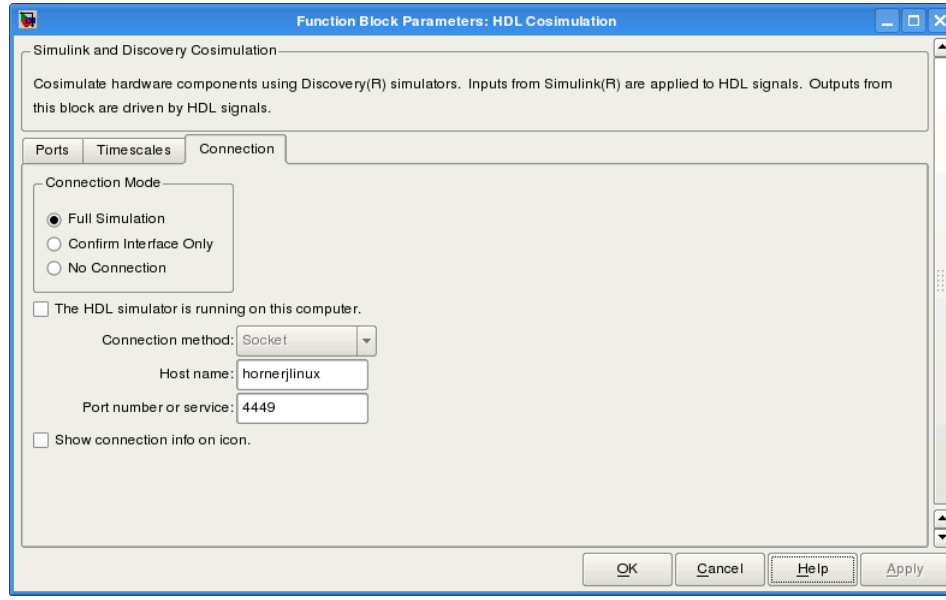
5 If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following:

   - **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 2nd and 3rd options, EDA Simulator Link DS software does not communicate with the HDL simulator during Simulink simulation.

6 Click **Apply**.

The following example dialog 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 is put through 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, but 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 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 examples show the use 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.

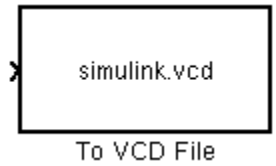
Note The To VCD File block logs changes to states `'1'` and `'0'` only. The block does *not* log changes to states `'X'` and `'Z'`.

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.



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.

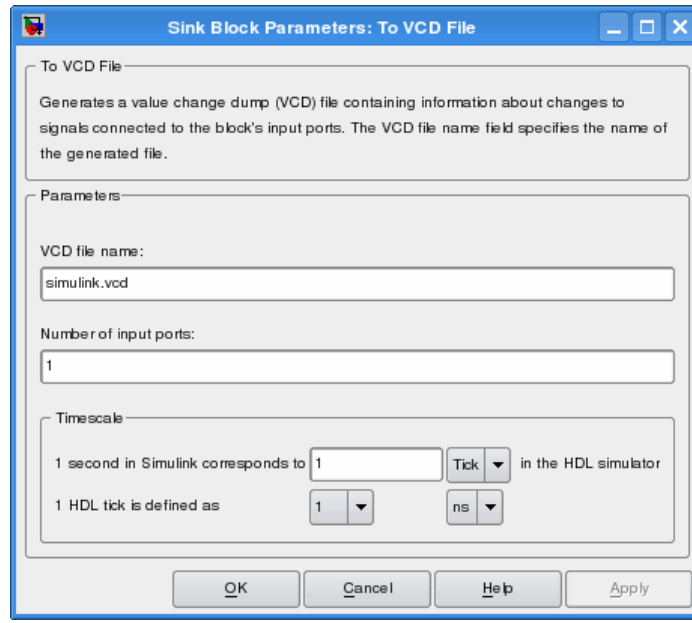
5 Connect the block ports to appropriate blocks in your Simulink model.

Note Because multi-dimensional 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, as follows.

a Double-click the block icon. Simulink displays the following dialog.



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.

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.

---

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 6-18.

Running Cosimulation Sessions

In this section...

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, re-run the simulation or click the Update diagram button  or select **Edit > Update Diagram** so that the diagram reflects those changes.

