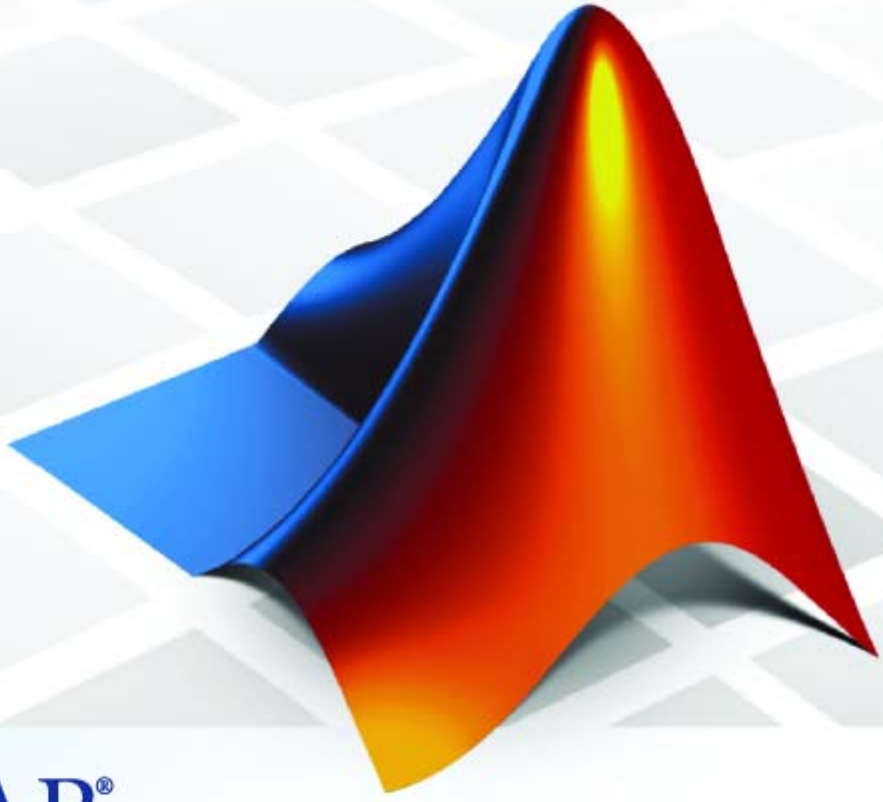
Link for Cadence Incisive 2
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Link for Cadence Incisive User's Guide

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Cadence, Incisive, and the Cadence logo are either trademarks or registered trademarks of Cadence Design Systems, Inc. in the United States and/or other jurisdictions. Cadence's trademarks are used by The MathWorks, Inc. under license.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

Revision History

September 2006	Online only	New for Version 1 (Release 2006b+)
March 2007	Online only	Updated for Version 2.0 (Release 2007a)
September 2007	Online only	Updated for Version 2.1 (Release 2007b)

Contents

1 Getting Started

2 Linking MATLAB to Cadence Incisive

3 Linking Simulink to Cadence Incisive

C Link for Cadence Incisive Machine Configuration Requirements

D TCP/IP Socket Communication

Index

1

Getting Started

What Link for Cadence Incisive? (p. 1-2)	Identifies typical applications and expected users, lists key product features, describes the Link for Cadence Incisive cosimulation environment, and provides an overview of how you work with the integrated tool environment.
Prerequisites for Using Link for Cadence Incisive (p. 1-8)	Describes what you need to know and what other products are required to use Link for Cadence Incisive
Setting Up Your Environment for Link for Cadence Incisive (p. 1-11)	Explains how to install and set up Link for Cadence Incisive software.
Starting the HDL simulator for Use with Link for Cadence Incisive (p. 1-21)	Explains and demonstrates how to invoke the Incisive or NC simulator so that it will work with Link for Cadence Incisive
Workflow for Using Link for Cadence Incisive with MATLAB (p. 1-23)	Describes very basic steps for creating MATLAB — Incisive or NC simulator applications
Workflow for Using Link for Cadence Incisive with Simulink (p. 1-24)	Describes very basic steps for creating Simulink — Incisive or NC simulator cosimulations
Learning More About Link for Cadence Incisive (p. 1-25)	Identifies and explains how to gain access to available documentation online help, demo, and tutorial resources.

What Link for Cadence Incisive?

In this section...

Overview

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

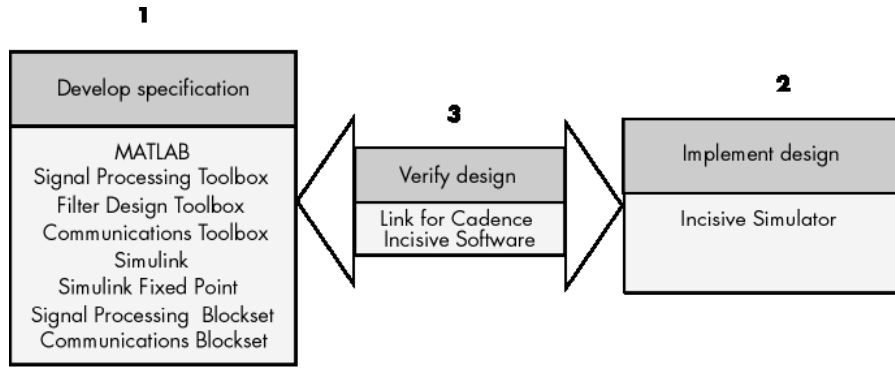
- Incisive simulator — Hardware modeling in HDL and simulation
- MATLAB — Numerical computing, algorithm development, and visualization
- Simulink — Simulation of system-level designs and complex models

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

Link for Cadence Incisive 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, Link for Cadence Incisive software connects tools that are traditionally used discretely to accomplish specific steps in the design process. By connecting the tools, Link for Cadence Incisive software 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.

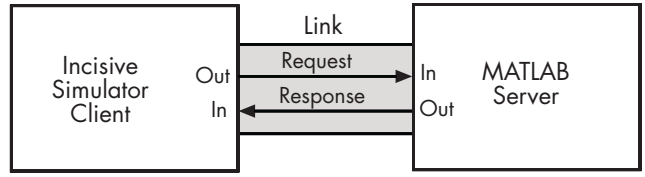
In addition to the preceding design scenario, Link for Cadence Incisive software enables you to use

- 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 model using SystemVerilog and/or SystemC with MATLAB or Simulink using Link for Cadence Incisive. Write simple wrappers around the SystemC and make sure that the SystemVerilog cosimulation connections are to ports or signals of data types supported by Link for Cadence Incisive.

How Link for Cadence Incisive Works with MATLAB and the Incisive or NC simulator

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

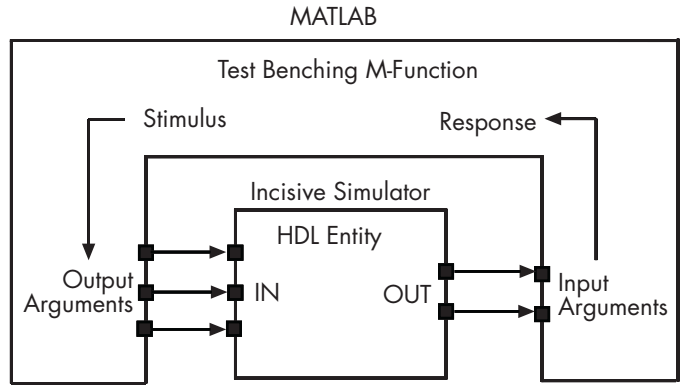


In this scenario, a MATLAB server function waits for service requests that it receives from an Incisive simulation session. After receiving a request, the server establishes a communication link and invokes a specified MATLAB function wrapper that computes data for, verifies, or visualizes the HDL model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

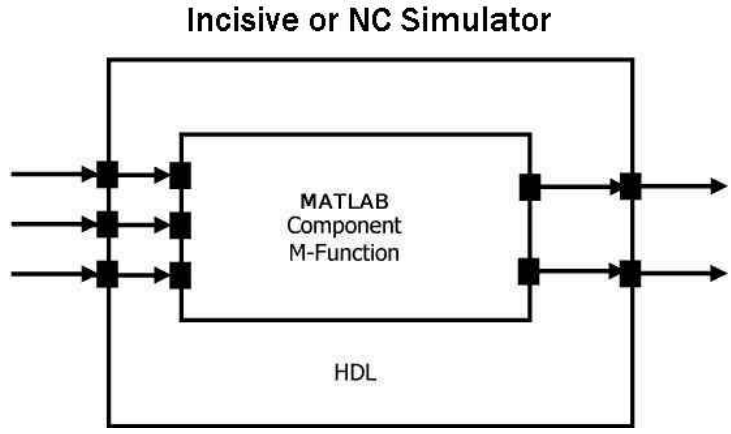
After the server is running, you can start and configure the HDL simulator for use with MATLAB with the supplied Link for Cadence Incisive function nclaunch. Optional parameters allow you to specify

- Tcl commands that execute as part of startup
- A specific Incisive or NC simulator executable to start
- The name of an Incisive or NC simulator 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, Link for Cadence Incisive equips the HDL simulator with a set of Link for Cadence Incisive command extensions you use to

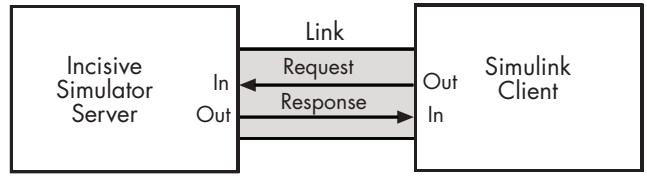
- Load the HDL simulator, `ncsim`, with an instance of a HDL entity to be tested with MATLAB
- Initiate a MATLAB test bench or component session for that instance

When you initiate a specific test bench session, you specify parameters that identify

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

How Link for Cadence Incisive Works with Simulink and the Incisive or NC 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 initiate 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 Incisive simulator wave window to monitor simulation timing diagrams.

Using the Block Parameters dialog box for a Link for Cadence Incisive HDL Cosimulation block, you can configure

- Block input and output ports that correspond to signals (including internal signals) of an HDL model. You can specify sample times and fixed-point data types for individual block output ports if desired.
- Type of communication and communication settings used for exchanging data between the simulation tools.

- Rising-edge or falling-edge clocks to apply to your model. The period of each clock is individually specifiable.
- Tcl commands to run before and after the simulation.

Link for Cadence Incisive software equips the HDL simulator with a set of Link for Cadence Incisive command extensions. Using the supplied command `hdlsimulink`, you execute the HDL simulator with an instance of an HDL model for cosimulation with Simulink. After the model is loaded, you can start the cosimulation session from Simulink.

Link for Cadence Incisive software also includes a block for generating value change dump (VCD) files. You can use VCD files generated with this block

- To view Simulink simulation waveforms in your HDL simulation environment
- To compare results of multiple simulation runs, using the same or different simulation environments
- As input to post-simulation analysis tools

Modes of Communication

The mode of communication that Link for Cadence Incisive uses for a link between the HDL simulator and MATLAB or Simulink somewhat depends on whether your simulation application runs in a local, single-system configuration or in a network configuration. If the HDL simulator and 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 Appendix D, “TCP/IP Socket Communication”.

Prerequisites for Using Link for Cadence Incisive

In this section...

What You Need to Know

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

- Hardware design and system integration
- VHDL and/or Verilog
- Incisive simulators from Cadence Design Systems, Inc.
- MATLAB

Experience with Simulink and Simulink Fixed Point 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

Link for Cadence Incisive requires the following:

Platform	For the specific platforms supported with the current release of the Link for Cadence Incisive software, visit The MathWorks Link for Cadence Incisive product requirements page.
Application software	Incisive HDL Simulator, Incisive Design Team Simulator, or Incisive Enterprise Specman Simulator. Visit The MathWorks Link for Cadence Incisive product requirements page for specific versions supported with the current release of the Link for Cadence Incisive software. MATLAB
Additional application software required for cosimulation with Simulink	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 Link for Cadence Incisive demos require one or more of the above.

Platform-specific software

The Link for Cadence Incisive shared libraries (`liblfihdls*.so`, `liblfihdlc*.so`) are built using the `gcc` included in the Incisive simulator platform distribution. If you are linking your own applications into the HDL simulator, the recommendation is that you also build against this `gcc`. See the HDL simulator documentation for more details about how to build and link your own applications.

Setting Up Your Environment for Link for Cadence Incisive

In this section...

Installing the Link for Cadence Incisive Software

For details on how to install Link for Cadence Incisive software, see the MATLAB installation instructions.

Installing Related Application Software

Based on your configuration decisions and the software required for your Link for Cadence Incisive 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 for Cadence Incisive MATLAB server, 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 an Incisive or NC simulator, see the installation instructions for that product. For information on installing MathWorks products, see the MATLAB installation instructions.

Setting Up Cadence Incisive Software for Use with Link for Cadence Incisive

Link for Cadence Incisive provides a guided set-up script (`syschecklfi`) for configuring your simulator. The script works whether you have installed Link for Cadence Incisive and MATLAB on the same machine as the HDL simulator or installed them on different machines.

The set-up script creates a configuration file containing the location of the appropriate Link for Cadence Incisive MATLAB and Simulink libraries. You

can then include this configuration with any other calls you make using the Cadence Incisive `ncsim` function from the HDL simulator. You only need to run this script once.

Note The Link for Cadence Incisive configuration and diagnostic script works only on UNIX and Linux. Windows users: please see instructions below.

If you plan to use the MATLAB `nclaunch.m` function instead, no setup is required. Refer to "Using the Link for Cadence Incisive Libraries" on page 1-16 for the correct Link for Cadence Incisive library for your platform. Then see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21.

After you've created your configuration files, see "Starting the Incisive or NC simulator from a Shell" on page 1-22.

Using the Configuration and Diagnostic Script for UNIX/Linux

`syschecklfi` provides an easy way to configure your simulator setup to work with Link for Cadence Incisive.

The following is an example of running `syschecklfi` on a Linux 64 machine with Link for Cadence Incisive libraries in a different location than where they were initially installed and specifying a TCP/IP connection.