4

EDA Simulator Link™ DS MATLAB® Function Reference

breakHDLSim

Purpose

Execute a `stop` command on the 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`

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 interface

Syntax

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

Description

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`)

Although you can use TCP/IP on a single system (one that is running both MATLAB and the HDL simulator), using shared memory communication when your application configuration consists of a single system 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 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 the socket port is identified, 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 1-8. 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 that 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 a shared memory link is in use, the value of `comm` is `'shared memory'` and the value of `ipc_id` is 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

The following property name/property value pairs are valid for `hdldaemon`:

`'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 B-4 for some valid `tcp_spec` examples.

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

Note If you specify the operating system option (`'0'` 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 are still shown.

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

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 B-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 it is not necessary to use TCP/IP socket communication on a single-computer application, you can use that mode of communication locally. A time resolution is not specified. Thus, the default, scaled simulation time resolution is applied 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. A 64-bit time resolution format is also specified:

```
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 Discovery VCSMX 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 Discovery 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. Valid values are:

  - "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.

  The default value is "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. The default value is 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. The default value is all cosimulation blocks in the current model.

`'HostComm','commtype'`
For Simulink link sessions only. Specifies the communications mechanism between Simulink and a local HDL simulator. Valid values are:

- "AutoGenSocketPort": Find an available TCP port on the current host and program the CosimBlockList with that port.

- "SharedPipe": Program the CosimBlockList to use a shared pipe connection.
- "GetFromCosimBlock": Use whatever communication parameters are saved in the cosimulation block mask(s).
- "<portnumber>": Program the CosimBlockList with a numeric socket port value, "<portnumber>", specified as a string.
- "<servicename>": Program the CosimBlockList with an OS TCP/IP servicename, "<servicename>", specified as a string.

The default value is "AutoGenSocketPort".

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

`'HostBits', '32'`
: Specifies '32' to run in 32-bit mode on a 64-bit machine.

`'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.

`'SkipScriptGeneration', 'true|false'`
: When true, instructs HDL simulator *not* to write the compilation and launch scripts. The default value is false.

`'SkipCompilation', 'true|false'`
: When true, instructs the HDL simulator *not* to execute the compilation script. The default value is false.

`'SkipLaunch', 'true|false'`
: When true, instructs the HDL simulator *not* to execute the launch script. The default value is 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 the `UseDefaultVgGnuPackage` property is set to "true", `VgGnuPackage` and `VgGnuGccVersion` are ignored. To guarantee inter-operability of the link application with the Synopsys Discovery software, keep this property set to "true". The default value is "true".

`'VgGnuPackage', 'dirpath'`
: Specifies the full directory path to a non-default installation (an installation outside of the VCS tree) of VG_GNU_PACKAGE. The default value is "". The value if you don't specify any (i.e., leave it an empty string) is $VCS_HOME/gnu/{arch}, so in effect it has a dynamic default value.

`'VgGnuGccVersion', 'version'`
: Specifies the version of GCC to use. For the 2008.03 VG_GNU_PACKAGE downloads, versions include:

  - gcc-3.3.6
  - gcc-3.4.6
  - gcc-4.1.2

  The default is the soft link provided by the VG_GNU_PACKAGE installation, "gcc-*HostBits*", where *HostBits* is "32" for 32-bit architectures and "64" for 64-bit architectures.

The expected use-cases for these properties are:

- Use default GCC version in the default VG_GNU_PACKAGE installation location. You specify nothing. These defaults are determined by Synopsys and the VG_GNU_PACKAGE distribution.
- Use default GCC version in a non-default VG_GNU_PACKAGE installation location. For this, you must specify:
  - 'UseDefaultVgGnuPackage', false
  - 'VgGnuPackage', '/path/to/vg/gnu/installation'
- Use non-default GCC version in the default VG_GNU_PACKAGE installation location. For this, you must specify:
  - 'UseDefaultVgGnuPackage', false
  - 'VgGnuGccVersion', 'gcc-4.1.2' (for example)
- Use non-default GCC version in a non-default VG_GNU_PACKAGE installation location. For this, you must specify:
  - 'UseDefaultVgGnuPackage', false
  - 'VgGnuPackage', '/path/to/vg/gnu/installation'
  - 'VgGnuGccVersion', 'gcc-3.4.6' (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 the scripts are finalized, 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('multivalued_logic_string')
mvl2dec('multivalued_logic_string', signed)
```

Description

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

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

Syntax

```
pingHdlSim(timeout)
pingHdlSim(timeout, 'portnumber')
pingHdlSim(timeout, 'portnumber', 'hostname')
```

Description

`pingHdlSim(timeout)` blocks cosimulation by not returning until the Simulink server is loaded or until the specified timeout occurs. `pingHdlSim` returns the process ID of the HDL simulator or -1 if a timeout occurs. You must enter a timeout value.

This function is useful if you are trying to automate a cosimulation and you need to know that the Simulink 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 Simulink server is loaded 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 Simulink server is loaded or until 20 seconds have passed:

```
pingHdlSim(20, '5678')
```

The following function call blocks further cosimulation on port 5678 on hostname msuser until the Simulink server is loaded or until 20 seconds have passed:

```
pingHdlSim(20, '5678', 'msuser')
```

5

EDA Simulator Link™ DS Command Extensions for the HDL Simulator 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>]
```

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`. Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Note Do not specify an instance of an HDL module that has already been associated with a MATLAB test bench 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 specified MATLAB function is called. Each time value is relative to the current simulation time. The MATLAB function is always called once at the start of the simulation, even if you do not specify a time. Multiple time values are separated by a space, for example: `matlabcp vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns`
`-repeat <time>`	Specifies that the MATLAB function be called repeatedly based on the specified `<timen>,...` pattern. The time values are relative to the value of `tnow` at the time the MATLAB function is first called.
`-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 MATLAB function is first called. If you do not specify a cancel time, the command calls the MATLAB function.

Note Time specifications must be placed after the `matlabcp` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the matlabcp functions are represented by 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 B-4 for some valid `tcp_spec` examples.

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

If the HDL simulator and MATLAB are running 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 omit `-socket <tcp_spec>` from 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 1-8. 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 specified MATLAB function is called 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 specified MATLAB function is called when 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 specified MATLAB function is called when any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals in the sensitivity list can be any type and can be at any level in the hierarchy of the HDL model.

`-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.

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

This example associates the Verilog module vlogtestbench_top.u_matlab_component with the MATLAB function `vlogmatlabc` using socket communication on port 4449. The `'-mfunc'` option specifies the m-function to connect to and `'-socket'` option specifies the port number for socket connection mode.

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

This example also associates the component with the MATLAB function but includes explicit times and uses the -cancel option.

```
matlabcp vlogtestbench_top 1e6 fs  3 2e3 ps -repeat 3 ns -cancel 7n
```

This example also associates the component with the MATLAB function and also uses rising and falling edges.

```
matlabcp vlogtestbench_top  1 2 3 4 5 6 7 -rising outclk3 -falling
```

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>]
```

Arguments

`<instance>`

Specifies the instance of an HDL module that is to be associated 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`. Alternatively, you can specify a different MATLAB function with `-mfunc`.

Note Note Do not specify an instance of an HDL module that has already been associated with a MATLAB component 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 specified MATLAB function is called. Each time value is relative to the current simulation time. Even if you do not specify a time, the command calls the MATLAB function once at the start of the simulation. Multiple time values are separated by a space, for example: `matlabtb vlogtestbench_top 1e6 fs 3 2e3 ps -repeat 3 ns -cancel 7ns`
`-repeat <time>`	Specifies that the MATLAB function be called repeatedly based on the specified `<timen>,...` pattern. The time values are relative to the value of `tnow` at the time the MATLAB function is first called.
`-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 MATLAB function is first called. If you do not specify a cancel time, the command calls the MATLAB function.

Note Time specifications must be placed after the `matlabtb` instance and before any additional command arguments; otherwise the time specifications are ignored.

All time specifications for the matlabtb functions are represented by 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 B-4 for some valid `tcp_spec` examples.

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

If the HDL simulator and MATLAB are running 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 omit `-socket <tcp_spec>` from 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 1-8. 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 specified MATLAB function is called 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 specified MATLAB function is called when 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 specified MATLAB function is called when any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals in the sensitivity list can be any type and can be 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.