Start `syschecklfi`:

```
% syschecklfi
********************************************************************************

Kernel name: Linux
Kernel release: 2.6.11.4-20a-smp
Machine: x86_64
********************************************************************************
```

The script first returns the location of the HDL simulator installation (`ncsim.exe`). If it does not find an installation, you receive an error message. Either provide the path to the installation or quit the script and install the

HDL simulator. You are then prompted to accept this installation or provide a path to another one, after which you receive a message confirming the HDL simulator installation:

```
Found /hub/share/apps/HDLTools/IUS/incisive-lfi-1/glnx/tools/bin/64bit/ncsim on the path.
Press Enter to use the path we found or enter another one:

********************************************************************************

/hub/share/apps/HDLTools/IUS/incisive-lfi-1/glnx/tools/bin/64bit/ncsim -version
TOOL: ncsim(64) 05.70-s008
Cadence Incisive mode: 64 bits
********************************************************************************
```

Next, the script needs to know if the Link for Cadence Incisive libraries are in the default directory (where they were initially installed) or if you have moved them to another directory. If you have the Incisive or NC simulator and MATLAB on separate machines, move the Link for Cadence Incisive libraries to the machine with the HDL simulator.

```
Select method to search for Link for Cadence Incisive libraries:
1. Use libraries installed with Link for Cadence Incisive.
2. Prompt me to specify the direct path to the libraries.
2
Enter the path to liblfihdlc_gcc32.so and liblfihdls_gcc32.so:
/tmp/lficonfig/linux64
Found /tmp/lficonfig/linux64/liblfihdlc_gcc32.so
and /tmp/lficonfig/linux64/liblfihdls_gcc32.so.
```

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

```
********************************************************************************

Running dependency checker "ldd /tmp/lficonfig/linux64/liblfihdlc_gcc32.so".
Dependency checker passed.
Dependency status:
     librt.so.1 => /lib64/tls/librt.so.1 (0x00002aaaaac1c000)
     libpthread.so.0 => /lib64/tls/libpthread.so.0 (0x00002aaaaad24000)
     libstdc++.so.5 => /usr/lib64/libstdc++.so.5 (0x00002aaaaae39000)
```

```
        libm.so.6 => /lib64/tls/libm.so.6 (0x00002aaaab016000)
        libgcc_s.so.1 => /devel/Ahdl/nightly/matlab/sys/os/glnxa64/libgcc_s.so.1 (0x00002aaaab16d000)
        libc.so.6 => /lib64/tls/libc.so.6 (0x00002aaaab27a000)
        /lib64/ld-linux-x86-64.so.2 (0x0000555555555000)
*******************************************************************************
```

This next step loads the Link for Cadence Incisive libraries and compiles a test module to verify the libraries loaded correctly.

```
Press Enter to load Link for Cadence Incisive or enter 'n' to skip this test:

ncvlog(64): 05.70-s008: (c) Copyright 1995-2006 Cadence Design Systems, Inc.
ncelab(64): 05.70-s008: (c) Copyright 1995-2006 Cadence Design Systems, Inc.

*******************************************************************************

ncsim(64): 05.70-s008: (c) Copyright 1995-2006 Cadence Design Systems, Inc.
ncsim> source ../hub/share/apps/HDLTools/IUS/incisive-lfi-1/glnx/tools/inca/files/ncsimrc
ncsim> call nomatlabtb
ncsim> exit

*******************************************************************************

Link for Cadence Incisive libraries loaded successfully.
*******************************************************************************
```

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

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

Enter an available port [5001]

Enter remote host [localhost]

Press Enter to continue

ttcp_glnx -t -p5001 localhost
```

```
Connection successful
```

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

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

********************************************************************************

Created template files simulink17032.arg and matlab17032.arg. Inspect and modify if necessary.
********************************************************************************

Diagnosis Completed
```

The template file names, in this example `simulink17032.arg` and `matlab17032.arg`, have different names each time you run this script.

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

Using the Configuration and Diagnostic Script with Windows

`syschecklfi` does not run on Windows. To use the configuration script on Windows, create two files according to the following instructions:

1. Create a MATLAB configuration file and name it. There are no file-naming restrictions. Enter the following text:

```
//Command file for MATLAB Link for Cadence Incisive.
//Loading of foreign Library, usage example: ncsim -f matlab17032.arg entity.
//You can manually change the following line to point to the correct library.
//The default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/incisive/Windows32/liblfihdlc_vs05.dll.

-loadcfc /path/liblfihdlc_vs05.dll:matlabclient
//TCL wrappers for MATLAB commands
-input @proc" "nomatlabtb" "{args}" "{call" "nomatlabtb" "\$args}
-input @proc" "matlabtb" "{args}" "{call" "matlabtb" "\$args}
-input @proc" "matlabcp" "{args}" "{call" "matlabcp" "\$args}
-input @proc" "matlabtbeval" "{args}" "{call" "matlabtbeval" "\$args}
```

where *path* is the path to the particular Link for Cadence Incisive shared library you want to invoke (in this example, `liblfihdlc_vs05.dll`. See "Using the Link for Cadence Incisive Libraries" on page 1-16).

The comments in the above text are optional.

2 Create a Simulink configuration file and name it. There are no file-naming restrictions. Enter the following text:

```
//Command file for Simulink Link for Cadence Incisive.
//Loading of foreign Library, usage example: ncsim -f simulink17032.arg entity.
//You can manually change the following line to point to the correct library.
//For example the default location of the 32-bit Windows library is at
//MATLABROOT/toolbox/incisive/linux32/liblfihdls_vs05.dll.

//For socket connection uncomment and modify the following line:
+socket=5001 -loadvpi /path/liblfihdls_vs05.dll:simlinkserver

//For shared connection uncomment and modify the following line:
//-loadvpi /path/liblfihdls_vs05.dll:simlinkserver
```

Where *path* is the path to the particular Link for Cadence Incisive shared library you want to invoke (in this example, `liblfihdls_vs05.dll`. See "Using the Link for Cadence Incisive Libraries" on page 1-16.

Note If you are going to use a TCP/IP socket connection, first confirm that you have an available port to put in this configuration file. Then, comment out whichever type of communication you will not be using.

The comments in the above text are optional.

After you've finished creating the configuration files, you can leave the files where they are or move them to another location that is convenient.

Using the Link for Cadence Incisive Libraries

In general, you want to use the same compiler for all libraries linked into the same executable. Link for Cadence Incisive provides many versions

of the same library by using the same compilers that MATLAB is compiled with (they vary by platform) as well as the 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 Link for Cadence Incisive library that matches the compiler used for that code:

- Link other 3rd 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 our library. A default library version is understood by the `ncsim` MATLAB command.

Note Link for Cadence Incisive supports running in 32-bit mode on a 64-bit Solaris machine, but it does not support running on a 32-bit Solaris platform.

Library Names

The Link for Cadence Incisive libraries are named according to the following format:

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

where

`productdir`	incisive
`arch`	linux32, linux64, solaris32, solaris64, or windows32
`product_short_name`	lfi
`client_server_tag`	c or s (MATLAB or Simulink)

`compiler_tag`	gcc32, gcc33, gcc40, gcc41, spro11, or vs05
`libext`	dll or so

Not all combinations are supported. See "Default Libraries" on page 1-18 for valid combinations.

Default Libraries

Link for Cadence Incisive scripts fully support the use of designated default libraries. The default libraries are different for each of the products, mostly due to backward-compatibility issues.

With Link for Cadence Incisive, 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 Link for Cadence Incisive. The default libraries for each platform are in bold text.

Platform	MATLAB Library	Simulink Library

Linux32, Linux64	**`liblfihdlc_gcc41.so`**
	`liblfihdlc_gcc32.so`
	`liblfihdlc_gcc40.so`
		`liblfihdls_gcc41.so`
		`liblfihdls_gcc32.so`
		`liblfihdls_gcc40.so`

Solaris32, Solaris64	**`liblfihdlc_spro11.so`** `liblfihdlc_gcc33.so`	**`liblfihdls_spro11.so`** `liblfihdls_gcc33.so`

Windows32	**`liblfihdlc_vs05.dll`**
	`liblfihdlc_gcc32.dll`
	`liblfihdlc_gcc33.dll`
		`liblfihdls_vs05.dll`
		`liblfihdls_gcc32.dll`
		`liblfihdls_gcc33.dll`

Using an Alternative Library

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

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

In MATLAB:

```
>> currPath = getenv('PATH');
>> currLdPath = getenv('LD_LIBRARY_PATH');
>> setenv('PATH', ['/tools/IUS-583p2/bin:' currPath]);
>> nclaunch('tclstart', { 'exec ncvhdl inverter.vhd', ...
                          'exec ncelab -access +rwc inverter', ...
                          'hdlsimulink -gui inverter' }, ...
            'libfile', 'liblfihdls_spro11');
```

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

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

```
ncsim> exec ldd /path/to/liblfihdls_spro11.so
       libxnet.so.1 =>  /lib/libxnet.so.1
       librt.so.1 =>    /lib/librt.so.1
       libm.so.2 =>     /lib/libm.so.2
       libc.so.1 =>     /lib/libc.so.1
       libstlport.so.1 => /tools/SUNWspro_studio11_20070319/opt/SUNWspro/lib/stlport4/libstlport.so.1
       libCrun.so.1 =>  /usr/lib/libCrun.so.1
       libaio.so.1 =>   /lib/libaio.so.1
       libmd5.so.1 =>   /lib/libmd5.so.1
       libm.so.1 =>     /lib/libm.so.1
       libsocket.so.1 =>        /lib/libsocket.so.1
       libnsl.so.1 =>   /lib/libnsl.so.1
       libmp.so.2 =>    /lib/libmp.so.2
       libscf.so.1 =>   /lib/libscf.so.1
```

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

Example: Link for Cadence Incisive Alternate Library Using System Shell. This example shows how to load an Incisive simulator session by explicitly specifying the Link for Cadence Incisive library (default or not). By explicitly using a system shell, you can execute this example on the same machine as MATLAB, on a different machine, and even on a machine with a different operating system.

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

In a csh-compatible system shell:

```
csh> setenv PATH /tools/ius-583p2/lnx/tools/bin/64bit:${PATH}
csh> setenv LD_LIBRARY_PATH /tools/ius-583p2/lnx/tools/systemc/gcc/3.4.6-x86_64
    /install/lib64:${LD_LIBRARY_PATH}
csh> ncvhdl inverter.vhd
csh> ncelab -access +rwc inverter
csh> ncsim -tcl -loadvpi /tools/matlab-7b/toolbox/incisive/linux64
    /liblfihdlc_gcc34:matlabclient inverter.vhd
```

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

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

Starting the HDL simulator for Use with Link for Cadence Incisive

In this section...

Starting the Incisive or NC simulator from MATLAB

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

To start the Incisive or NC simulator from MATLAB, enter `nclaunch` at the MATLAB command prompt:

```
>> nclaunch
```

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

See "nclaunch Examples" on page 1-21 for examples of using `nclaunch` with various property/name value pairs and other parameters.

When you specify a communication mode using `nclaunch`, the `nclaunch` function applies the specified communication mode to all invocations of MATLAB or Simulink from the Incisive or NC simulator.

nclaunch Examples

The following example changes the directory location to `VHDLproj` and then calls the function `nclaunch`. Because the command line omits the

'hdlsimdir' and 'startupfile' properties, nclaunch creates a temporary file. The 'tclstart' property specifies Tcl commands that load and initialize the HDL simulator for test bench instance modsimrand.

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

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

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

Starting the Incisive or NC simulator from a Shell

To start the HDL simulator from a shell and include the Link for Cadence Incisive libraries, you need to first run the configuration script. See “Using the Configuration and Diagnostic Script for UNIX/Linux” on page 1-12.

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

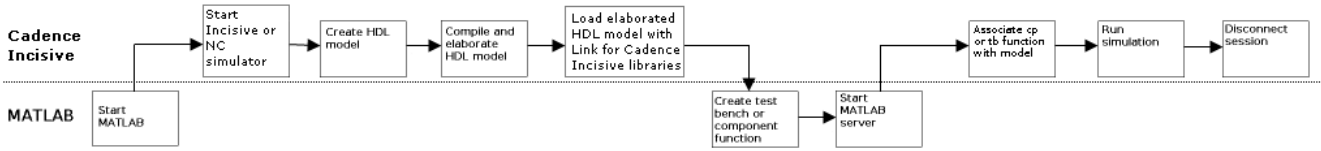
```
% ncsim -f matlabconfigfile entity
```

matlabconfigfile should be the name of the MATLAB configuration file you created either with the guided script (Linux/UNIX) or by creating the file yourself (Windows). If you are connecting to Simulink, this should be the name of the Simulink configuration file. Either way, you must also specify the path to the configuration file if it does not reside in the same directory as ncsim.exe.

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

Workflow for Using Link for Cadence Incisive with MATLAB

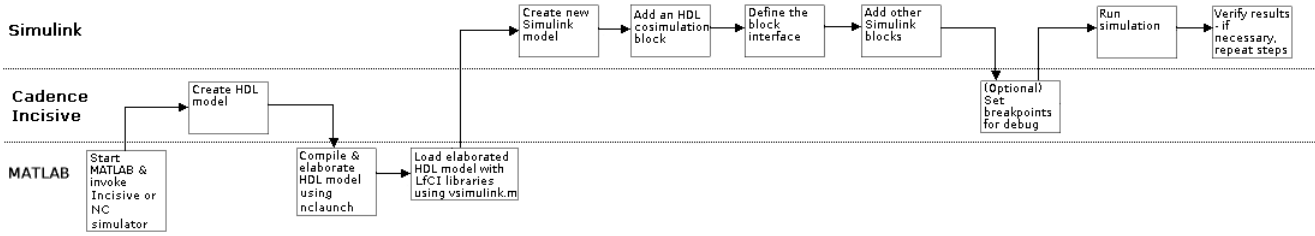
The following diagram illustrates the steps necessary to create and run a MATLAB test bench or component session.



This diagram assumes you are using the configuration and diagnostic script (either UNIX/Linux or Windows) and starting the Incisive or NC simulator outside of MATLAB.

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

Workflow for Using Link for Cadence Incisive with Simulink



This diagram assumes you are using the configuration and diagnostic script (either UNIX/Linux or Windows) and starting the Incisive or NC simulator outside of MATLAB.

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

Learning More About Link for Cadence Incisive

In this section...

Documentation Overview

The following documentation is available with this product.

Title	Description
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.
Chapter 2, “Linking MATLAB to Cadence Incisive”	Explains how to code HDL models and MATLAB functions for Link for Cadence Incisive MATLAB applications. Provides details on how the Link for Cadence Incisive interface maps HDL data types to MATLAB data types and vice versa. Explains how to start and control Incisive or NC simulator and MATLAB test bench sessions.
Chapter 3, “Linking Simulink to Cadence Incisive”	Explains how to use the Incisive or NC simulator and Simulink for cosimulation modeling.
MATLAB Functions — Alphabetical List	Describes Link for Cadence Incisive functions for use with MATLAB.