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, `hdldaemon`, 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 command associates an instance of the entity myfirfilter with the MATLAB function myfirfilter, and initiates 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:

```
> matlabtb myfirfilter 5 ns -repeat 10 ns -socket 4449
```

This example also associates the component with the MATLAB function but includes explicit times and uses the -cancel option.

```
matlabtb vlogtestbench_top 1e6 fs  3 2e3 ps -repeat 3 ns -cancel 7ns
```

This example also associates the component with the MATLAB function and also uses rising and falling edges.

```
matlabcp vlogtestbench_top  1 2 3 4 5 6 7 -rising outclk3 -falling u_ma
```

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 B-4 for some valid `tcp_spec` examples.

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

If the HDL simulator and MATLAB are running 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 omit `-socket <tcp-spec>` from 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 1-8. 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 until this happens.

Examples

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

```
> nomatlabtb
```

See Also

`matlabtb`, `matlabcp`

6

EDA Simulator Link™ DS Simulink® Block Reference

HDL Cosimulation

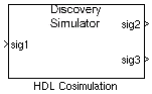
Purpose

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 a *relative* timing relationship (Simulink seconds correspond to an HDL simulator-defined tick interval) or an *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 an input port is connected to a signal that has an explicit sample period, based on forward propagation, Simulink applies that rate to the port.
- If an input port is connected 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.

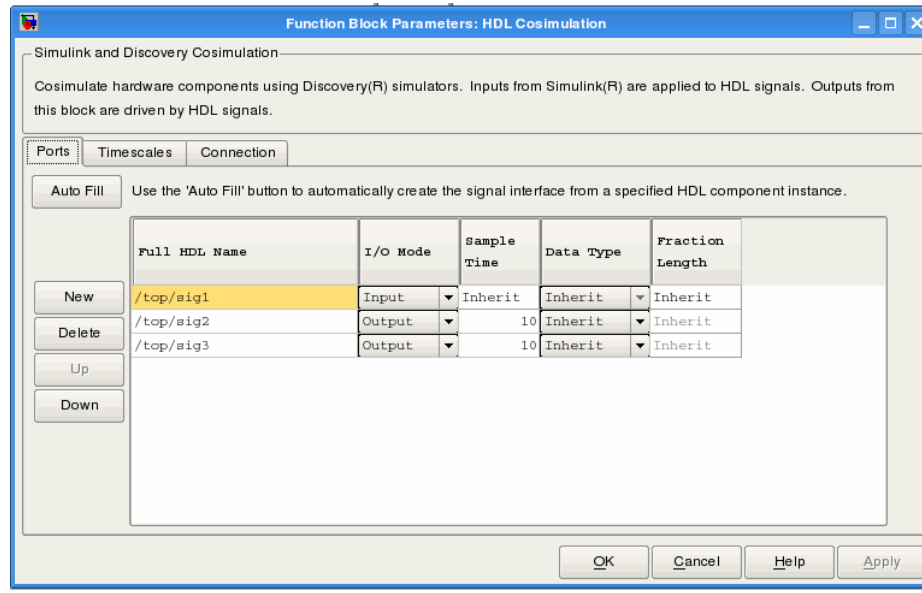
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. An explicit sample time must be specified for each output port.

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`.

(Note, however, that can not force width; the width is always inherited from the HDL simulator.)

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

- VHDL signals of any logic type, such as`STD_LOGIC` or `STD_LOGIC_VECTOR`
- Verilog signals of `wire` or `reg` type



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 3-29 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**.

Note When importing VHDL signals, signal names are returned in all capitals.

To edit the 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 3-26.

I/O Mode

Select either `Input` or `Output`.

`Input` designates signals of your HDL module that are to be driven by Simulink. 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 are to be read by Simulink. For output signals, you must specify an explicit sample time. You can also specify a data type (except width), if desired (see Date Type and Fraction Length in a following section).

Since Simulink signals do not have the semantic of tri-states (there is no 'Z' value), it is not meaningful to connect to a bi-directional HDL signal directly. In order to interface with bi-directional signals, you can interface to the input of the output driver, the enable of the output driver, and the output of the input driver. This leaves the actual tri-state buffer in HDL where resolution functions can handle interfacing with other tri-state buffers.

Sample Time

This property is enabled only for output signals. 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 3-12.

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 is enabled when the **Data Type** property is not set to `Inherit`.

Output port data types are determined by 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.

  `Inherit` is the default setting. When `Inherit` is selected, the **Fraction Length** edit field is disabled.

  Simulink always double checks that the word-length back propagated by Simulink matches the word length queried from the HDL simulator. If they don't match an error is generated. 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 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 `Signed` is selected, 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**.

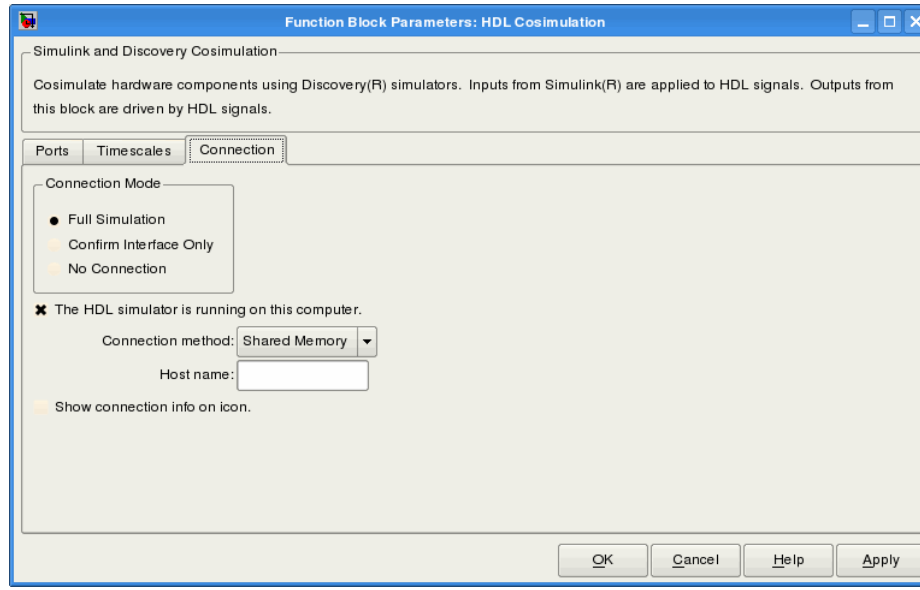
  For example, if you specify **Data Type** as `Signed` and a **Fraction Length** of `5` for a 16-bit signal, Simulink forces the data type to `sfix16_En5`. For the same signal with a **Data Type** set to `Signed` and **Fraction Length** of `-5`, Simulink forces the data type to `sfix16_E5`.

- Select `Unsigned` from the **Data Type** list if you want to explicitly assign an unsigned fixed point data type When `Unsigned` is selected, the **Fraction Length** edit field is enabled. The port is assigned a fixed point type `ufixN_EnF`, where `N` is the signal width and `F` is the **Fraction Length**.

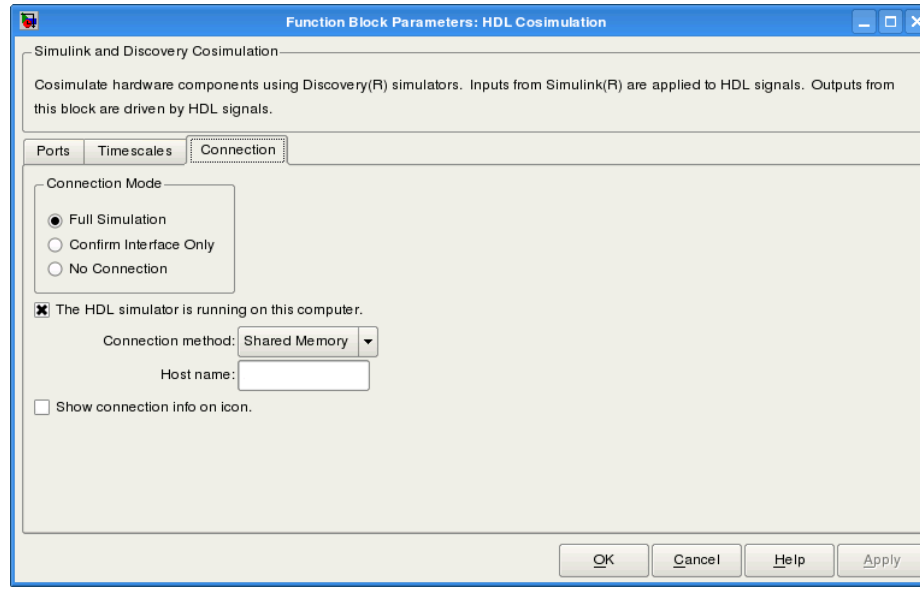
  For example, if you specify **Data Type** as `Unsigned` and a **Fraction Length** of `5` for a 16-bit signal, Simulink forces the data type to `ufix16_En5`. For the same signal with a **Data Type** set to `Unsigned` and **Fraction Length** of `-5` , Simulink forces the data type to `ufix16_E5`.