Title	Description
HDL Simulator Tcl Commands — Alphabetical List	Describes Link for Cadence Incisive Tcl commands for use with the HDL simulator.
Simulink Blocks — Alphabetical List	Describes Link for Cadence Incisive blocks for use with Simulink.
Appendix B, "VHDL and Verilog Language Support"	Provides and overview of how these languages are supported by Link for Cadence Incisive
Appendix C, "Link for Cadence Incisive Machine Configuration Requirements"	Explains the machine configurations permissible when linking the Incisive or NC simulator to MATLAB or Simulink
Appendix D, "TCP/IP Socket Communication"	Provides information for choosing TCP/IP socket ports.

Online Help

The following online help is available:

- Online help in the MATLAB Help browser. Click the Link for Cadence Incisive product link in the browser's Contents.
- M-help for Link for Cadence Incisive MATLAB functions and HDL simulator commands. This help is accessible with the MATLAB `doc` and `help` commands. For example, enter the command

  ```
  doc incisive
  ```

  or

  ```
  help incisive
  ```

  at the MATLAB command prompt.
- Block reference pages accessible through the Simulink interface.

Demos and Tutorials

The Link for Cadence Incisive product 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. Tutorials provide procedural instruction on how to apply the product.

To see a list of Link for Cadence Incisive demos and tutorials that you can run, type the following at a MATLAB command prompt:

```
>> demos
```

Select **Links and Targets > Link for Cadence Incisive** from the navigational pane.

2

Linking MATLAB to Cadence Incisive

MATLAB — HDL Simulator Workflow (p. 2-2)	Provides an overview of the steps involved in coding and running MATLAB functions for use with Link for Cadence Incisive.
Coding a Link for Cadence Incisive MATLAB Application (p. 2-4)	Explains how to code HDL modules and MATLAB functions for use with Link for Cadence Incisive. Provides details on how Link for Cadence Incisive maps HDL data types to MATLAB data types and vice versa.
Associating a MATLAB Link Function with an HDL Model (p. 2-34)	Describes scheduling and communications options for a MATLAB link session with the HDL simulator
Running MATLAB Link Sessions (p. 2-45)	Explains how to start and control Cadence Incisive and MATLAB link sessions.

MATLAB — HDL Simulator Workflow

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

In MATLAB...	In Incisive or NC simulator...
1 Start MATLAB and invoke Incisive or NC simulator (see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21)	
	2 Create the HDL model. **3** Compile and elaborate the HDL model. **4** Load elaborated HDL model with Link for Cadence Incisive libraries. See "Loading an HDL Design for Verification" on page 2-11.

In MATLAB...	In Incisive or NC simulator...
5 Create test bench or component function (see "Coding a Link for Cadence Incisive MATLAB Application" on page 2-4). **6** Start the server. See "Starting the MATLAB Server" on page 2-47.	
	7 Use `matlabcp`, `matlabtb`, or `matlabtbeval` to associate the function you wrote in step 5 with a component or entity of the loaded model currently in the Incisive or NC simulator (see "Associating a MATLAB Link Function with an HDL Model" on page 2-34). For additional scheduling and communication options, see "Scheduling Options for a Link Session" on page 2-37. See also the reference pages for `matlabcp`, `matlabtb`, and `matlabtbeval`. **8** Run the simulation. **9** Disconnect the session by using `nomatlabtb`.

Coding a Link for Cadence Incisive MATLAB Application

In this section...

Overview

Link for Cadence Incisive software supports two types of MATLAB functions that interface to HDL models:

- *Test bench functions* are functions that 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* are functions that 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)
  ```

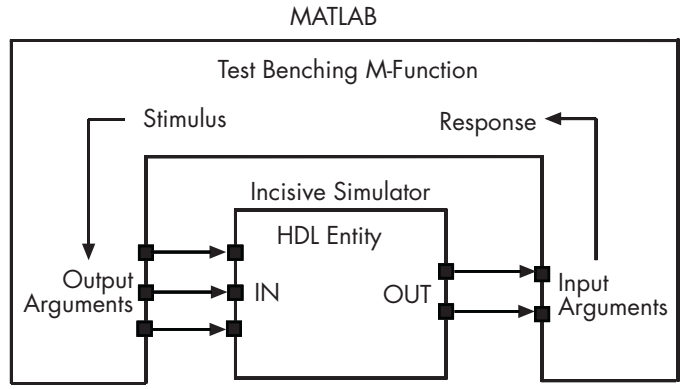
These two types of MATLAB functions are referred to collectively as MATLAB *link functions*, and a test bench or component session may be referred to as a MATLAB *link session*.

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.

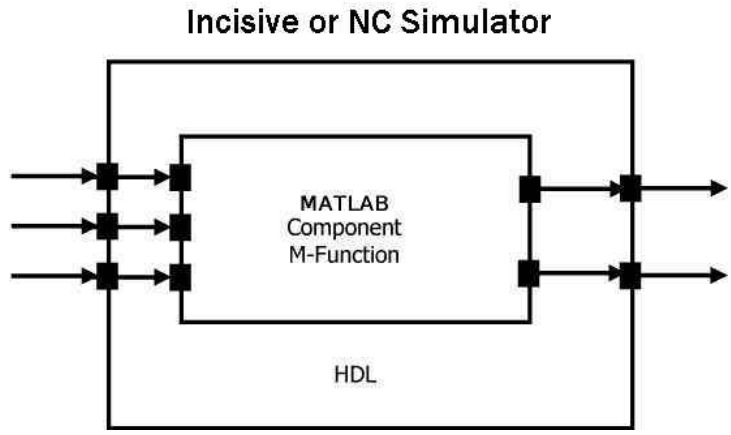
This section provides an overview of the steps required to develop an HDL model for use with MATLAB and Link for Cadence Incisive software. To program the HDL component of a Link for Cadence Incisive application:

1. Code the HDL model for MATLAB verification “Coding HDL Modules for MATLAB Verification” on page 2-7.
2. Compile the HDL model.
3. Code the required MATLAB test bench or MATLAB component functions.
4. Place the MATLAB functions on the MATLAB search path.

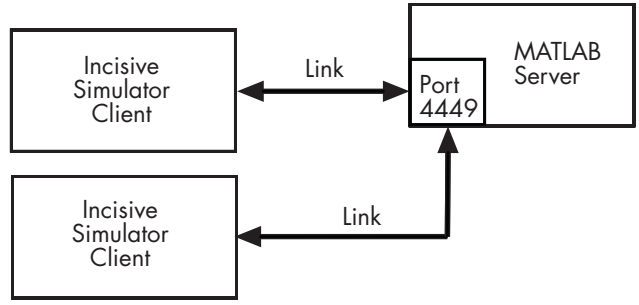
The following figure shows how a MATLAB function wraps around and communicates with the HDL simulator during a test bench simulation session.



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 Incisive or NC simulator sessions and HDL entities. However, you should adhere to recommended guidelines to ensure the server can track the I/O associated with each entity and session. The MATLAB server, which you start with the supplied MATLAB function `hdldaemon`, waits for connection requests from instances of the HDL simulator running on the same or different computers. When the server receives a request, it executes the specified MATLAB function you have coded to perform tasks on behalf of an entity 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 C, “Link for Cadence Incisive Machine Configuration Requirements” for valid machine configurations.

Coding HDL Modules for MATLAB Verification

The most basic element of communication in the Link for Cadence Incisive interface is the HDL module. The interface passes all data between the Incisive or NC simulator and MATLAB as port data. Link for Cadence Incisive works with any existing VHDL entity or Verilog 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 following sections 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, Link for Cadence Incisive assumes that an HDL module and its simulation function share the same name. See “Naming a MATLAB Link Function” on page 2-34.

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

Specifying Port Direction Modes

In your entity or 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 VHDL or Verilog modules. For details on how Link for Cadence Incisive converts data types for the MATLAB environment, see "Performing Data Type Conversions" on page 2-19.

Note If you use unsupported types, Link for Cadence Incisive 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, which you plan to test with MATLAB, with a VHDL data type that is supported by Link for Cadence Incisive. 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 Link for Cadence Incisive does not support VHDL extended identifiers for

- 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, which you plan to test with MATLAB, with a Verilog port data type that is supported by Link for Cadence Incisive. The interface can convert data of the following Verilog port types to comparable MATLAB types:

- `reg`
- `integer`
- `wire`

Note Link for Cadence Incisive 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 exercised by 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 Link for Cadence Incisive converts data of standard logic scalar and array types for use in the MATLAB environment, see "Performing Data Type Conversions" on page 2-19.

```
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 design source files, use the HDL simulator tools to compile and elaborate the code. The Incisive simulator allows for 1-step and 3-step processes for HDL compilation, elaboration, and simulation.

The following Incisive simulator command compiles the VHDL file modsimrand.vhd:

```
sh> ncvhdl modsimrand.vhd
```

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

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

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

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

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

See the Incisive simulator documentation for complete details on compiling and elaborating your HDL designs. For more examples, see Link for Cadence Incisive demos and tutorials.

Loading an HDL Design for Verification

After you start the Incisive or NC simulator from MATLAB with a call to `nclaunch`, load an instance of a VHDL entity or Verilog module for verification with the HDL simulator command `hdlsimmatlab`. At this point, it is assumed that you have coded and compiled your HDL model as explained in "Coding a Link for Cadence Incisive MATLAB Application" on page 2-4. Issue the HDL simulator command `hdlsimmatlab` for each instance of an entity or module in your model that you want to cosimulate. For example:

```
hdlsimmatlab work.modsimrand
```

This command loads the Link for Cadence Incisive library, opens a simulation workspace for `modsimrand`, and displays a series of messages in the HDL simulator command window as the simulator loads the entity.

Coding MATLAB Link Functions

When coding a MATLAB function that is to verify or visualize an HDL model, 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 Incisive or NC simulator. The following sections cover these topics:

Process for Coding MATLAB Link Functions

To code a MATLAB link function that is to verify or visualize an HDL model,

1. Learn the syntax for a MATLAB link function (see "Defining Link Functions and Link Function Parameters" on page 2-13).
2. Understand how Link for Cadence Incisive converts VHDL entity or Verilog module data for use in the MATLAB environment (see "Performing Data Type Conversions" on page 2-19).
3. Consider naming the MATLAB function with the name of the VHDL entity or Verilog module the function is to test (see "Naming a MATLAB Link Function" on page 2-34).
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 2-17).
7. Convert data for manipulation in the MATLAB environment, as necessary (see "Converting Data to Send to MATLAB" on page 2-19).
8. Convert data that needs to be returned to the HDL simulator (see "Converting Data for Return to the HDL Simulator" on page 2-25).

Examples in This Chapter Use the Oscfilter Demo In the Oscillator demo, a VHDL model implements an oscillator (`simple_osc`) whose output is wired to the input of a stub component (`osc_filter`). The sole purpose of `osc_filter` is to invoke a MATLAB component function (`oscfilter`). The `osc_filter` component passes its input signal into the MATLAB function and receives signal data returned by the function. The `oscfilter` function implements a smoothing filter that filters the signal at the model's base rate and at two oversampling (4x and 8x) rates.

You may find it helpful to refer to the demo files while reading this discussion. The directory *`matlabroot`*`\toolbox\incisive\incisivedemos\vhdl\Oscillator` contains the VHDL source code files:

- `simple_osc.vhd`: Contains entity and architecture definitions for oscillator component.
- `osc_filter.vhd`: Contains entity and architecture (stub) definitions for filter component.
- `osc_top.vhd`: Top-level VHDL behavioral model; instantiates and connects oscillator and filter components.

This directory also contains the demo M-files:

- `oscincisive.m`: Top-level demo script; starts up `hdldaemon` and the Incisive or NC simulator, passing in a cell array of Tcl commands to `ncsim`. The Tcl commands direct compilation and simulation of the model
- `oscfilter.m`: MATLAB component function, called from the HDL simulator; performs filter computations.

Defining Link Functions and Link Function Parameters

The syntax of a MATLAB component function is

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

The syntax of a MATLAB test bench function is

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

The 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 output ports of the associated VHDL entity or Verilog module.	*Input* Structure that receives signal values from the input ports defined for the associated VHDL entity or Verilog module at the time specified by `tnow`
`tnext`	*Output, optional* Specifies the time at which the HDL simulator schedules the next callback to MATLAB. `tnext` should be initialized to an empty value ([]). If `tnext` is not subsequently updated, no new entries are added to the simulation schedule. In that case, callback scheduling is controlled by the `matlabtb` or `matlabtbeval` command.	*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 subsequently updated, no new entries are added to the simulation schedule. In that case, callback scheduling is controlled by the `matlabcp` command.
`oport`	*Input* Structure that receives signal values from the input ports defined for the associated VHDL entity or Verilog module at the time specified by `tnow`	*Output* Structure that forces (by deposit) values onto signals connected to output ports of the associated VHDL entity or Verilog 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 2-37.	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 VHDL entity or Verilog 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 2-17.	Same as test bench

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 VHDL entity or Verilog module.

We strongly recommend that you 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 2-37.

Oscfilter Function Example. In the Oscillator demo, the function definition for the `oscfilter` function represents the communication channel between MATLAB and the HDL simulator. The following code is the function definition of the `oscfilter` MATLAB component function.

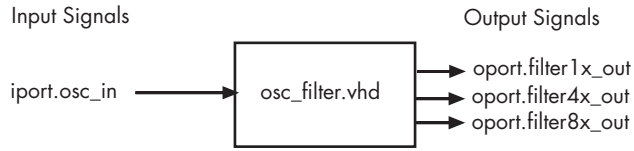
```
function [oport,tnext] = oscfilter(iport, tnow, portinfo)
```

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 VHDL entity 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 VHDL 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 VHDL entity's ports and the MATLAB function's `iport` and `oport` parameters.



Gaining Access to and Applying Port Information. Link for Cadence Incisive 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. You could use the port information to create a generic MATLAB function that operates differently depending on the port information supplied at startup. 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

HDL Port Information (Continued)

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

To use `portinfo` in your MATLAB function to verify port data, perform the following tasks:

1. Check whether `portinfo` data has been passed with a call to the MATLAB function `nargin`. For example:

   ```
   if(nargin == 3),
   ```

2. If data has been passed, you can then verify it. The following code fragment checks whether the resolution limit for time has been set to `1 ns`:

```
.
.
.
    tscale = portinfo.tscale;
    if abs(tscale - 1e-9) > eps,
    error('This test requires a resolution limit of 1 ns');
    end
```

Example

In the Oscillator demo, all I/O ports for the `osc_filter` entity are defined as

```
STD_LOGIC_VECTOR(21 DOWNTO 0)
```

The MATLAB component must convert input signal data from this representation to a 22-bit column or row vector of characters where each bit maps to standard logic character 0 or 1. The inverse conversion is required when outputs are returned to the HDL simulator.

Performing Data Type Conversions

To successfully use Link for Cadence Incisive with Cadence Incisive and MATLAB or Simulink, you need to understand the data type conversions that Link for Cadence Incisive performs in order to transmit and receive data between HDL modules and the MATLAB environment.

Converting Data to Send to MATLAB. If your HDL simulator application needs to send data to a MATLAB function, it may be necessary for you to first convert the data to a type supported by MATLAB and Link for Cadence Incisive. This section explains the following types of conversions:

- VHDL Data Type Conversions
- Verilog Data Type Conversions

VHDL Data Type Conversions

Link for Cadence Incisive converts VHDL entity data to types that apply in the MATLAB environment. To program a MATLAB function for a VHDL model, you must understand the type conversions required by your

application. You may also need to handle differences between the array indexing conventions employed by VHDL and MATLAB.

The data types of arguments passed in to the function determine

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

The following table summarizes how Link for Cadence Incisive 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 Cadence Incisive on page 2-25) 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`.

Array Indexing Differences Between MATLAB and VHDL

In multidimensional arrays, the same underlying OS memory buffer maps to different elements in MATLAB and the VHDL 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.

Verilog Data Type Conversions

Link for Cadence Incisive converts Verilog module data to types that apply in the MATLAB environment. To program a MATLAB function for a Verilog model, you must understand the type conversions required by your application.

The data types of arguments passed in to the function determine

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

The following table summarizes how Link for Cadence Incisive 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).

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);`
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.
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.
Test port values of VHDL type `STD_LOGIC` and `STD_LOGIC_VECTOR`	Use the `all` function as follows: `all(oport.val == '1' | oport.val == '0')` This example returns `True` if all elements are `'1'` or `'0'`.

Example

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);
```

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.

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 Link for Cadence Incisive. The following tables list circumstances under which such conversions are required for VHDL and Verilog.

VHDL Conversions for Cadence Incisive

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`

VHDL Conversions for Cadence Incisive (Continued)

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

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. Keep in mind that MATLAB uses a column-major numbering scheme to represent data elements internally and begins at 1. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on through the last column. VHDL array indexing:
	`PORT (`
	`sta : OUT ARRAY(1 TO 2) OF`
	`BIT_VECTOR(1 TO 8););`
	`.`
	`.`
	`.`
	`sta(2)(7) <= '1'`
	MATLAB equivalent array indexing:
	`iport.sta(7,2) = '1';`

`INTEGER` or `NATURAL`	Declare the data as an array of type `int32` with a size that is equivalent to the VHDL array size. Alternatively, convert the data to an array of type `int32` with the MATLAB `int32` function before returning it. Be sure to limit the data to values with the range of the VHDL type. If necessary, check the `right` and `left` fields of the `portinfo` structure. For example: `iport.int = int32(1:10)';`
`REAL`	Declare the data as an array of type `double` with a size that is equivalent to the VHDL port size. For example: `iport.dbl = ones(2,2);`

VHDL Conversions for Cadence Incisive (Continued)

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

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

Enumerated types	Declare the data as a string for scalar ports or a cell array of strings for array ports with each element equal to a label for the defined enumerated type. The `'label'` field of the `portinfo` structure lists all valid labels (see "Gaining Access to and Applying Port Information" on page 2-17). 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 Cadence Incisive

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 Link for Cadence Incisive software. The example uses a VHDL entity and MATLAB function code drawn from the decoder portion of the Manchester Receiver demo. For the complete VHDL listing, see *matlabroot*`/toolbox/incisive/incisivedemos/manchester/decoder.vhd`.

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 function definition on the first line represents the communication channel between MATLAB and the Incisive simulator. The following code is the function definition of the `manchester_decoder` MATLAB function.

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

The function definition

- Names the function. This definition 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 Tcl command.
- Defines required input and output parameters. A MATLAB test bench function *must* include two input parameters, `iport` and `tnext`, and three output parameters, `oport`, `tnow`, and `portinfo`, and *must* appear in the order shown.

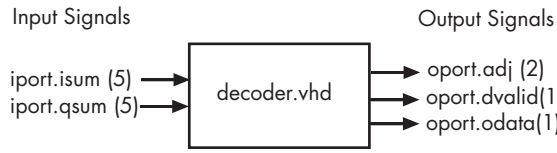
  Note that the function outputs must be initialized to empty values, as in the following code example:

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

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

`iport`	Forces (by deposit) a value onto the signal connected to the entity's input ports, `isum` and `qsum`.
`tnext`	Specifies a time value that indicates when the Incisive simulator is to call back the MATLAB function.
`oport`	Receives VHDL signal values from the entity's output ports, `adj`, `dvalid`, and `odata`.
`tnow`	Receives the simulation time at which the Incisive simulator calls the MATLAB function.
`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 entity's ports and the MATLAB function's `iport` and `oport` parameters.



For more information on the required MATLAB 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 under simulation.

Link for Cadence Incisive 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 Incisive simulator and needs to return to the Incisive 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.

VHDL Example Port Definitions (Continued)

Port	Direction	Type...	Converts to/Requires Conversion to...
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 2-19.

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) = bin2dec(oport.adj');
data(isum) = bin2dec([oport.dvalid oport.odata]);
.
.
.
```

The two calls to `bin2dec` 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 2-19 provides a summary of the types of data conversions to consider when coding simulation MATLAB functions.

5 Convert data to be returned to the Incisive simulator.

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

```
 if isum == 17
   iport.isum = dec2bin(isum,5);
   iport.qsum = dec2bin(qsum,5);
else
   iport.isum = dec2bin(isum,5);
end
```

The three calls to `dec2bin` 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 2-25 provides a summary of the types of data conversions to consider when returning data to the Incisive simulator.

Associating a MATLAB Link Function with an HDL Model

In this section...

Overview

This section describes establishing a relationship between the link function and the HDL model in the HDL 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, Link for Cadence Incisive assumes the name for a MATLAB function matches the name of the VHDL entity or Verilog module that the function verifies or visualizes. For example, if you name the VHDL entity `mystdlogic`, Link for Cadence Incisive 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 filenames in the MATLAB documentation.

Associating the HDL Module Component with the MATLAB Link Function

In the Oscillator demo, the VHDL model instantiates a VHDL entity as the component `u_osc_filter` (see `osc_top.vhd`). After the HDL simulator compiles and loads the VHDL 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 (see `modsimosc.m`).

```
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. `matlabcp` also defines a mapping between the data modeled in the VHDL entity and data in the MATLAB environment. See the `matlabcp` reference documentation for further information.

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

These rules 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

- 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

- 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
```

Specifying TCP/IP Values

When providing TCP/IP information for a MATLAB link function, you can choose 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.

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.

Some valid TCP/IP specification examples follow:

Format	Example
Port number	`4449`
Port alias	`matlabservice`
Port number and remote hostname	`4449@compa`
Remote hostname and port number	`compa:4449`
Port alias and remote host Internet address	`matlabservice@123.34.55.23`

For example,

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

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 2-38)
- 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

- 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 Options for a Link Session" on page 2-37.

Scheduling Link Functions Using Link Function Parameters

By default, Link for Cadence Incisive 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 Link for Cadence Incisive is to invoke the relevant MATLAB function. Depending on your choices, you may need to modify the function or specify specific arguments when you initiate a MATLAB test bench session with the `matlabtb` or `matlabtbeval` function.

After you decide on the controls you need to apply for a link session, you are ready to register callbacks for a specific MATLAB link session with one of the link functions. These functions

- Identify the instance of an entity or module in the HDL model being simulated and test benched

- Define the communication link between the HDL simulator and MATLAB
- Specify a callback to a MATLAB function that executes in the context of MATLAB on behalf of the instance under simulation in the HDL simulator

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 attaches the specified instance to a MATLAB function that has the same name as the instance. For example, if the instance is `modsimrand`, the command links the instance with the MATLAB function `modsimrand` in file `modsimrand.m`. Alternatively, you can specify a different function name with the option `-mfunc`. See "Associating a MATLAB Link Function with an HDL Model" on page 2-34.

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.
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 pathname 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 pathname 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.

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.

Note When specifying signals with the `-rising`, `-falling`, and `-sensitivity` options, specify them in full pathname format. If you do not specify a full pathname, the command applies HDL simulator rules to resolve signal specifications.

Consider the following `matlabtb` command:

```
ncsim> matlabtb modsimrand -rising /modsimrand/clk,
-socket 4449
```

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

Arguments in the command line specify the following:

`modsimrand`	That an instance of the entity `modsimrand` be linked with the MATLAB function `modsimrand`.
`-rising /modsimrand/clk`	That the MATLAB function `modsimrand` be called when the signal `/modsimrand/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 string or 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...
String that includes a unit specification	Parses the string as a scaled time value with units of `fs` (femtoseconds), `ps` (picoseconds), `ns` (nanoseconds), `us` (microseconds), `ms` (milliseconds), or `s` (seconds). The value is scaled to the nearest multiple of the current time value resolution. For example, the following string scales to the simulation time nearest to 12.2 nanoseconds as a multiple of the current HDL simulator time resolution. `tnext = '12.2 ns'`
String that does not specify units	Parses the string as the number of ticks based on the HDL simulator time resolution limit. For example, the following string parses to 100 ticks of the current time resolution. `tnext = '1e2'`

Time Representations for tnext Parameter (Continued)

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`.

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

Link for Cadence Incisive offers flexibility in how you start and control an HDL model test bench or component session with MATLAB. A MATLAB link session is the application of a `matlabtb`, `matlabtbeval`, or `matlabcp` function. A session can consist of a single function invocation, a series of timed invocations, or invocations based on timing data returned by a MATLAB function to the HDL simulator.

This chapter helps you determine what your application's scheduling requirements might be, explains how to start the most basic simulation, and explains how to apply available scheduling mechanisms for finer levels of test bench or component control.

Process for Running MATLAB Link Sessions

To start and control the execution of a simulation in the MATLAB environment,

1. Place MATLAB link function on the MATLAB search path.

2 Check the MATLAB server's link status.

3 Start the MATLAB server.

4 Launch the HDL simulator for use with MATLAB.

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

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

7 Register callbacks for the MATLAB link session.

8 Apply test bench stimuli.

9 Run and monitor the test bench session.

10 Restart simulator during a test bench session.

11 Stop a test bench session.

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

The MATLAB function associated with a VHDL entity or Verilog 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\incisive\MySym\MyVhdlFunction.m
```

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

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 Link for Cadence Incisive is to
   - 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 with the `matlabtb` or `matlabtbeval` 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 "Modes of Communication" on page 1-7. For more information on establishing the HDL simulator end of the communication link, see "Associating a MATLAB Link Function with an HDL Model" on page 2-34.

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

Note You cannot initiate Link for Cadence Incisive 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 Cadence Incisive for Use with MATLAB

Start Cadence Incisive directly from MATLAB by calling the MATLAB function `nclaunch` or see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21 for other options.

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 MATLAB link session environment. One way of applying stimuli is through the `iport` return parameter of the linked MATLAB function. This parameter drives signal values by deposit. Another option is to issue `force` commands in the HDL simulator main window.

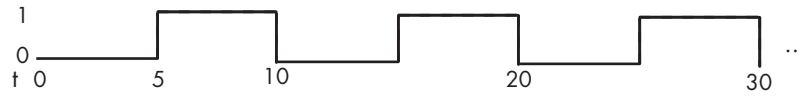
For example, the following sequence of `force` commands:

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

can be entered at the `ncsim` prompt or in the Tcl pane of the HDL Cosimulation block (in the presimulation entry box).

These commands drive

- The `clk` signal to 0 at 0 nanoseconds after the current simulation time and to 1 at 5 nanoseconds after the current HDL simulator 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.



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

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

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

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

All three approaches may lead to performance degradation.

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 or selecting the **Simulate > Run** option in the main window.

   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 a number of time steps. Alternatively, you can specify the `-all` option, which causes the simulation to run forever, until the simulation hits a breakpoint, or a breakpoint event occurs.

   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. The following table lists ways you can set breakpoints in each application environment.

Cadence Incisive Environment	MATLAB Environment
Enter the `bp` command	Click next to an executable statement in the breakpoint alley of the Editor/Debugger
Select **Simulate > Break** in the **Main** window	Click the Set/Clear Breakpoint button on the toolbar
Click the Break button on the **Main** or **wave** window toolbar	Select **Set/Clear Breakpoint** on the **Breakpoints** menu
	Select **Set/Clear Breakpoint** on the context menu
	Call the `dbstop` function

The following HDL simulator command sets a breakpoint at line 50 in the VHDL file `modsimrand.vhd`:

```
bp modsimrand.vhd 50
```

3 Step through the simulation and examine values. The following table lists ways you can step through code in each application environment.

Cadence Incisive Environment	MATLAB Environment
Click the **Step** or **Step Over** button on the **Main** or **wave** window toolbar	Click the Step, Step In, or Step Out toolbar button
Click the **Step** or **Step-Over** options on the **Simulate > Run** menu	Select the **Step**, **Step In**, or **Step Out** option on the **Debug** menu
Enter the `step` command	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. The following table lists ways you can resume a simulation in each application environment.

Cadence Incisive Environment	MATLAB Environment
Click the **Run Continue** button on the **Main** toobar	Click the Continue toolbar button
Select the **Continue** option on the **Simulate > Run** menu	Select the **Continue**, **Run**, or **Save and Run** option on the **Debug** menu
Enter the `run` command with the `-continue` option	Call the `dbcont` function

The following HDL simulator command instructs MATLAB to resume a simulation:

```
run -continue
```

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

Restarting a Link Session

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

1 Make the HDL simulator your active window, if your input focus was not already set to that application.

2 Reload HDL design elements and reset the simulation time to zero by doing one of the following:

- Click the **Restart** button on the **Source Window** toolbar.

- Click the **Restart** option on the **Simulate > Run** menu.
- Enter the `restart` command in the main window.

3. Reissue the `matlabtb` command.

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

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 by selecting the **Simulate > End Simulation** option on the main window.
3. Close your project by selecting the **File > Close > Project** option on the main window.
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 Cadence Incisive online help or documentation.

3

Linking Simulink to Cadence Incisive

Simulink — Cadence Incisive Workflow

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

In MATLAB...	In Incisive or NC Simulator...	In Simulink...
1 Start MATLAB and invoke the HDL simulator (see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21)		
	2 Create the HDL model.	
3 Compile and elaborate the HDL model using `nclaunch`. **4** Load elaborated HDL model with Link for Cadence Incisive libraries. See "Loading an HDL Design for Verification" on page 2-11.		

In MATLAB...	In Incisive or NC Simulator...	In Simulink...
		5 Create a new Simulink model. **6** Add an HDL Cosimulation block (see "Incorporating Hardware Designs Into a Simulink Model" on page 3-30). **7** Define the block interface (see "Defining the Block Interface" on page 3-35). **8** Add other Simulink blocks to complete the Simulink model.

In MATLAB...	In Incisive or NC Simulator...	In Simulink...
	9 (Optional) Set breakpoints for interactive HDL debug.	
		10 Run the simulation. **11** 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. **12** Consider using a To VCD File block to verify cosimulation results.

Introduction to Cosimulation

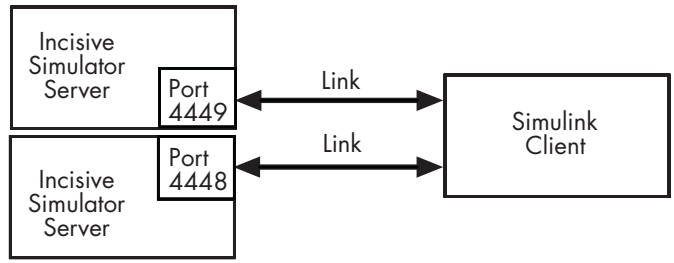
In this section...

Creating a Hardware Model Design for Use in Simulink

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

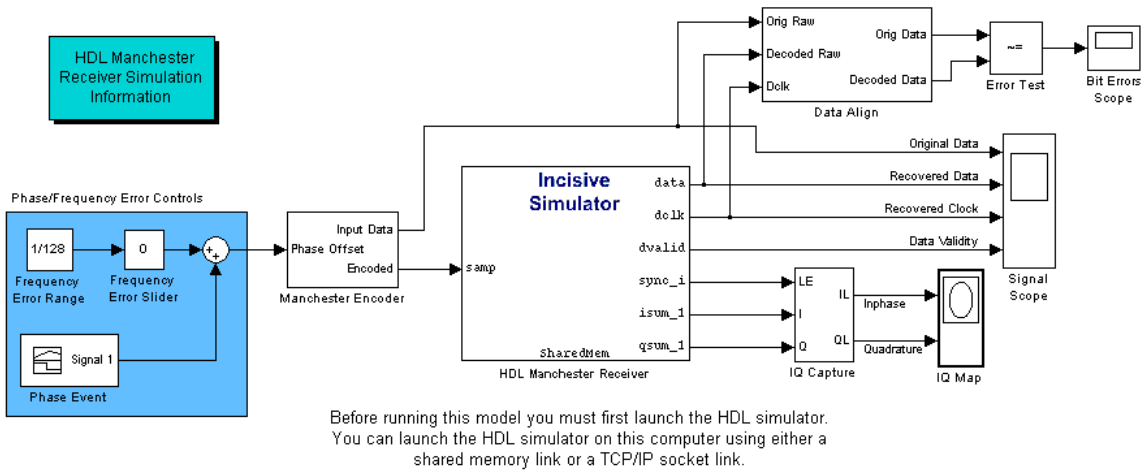
- Will you start by developing an HDL application, using an HDL simulator, and possibly MATLAB, and then test the results at a system level in Simulink?
- Will you start with a system-level model in Simulink with "black box hardware components" and, after the model runs as expected, replace the black 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.
- 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?

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 Link for Cadence Incisive 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. Multiple HDL Cosimulation blocks in a Simulink model can request the service of multiple instances of the HDL simulator, using unique TCP/IP socket ports.

This figure shows a sample Simulink model that includes an HDL Cosimulation block.



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

The HDL Cosimulation block (labeled HDL Manchester Receiver) models a Manchester receiver that is coded in VHDL and Verilog. 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
- Bit Errors block
- Data Scope block
- Discrete-Time Scatter Plot Scope block from the Communications Blockset

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

The Link for Cadence Incisive HDL Cosimulation Block

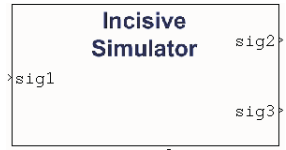
The Link for Cadence Incisive HDL Cosimuation Block links hardware components that are concurrently simulating in the HDL simulator to the rest of a Simulink model.

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.

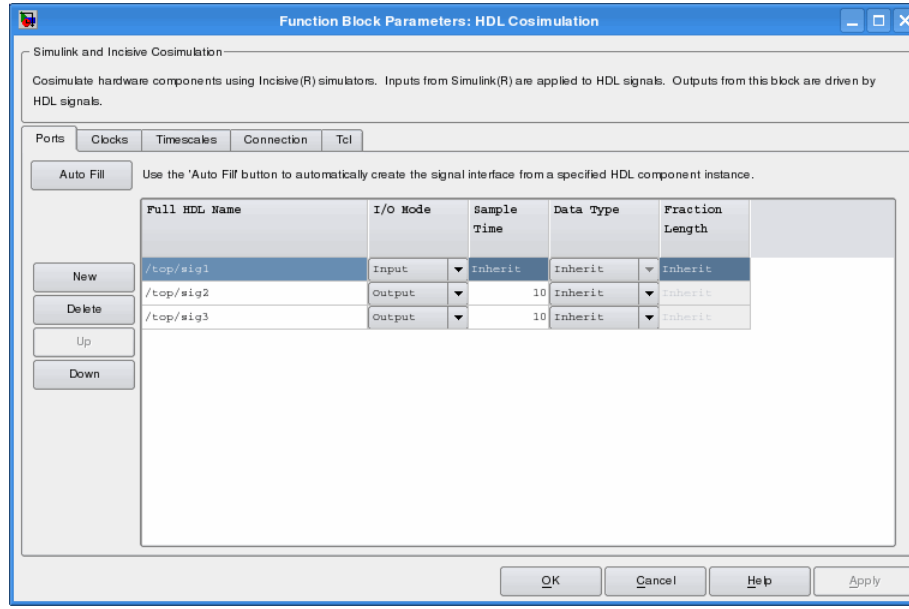
The block mask contains panels for entering port and signal information, setting communication modes, adding clocks, specifying pre- and post-simulation Tcl commands, and defining the timing relationship.

After you code one of your model's components in VHDL or Verilog and simulate it in the Incisive or NC 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 Link for Cadence Incisive library, is shown below.

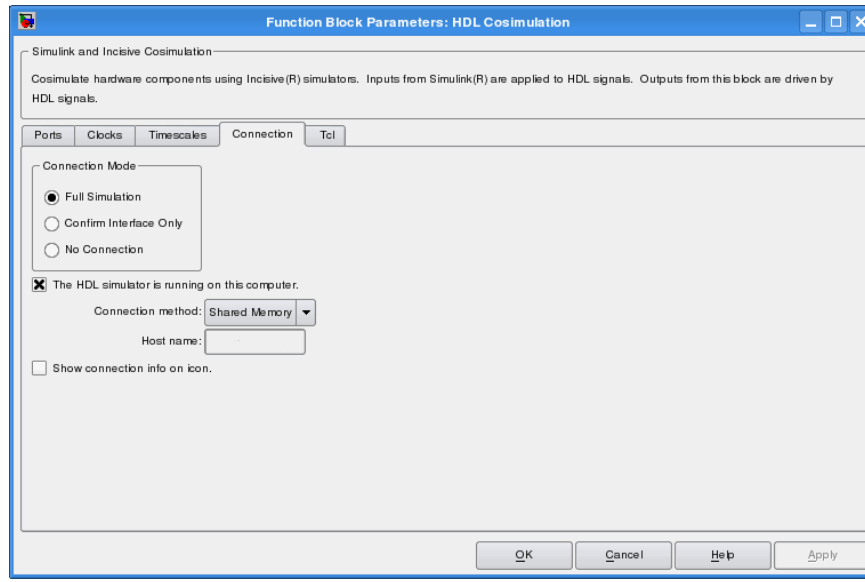


You configure an HDL Cosimulation block by specifying values for parameters in a block parameters dialog. The HDL Cosimulation block parameters dialog consists of five tabbed panes that specify the following:

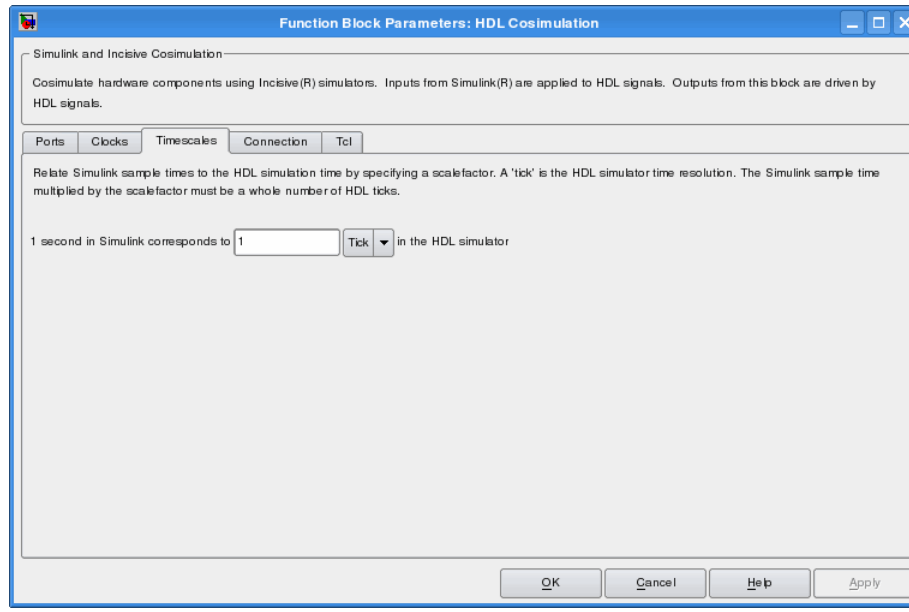
- **Ports Pane**: Block input and output ports that correspond to signals, including internal signals, of your HDL design, and an output sample time. See “Ports Pane” on page 6-6.



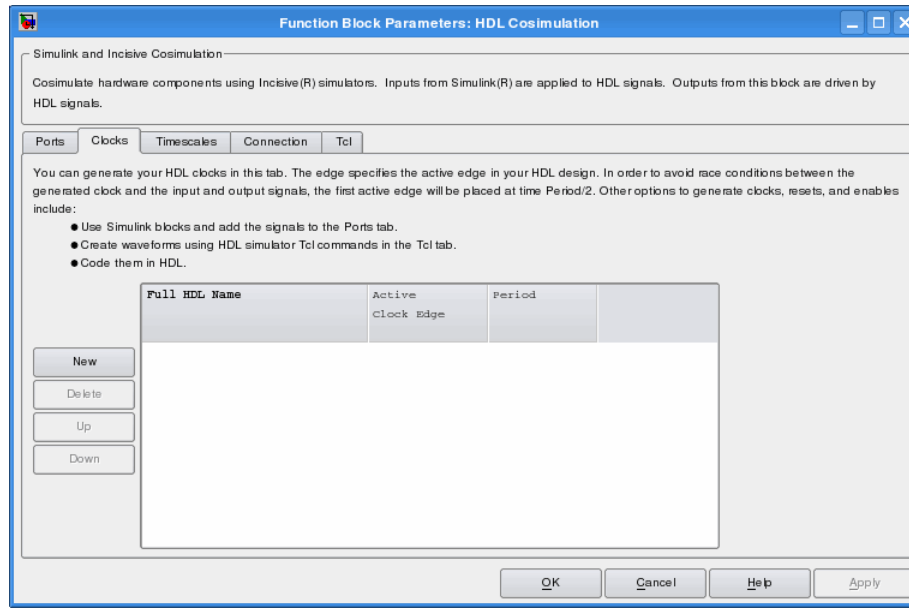
- **Connection Pane**: Type of communication and communication settings to be used for exchanging data between simulators. See “Connection Pane” on page 6-10.



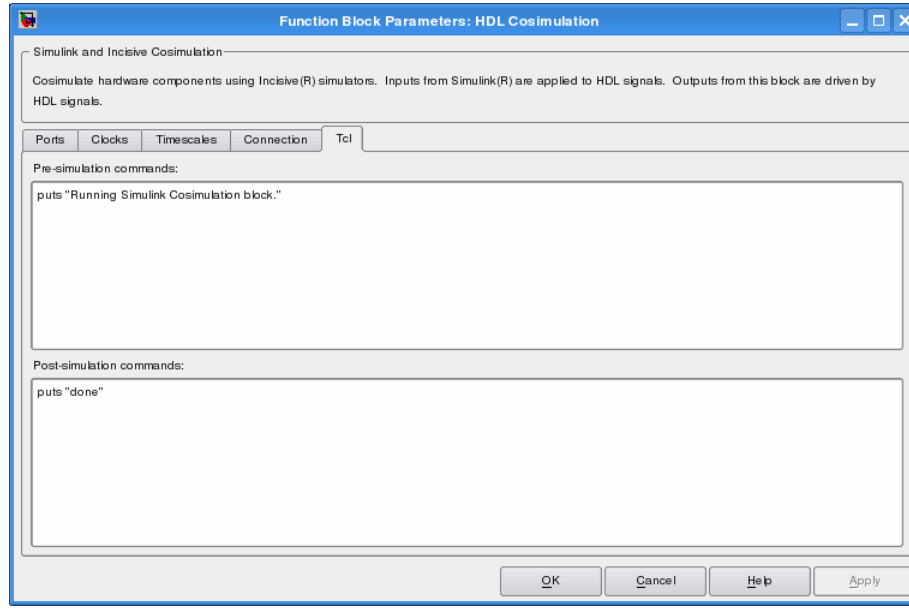
- **Timescales Pane**: Timing relationship between Simulink and Cadence Incisive. See “Timescales Pane” on page 6-14.



- **Clocks Pane**: Optional rising-edge and falling-edge clocks to apply to your model. See “Clocks Pane” on page 6-17.



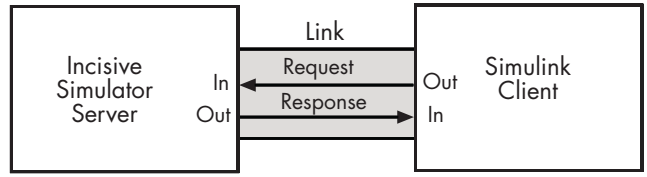
- **Tcl Pane**: Tcl commands to run before and after a simulation. See “Tcl Pane” on page 6-19.



Note You must make sure that signals being used in cosimulation have read/write access (this is done through the HDL simulator – see product documentation for details). This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

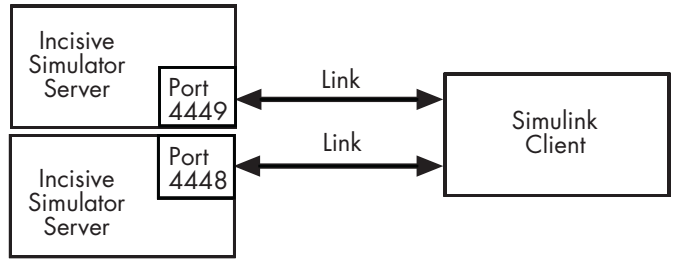
Communicating Between Cadence Incisive and Simulink

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 initiate a cosimulation session from Simulink. After a session is started, you can use Simulink and the HDL simulator to monitor simulation progress and results.

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

As you develop a Link for Cadence Incisive cosimulation application, you should be familiar with how signal values are handled across the simulation domains with respect to

- How Simulink drives cosimulation signals
- Representation of simulation time
- Handling multirate signals
- Handling Frame-based signals

- Clock signal latency
- 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 connect 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. Simulink uses the deposit method of changing signal values to drive input to a cosimulation block. The deposit method is the weakest method of forcing an HDL signal and can produce unexpected or undesired results when a signal is driven by multiple sources. To avoid such conditions, you should attach the input ports to signals that are not driven, such as the input ports of a top-level VHDL entity.

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 You must make sure that signals being used in cosimulation have read/write access (this is done through the HDL simulator – see product documentation for details). This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Representation of Simulation Time

In the HDL simulator, the unit of simulation time is referred to as a *tick*. The duration of a tick is defined by the HDL simulator *resolution limit*. The default resolution limit is 1 ns.

To determine the current HDL simulator resolution limit, enter `echo $timescale` at the HDL simulator prompt. See the HDL simulator documentation for the application you are using for further information.

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

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

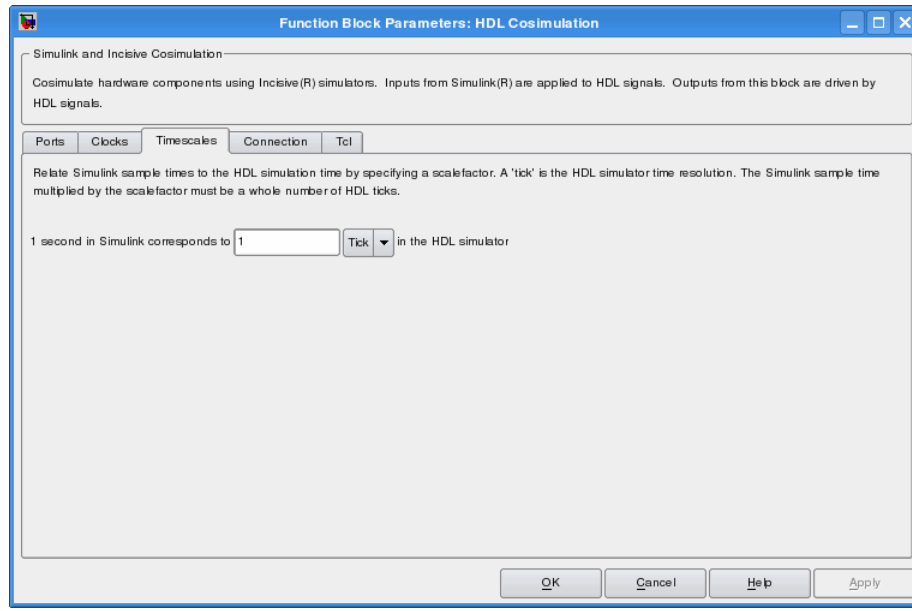
- Total simulation time
- Input port sample times
- Output port sample times
- Clock periods

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

The Link for Cadence Incisive software provides controls that let you configure the timing relationship between the Incisive simulator and Simulink and avoid timing errors caused by differences in timing representation.

Defining the Simulink and Cadence Incisive 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 Cadence Incisive:

- *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.

In relative timing mode, all sample times and clock periods in Simulink are rounded to the nearest integer number of seconds so that they can be directly translated into ticks in the HDL simulator. The following pseudocode shows how Simulink time units are quantized to HDL simulator ticks:

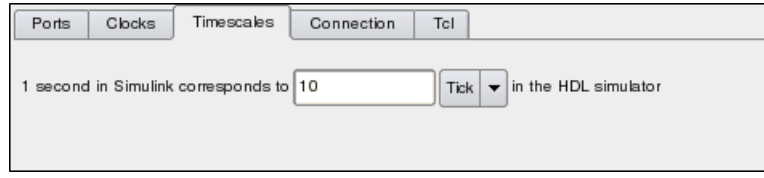
```
InTicks = N * tInSecs
```

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

To configure relative timing mode for a cosimulation:

1. Click 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.

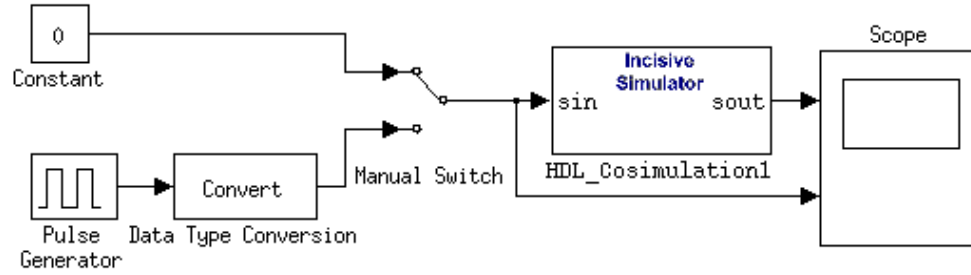
Operation of Relative Timing Mode. By default, the HDL Cosimulation block is configured for relative mode, with a scale factor of 1. Thus, 1 Simulink second corresponds to 1 tick in the HDL simulator. In the default case:

- If the total simulation time in Simulink is specified as `N` seconds, then the Cadence Incisive 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.
- Clocks operate in a similar fashion. Where a clock has a period of *T* seconds:
  - If *T* is even, the clock signal is forced in the HDL simulator as an input signal that stays low for *T/2* ticks and stays high for *T/2* ticks.
  - If *T* is odd, the clock signal is forced in the HDL simulator as an input signal that stays low for *T/2* ticks and stays high for *(T/2) + 1* ticks.

  Note that Simulink requires such clocks to have a period of at least 2 seconds. Simulink throws an error if specified value of *T* is less than 2 seconds.

To understand how relative timing mode operates, review cosimulation results from the following example model.



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`
  - Clock (`inverter_clock_vl.clk`) period: `5 s`
- HDL simulator resolution limit: `1 ns`

The previous example was excerpted from the Link for Cadence Incisive Inverter tutorial. For more information, see Link for Cadence Incisive demos.

Absolute Timing Mode

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

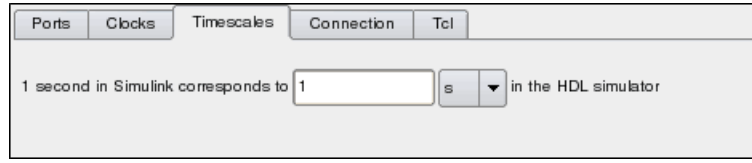
- *One second* in Simulink corresponds to *(N * Tu) seconds* in the HDL simulator, where *Tu* is an absolute time unit (e.g., ms, ns, etc.) and *N* is a scale factor.

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

To configure absolute timing mode for a cosimulation:

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.

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 3-18. Suppose that the model is reconsidered 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 cosimulates 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 a HDL Cosimulation block between consecutive cosimulation runs, you must restart the simulation in the Incisive or NC simulator.

Setting HDL Cosimulation Port Sample Times

In general, Simulink handles the sample time for the ports of a HDL Cosimulation block as follows:

- If an input port is connected to a signal that has an explicit sample time, based on forward propagation, Simulink applies that rate to that input port.
- If an input port is connected to a signal that *does not have* an explicit sample time, Simulink assigns a sample time that is equal to the least common multiple (LCM) of all identified input port sample times for the model.
- After Simulink sets the input port sample periods, it applies user-specified output sample times to all output ports. Sample times must be explicitly defined for all output ports.

If you are developing a model for cosimulation in *relative* timing mode, consider the following sample time guideline:

- Specify the output sample time for an HDL Cosimulation block as an integer multiple of the resolution limit defined in the HDL simulator. Use the HDL simulator command `report simulator state` to check the resolution limit of the loaded model. If the HDL simulator resolution limit is 1 ns and you specify a block's output sample time as 20, Simulink interacts with the HDL simulator every 20 ns.

Handling Multirate Signals

Link for Cadence Incisive 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, a 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. 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 “Requirements and Restrictions” on page 3-25. 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 Incisive or NC simulator does not change in any way. Simulink assumes that the Incisive or NC 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 Incisive or NC simulator one sample at a time.

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).

Clock Signal Latency

In the Incisive or NC simulator, it is not possible to guarantee the order in which clock signals (rising-edge or falling-edge) defined in the HDL Cosimulation block are applied, relative to the data inputs driven by these clocks. Therefore, it is possible that during a cosimulation, race conditions could develop between a clock and the data inputs associated with the clock.

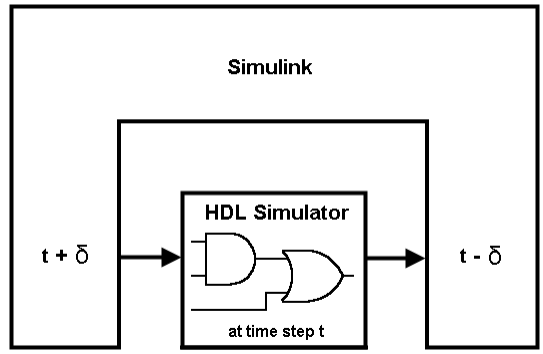
To avoid such race conditions, Link for Cadence Incisive delays all such clocks by ½ clock period, in effect inverting the sense of the rising or falling edge. The delay provides a setup and hold time for input data, ensuring that data inputs are always applied before the driving clock edge is applied. For example, in the case of a rising-edge clock, inputs are applied first, and ½ clock period later, the rising edge of the clock is applied.

Where the Simulink sample time is even, the clock delay will be exactly ½ period. For odd Simulink sample times, the ½ period delay is approximated as closely as possible. While this apparent "inversion" or delay by ½ period of the active edge of the clock can be confusing, it enables cosimulation to work correctly without race conditions and without requiring separately specified setup and hold times for the data.

Block Simulation Latency

Simulink and the Link for Cadence Incisive cosimulation blocks supplement the hardware simulator environment, rather than operate as part of it. During cosimulation, Simulink does not participate in 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 Link for Cadence Incisive cosimulation blocks. The latency is a time lag that occurs between when Simulink initiates 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 contain pure combinatorial paths, however, it might be necessary to adjust the sample time to achieve simulation performance required for circuit analysis.

For cosimulation blocks that model combinatorial circuits, you may want to experiment with a faster sample frequency for output ports. Although this type of parameter tuning can increase simulation performance, it can also make a model more difficult to debug. For example, you may need to adjust the output sample time for each cosimulation block.

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 Configuration Parameters

When you create a Simulink model that includes one or more Link for Cadence Incisive 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 Link for Cadence Incisive blocks. Gather and save test bench data that you can use later for comparing the model with a version that includes Link for Cadence Incisive blocks.

Simulink and HDL Simulator Communication Options

Select shared memory or socket communication. See “Modes of Communication” on page 1-7.

Starting the HDL Simulator

See “Starting the HDL simulator for Use with Link for Cadence Incisive” on page 1-21.

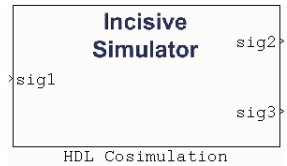
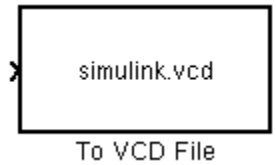
Incorporating Hardware Designs Into a Simulink Model

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:

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 Link for Cadence Incisive 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 3-54.

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 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 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
```