Connection Pane

This figure shows the default configuration of the **Connection** pane. By default, the block is configured for shared memory communication between Simulink and the HDL simulator, running 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 running a Simulink simulation, use these options to specify what type of simulation connection you want. Select one of the following:

- **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 2nd and 3rd 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 this option is deselected, only TCP/IP socket mode is available, and the **Connection method** list is disabled.

Connection method

This list is enabled when **The HDL Simulator is running on this computer** is selected. 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 1-8.

Host name

If Simulink and the HDL simulator are running on different computers, this text field is enabled. 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 B-2.

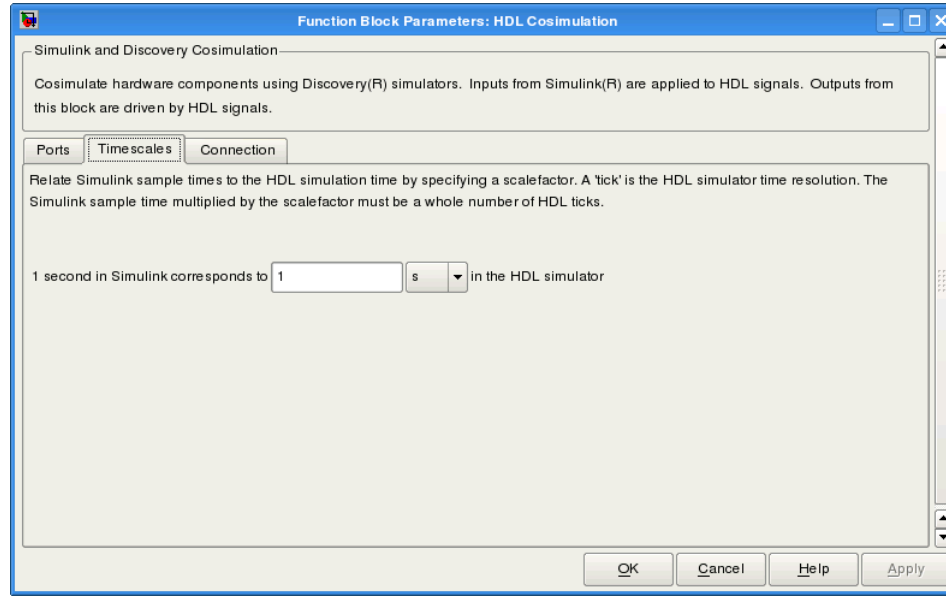
Show connection info on icon

When this option is selected, Simulink indicates information about the selected communication method and (if applicable) communication options information on the HDL Cosimulation block icon. If shared memory is selected, the icon displays the string `SharedMem`. If TCP socket communication is selected, the icon 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

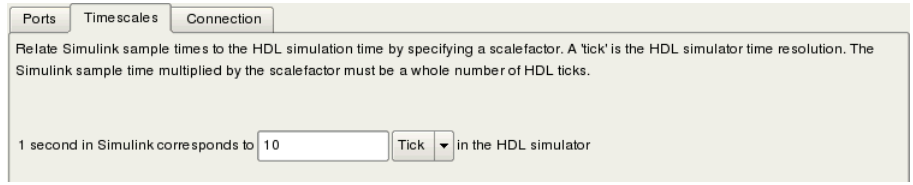
The **Timescales** pane of the HDL Cosimulation block parameters dialog lets you choose a timing relationship between Simulink and the HDL simulator. The following figure shows the default settings of the **Timescales** pane.



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:

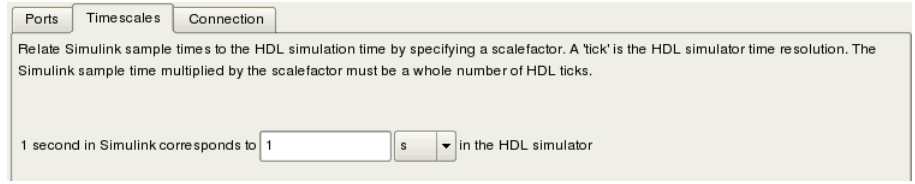
- In *relative* terms (i.e., as some number of HDL simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. Relative timing mode is the default.