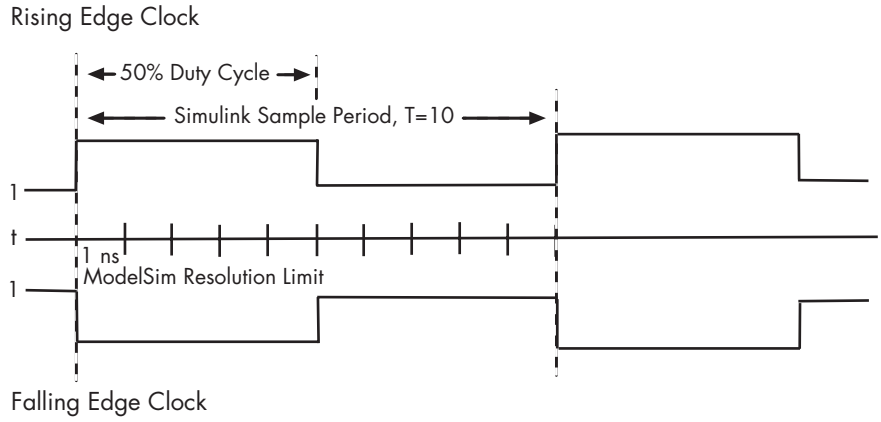
Driving Clocks, Resets, and Enables

Creating Optional Clocks

You can create rising-edge or falling-edge clocks that apply internal stimuli to your cosimulation model. When you specify a clock in your block definition, Simulink creates a rising-edge or falling-edge clock that drives the specified HDL signals by depositing them.

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

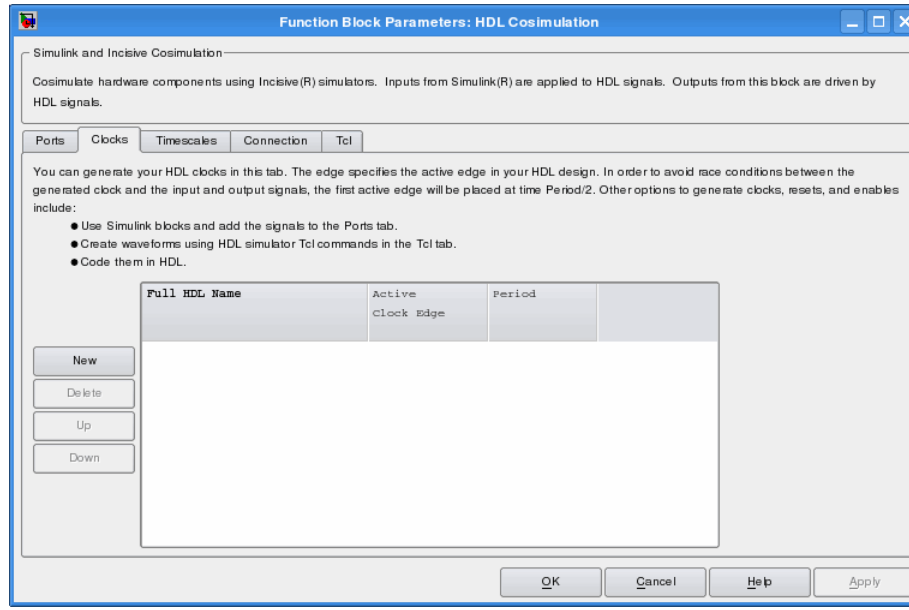
The following figure shows a timing diagram that includes rising and falling edge clocks with a Simulink sample time of `T=10` and an HDL simulator resolution limit of 1 ns. The figure also shows that given those timing parameters, the clock duty cycle is 50%.



To create clocks,

1. In the Incisive or NC simulator, determine the clock signal pathnames you plan to define in your block. To do this, you can use the same method explained for determining the signal pathnames for ports in “Mapping HDL Signals to Block Ports” on page 3-36.

2 Select the **Clocks** tab of the Block Parameters dialog. Simulink displays the dialog as shown below.



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

4 Edit the clock signal pathname directly in the table under the **Full HDL Name** column by double-clicking on the default clock signal name (`/top/clk`). Specify your new clock using HDL simulator pathname syntax. See “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 3-31.

Note that vectored signals in the **Clocks** pane are not supported. Signals must be logic types with ’1’ and ’0’ values.

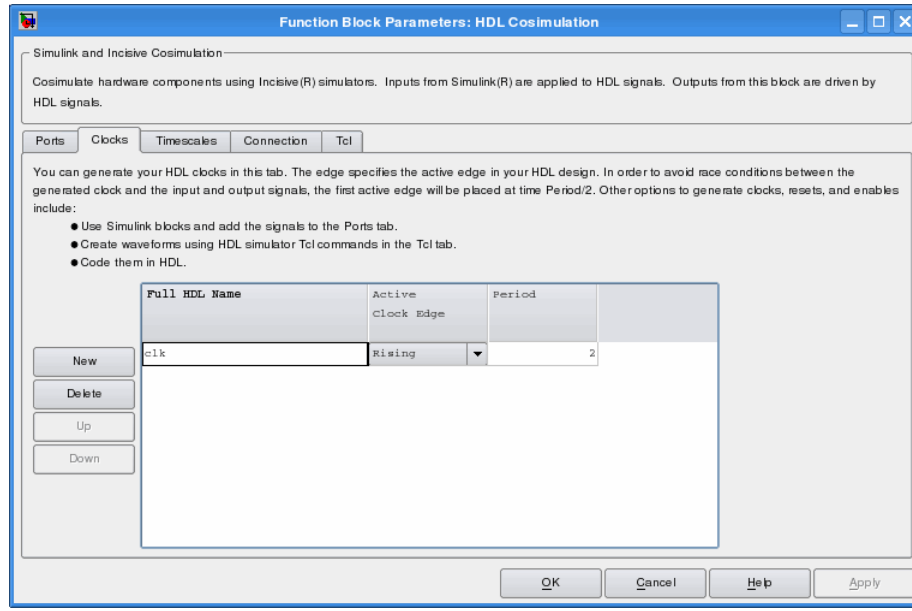
5 To specify whether the clock generates a rising-edge or falling edge signal, select `Rising` or `Falling` from the **Active Clock Edge** list.

6 The **Period** field specifies the clock period. Accept the default (`2`), or override it by entering the desired clock period explicitly by double-clicking in the **Period** field.

Specify the **Period** field as an even integer, with a minimum value of 2.

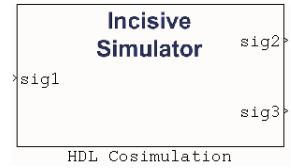
7 When you have finished editing clock signals, click **Apply** to register your changes with Simulink.

The following dialog defines the rising-edge clock `clk` for the HDL Cosimulation block, with a default period of 2.

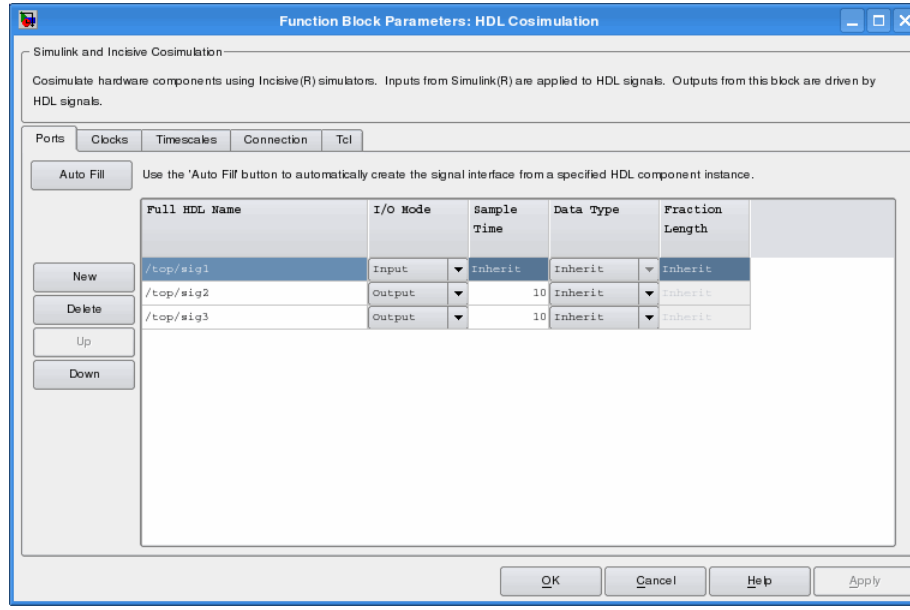


Defining the Block Interface

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 Link for Cadence Incisive 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

- Enter signal information manually into the **Ports** pane of the HDL Cosimulation Block Parameters dialog (see “Entering Signal Information Manually” on page 3-42). 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 a

large number of signals that you want to connect to your Simulink model. Note, however, that in many cases you will need to edit the signal data returned by the query. See "Obtaining Signal Information Automatically from Cadence Incisive" on page 3-37 for details.

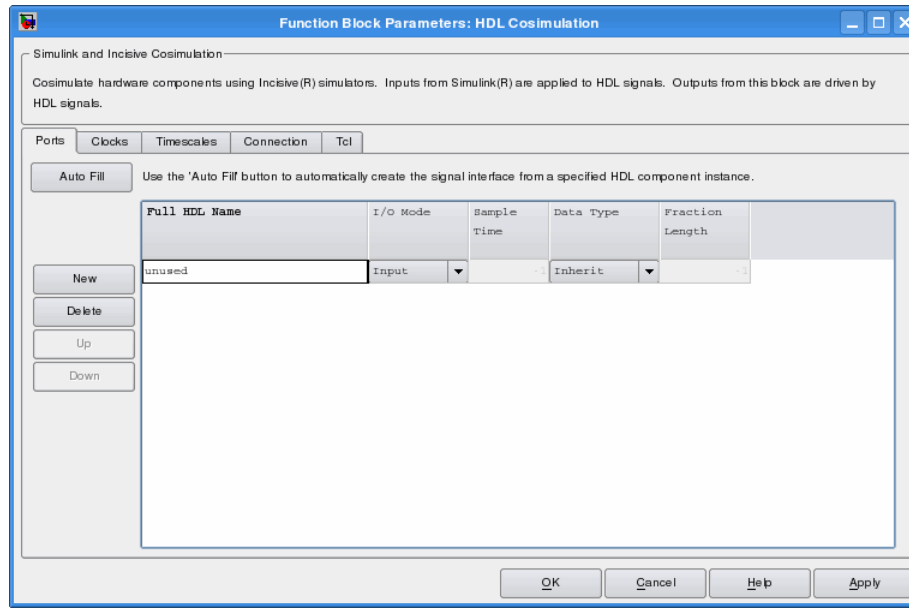
Note You must make sure that signals being used in cosimulation have read/write access (this is done through the HDL simulator – see product documentation for details). This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Obtaining Signal Information Automatically from Cadence Incisive.

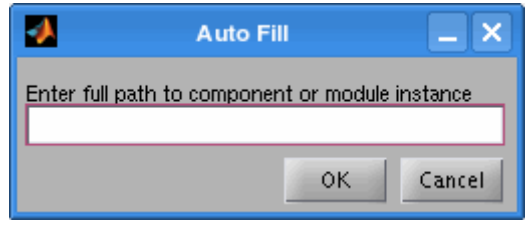
The **Auto Fill** button lets you initiate an HDL simulator query and supply a path to a component or module in an HDL model under simulation in the Incisive or NC simulator. Usually, some modification of the port information is required after the query completes.

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" on page 3-5), in which all signals were initially deleted from the **Ports** and **Clocks** panes.

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

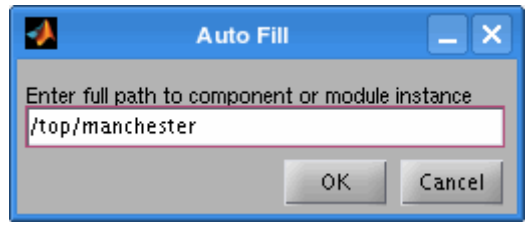


2 Click the **Auto Fill** button. The **Auto Fill** dialog opens.



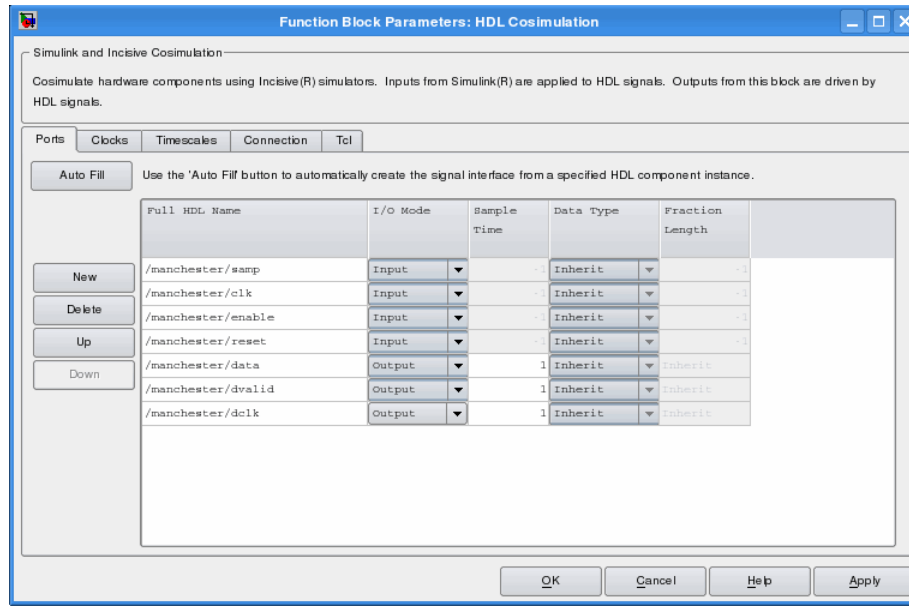
This modal dialog requests a path to a component or module in your HDL model; here you enter an explicit HDL path into the edit field.

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 almost instantaneously, as shown in the figure below.



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

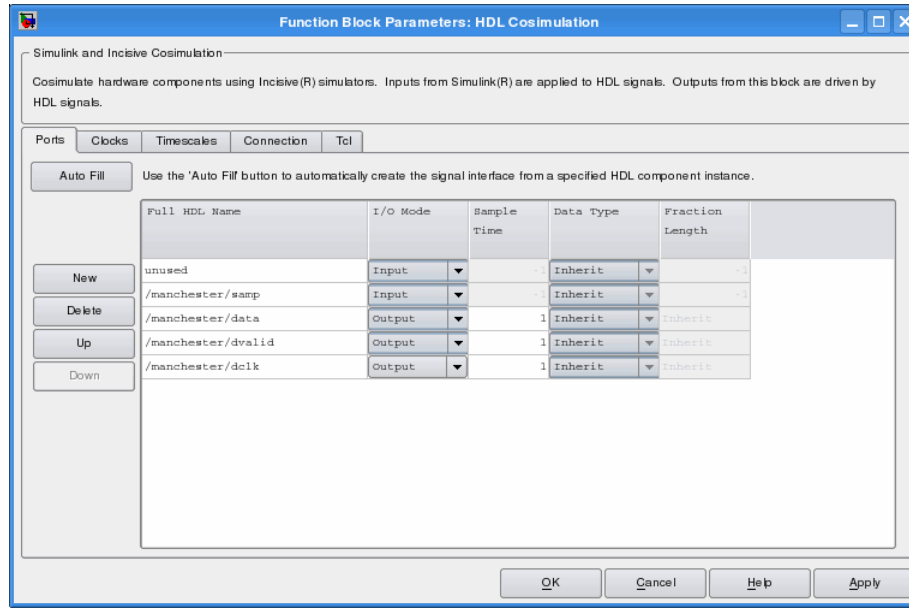
7 Observe that **Auto Fill** has returned information about *all* inputs and outputs for the targeted component. In many cases, this will include signals that function in the HDL simulator but cannot be connected in the Simulink model. You should delete any such entries from the list in the **Ports** pane unless you are adding blocks to the Simulink model to represent these signals.