To use relative mode, select `Tick` from the list on the right, and enter the desired number of ticks in the edit box. For example, in the figure below the **Timescales** pane is configured for a relative timing correspondence of `10` HDL simulator ticks to `1` Simulink second.



- In *absolute* units (such as milliseconds or nanoseconds). In this case, the cosimulation is said to operate in *absolute timing mode*.

  To use absolute mode, select a unit of absolute time (available units are `fs`, `ps`, `ns`, `us`, `ms`, `s`) from the list on the right. Then enter a scale factor in the left-side edit box. For example, in the figure below the **Timescales** pane is configured for an absolute timing correspondence of `1` HDL simulator second to `1` Simulink second.



  For more information on calculating relative and absolute timing modes, see “Defining the Simulink and HDL Simulator Timing Relationship” on page 3-13.

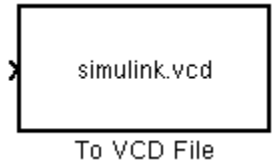
For detailed information on the relationship between Simulink and the HDL simulator during cosimulation, and on the operation of relative and absolute timing modes, see “Representation of Simulation Time” on page 3-12.

Purpose

Generate value change dump (VCD) file

Library

EDA Simulator Link DS



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. 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

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:

- 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

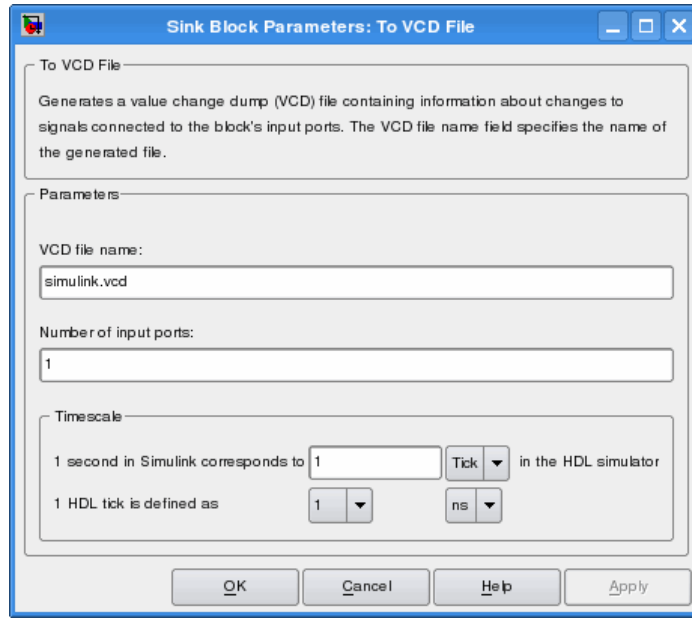
VCD files can grow very large for larger designs or smaller designs with longer simulation runs. However, the size of a VCD file generated

by the To VCD File block is limited only by 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 6-18.

Note The toVCD 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 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 occurs when the input port receives a multi-dimensional signal.

Because multi-dimensional signals are not part of the VCD specification, they are flattened 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. 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*. Relative timing mode is the 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 is said to operate 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.

Generated VCD File Format (Continued)

File Content	Description
`$scope module manchestermodel $end`	The scope of the module being dumped.

`$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.

Generated VCD File Format (Continued)

File Content	Description

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

`#630` `1!`	The starting point of logged value changes. Variable values are checked at each simulation time increment and are logged if a change occurs. This entry indicates that at 63 nanoseconds, the value of signal `Original Data` changed from 0 to 1.

Generated VCD File Format (Continued)

File Content	Description
. . . #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

A

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 theDiscovery™ 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 Discovery™ 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.

B

TCP/IP Socket Communication

Choosing TCP/IP Socket Ports (p. B-2)	Contains instructions for selecting TCP/IP socket ports
Specifying TCP/IP Values (p. B-4)	Provides some examples of valid TCP/IP values that can be used for TCP/IP socket communication.
TCP/IP Services (p. B-5)	Explains how using TCP/IP services may help optimize your application

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
  ```

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 theSimulink® 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.

C

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

Index

A

B

C

D

E

I

K

L

M

P

R

S

T

U

V

W

Z