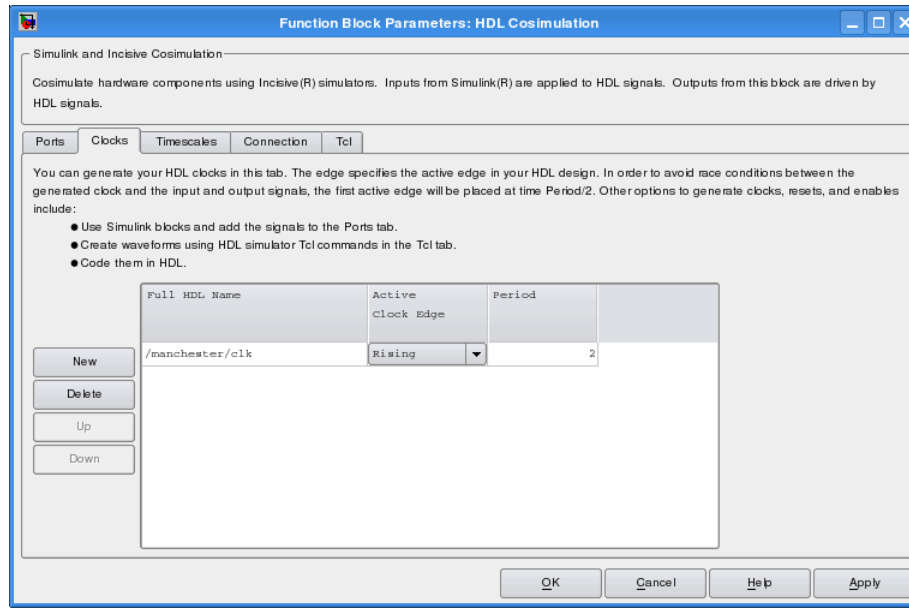
The figure above shows that the query entered clock, clock enable, and reset ports (labelled `clk`, `enable`, and `reset` respectively) into the ports list. In this example, the `clk` signal is entered in the **Clocks** pane, and the `enable` and `reset` signals are deleted from the **Ports** pane, as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

Note Enter force commands in the **Tcl** pane to drive the reset and enable signals; for example:

```
force design/reset value time
```

where *`value`* is '1' or '0' and *`time`* is in nanoseconds.





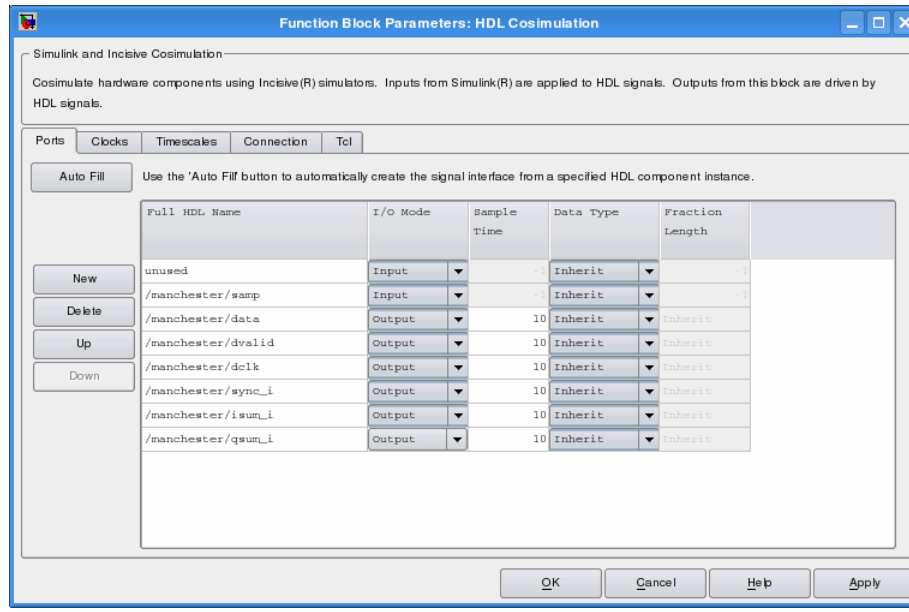
8 **Auto Fill** returns default values for output ports:

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

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 3-45.

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 `/manchester/sync_i`, `/manchester/isum_i`, and `/manchester/qsum_i`, as shown below.

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

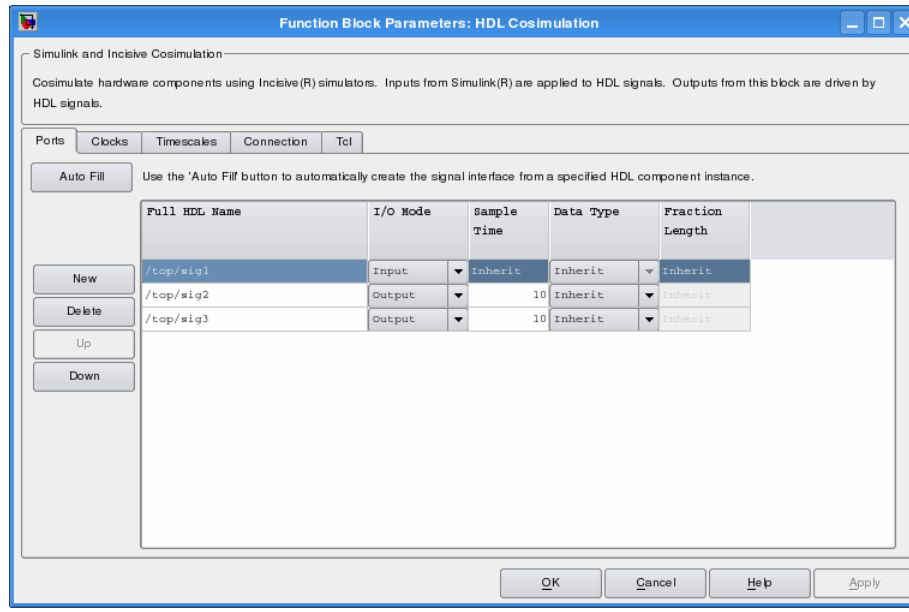


Entering Signal Information Manually. To enter signal information directly in the **Ports** pane:

1. In the HDL simulator, determine the test signal pathnames for the HDL signals you plan to define in your block. The HDL simulator signal pathname feature allows you to visualize and specify the hierarchy of signals in a HDL design.

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:

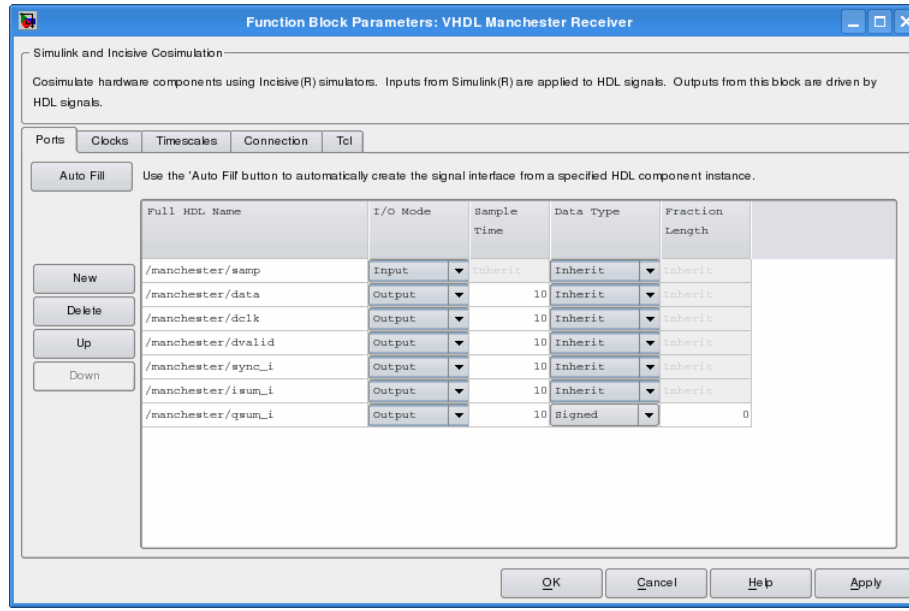
- For a device having both inputs and outputs: specify block input ports, block output ports, output sample times and output data types. For output ports, accept the default or enter an explicit sample time. Data types can be specified explicitly, or set to `Inherit` (the default). In the default case, 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 test signal pathnames in the **Full HDL name** column by double-clicking on the existing default signal. Use Incisive or NC simulator pathname syntax (see “Specifying HDL Signal/Port and Module Paths for Cosimulation” on page 3-31). 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.

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

The following dialog shows port definitions for an HDL Cosimulation block.



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

5 You must specify a sample time for the output ports. Output sample times are specified as integers. 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 Link for Cadence Incisive environment, see “Representation of Simulation Time” on page 3-15.

6 You can configure the fixed-point data type of each output port explicitly if desired, or use a default (`Inherited`) . In the default case, Simulink determines the data type for an output port as follows:

If Simulink can determine the data type of the signal connected to the output port, it applies that data type to the output port. For example, the data type of a connected Signal Specification block is known by back-propagation. Otherwise, Simulink queries the HDL simulator to determine the data type of the signal from the HDL model.

To assign an explicit fixed-point data type to a signal:

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.

- 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

- 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`.

Relating Simulink's and Cadence Incisive's Sense of Time

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 3-15 to understand the supported timing modes and the issues that will determine your choice of timing mode.

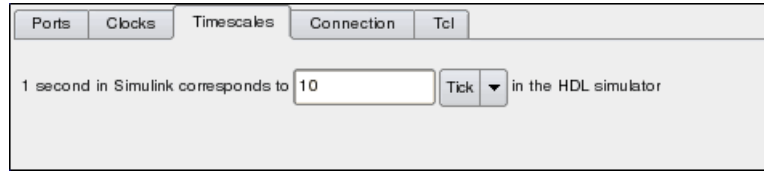
You can specify either a relative or an absolute timing relationship between Simulink and the HDL simulator, as described in the sections below.

Specifying a Relative Timing Relationship

To configure relative timing mode for a cosimulation:

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

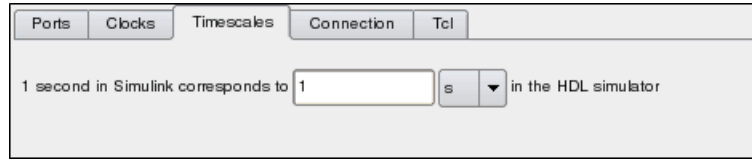
To configure absolute timing mode for a cosimulation:

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. The figure that follows shows the steps in a flow diagram:

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 D-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 “Modes of Communication” on page 1-7.

   If you choose TCP/IP socket communication, specify a valid port number or service for your computer system in the **Port number or service** text field. For information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page D-2.

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

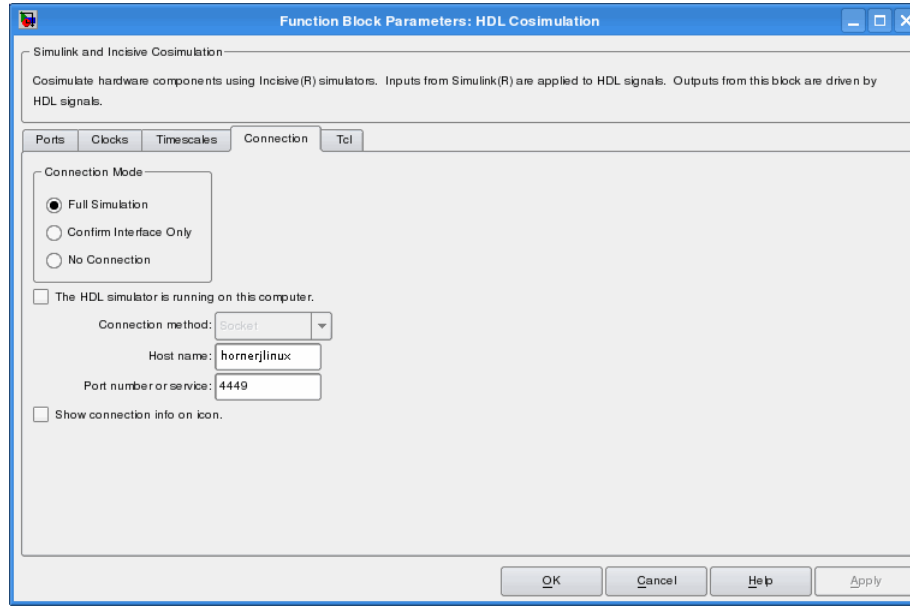
5. If you want to bypass the HDL simulator when you run a Simulink simulation, use the **Connection Mode** options to specify what type of simulation connection you want. Select one of the following:

   - **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, Link for Cadence Incisive 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.



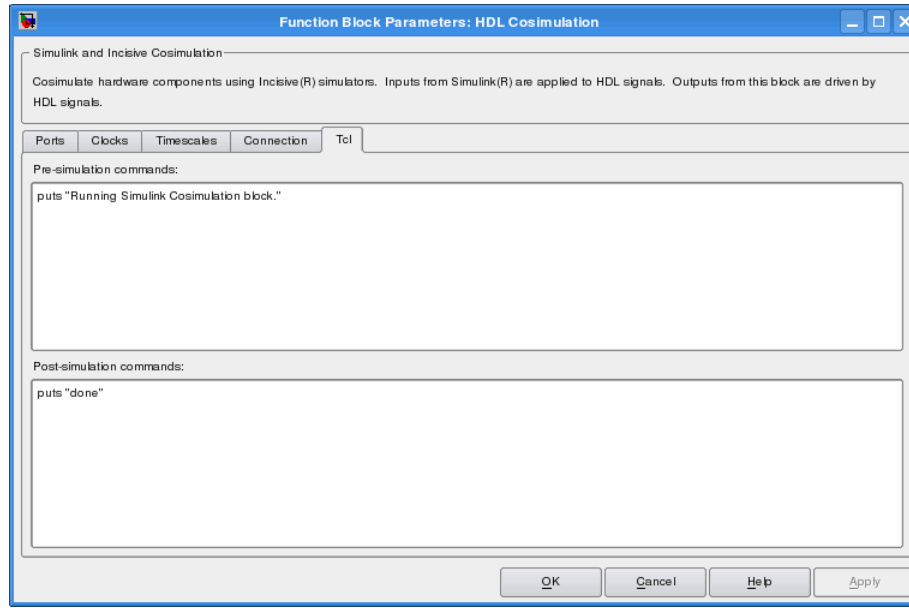
Specifying Pre- and Post-Simulation Tcl Commands with HDL Cosimulation Block Parameters Dialog Box

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

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

To specify Tcl commands,

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



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

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

3 Click **Apply**.

Programmatically Controlling the Block Parameters

One way to control block parameters is through theHDL 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 Link for Cadence Incisive 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 demonstrate 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 demonstrates 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);
```

Programmatically Adding Debug Commands to Pre/Post Tcl Commands. This example code demonstrates how to dynamically control the use of HDL waveform generation when you move from interactive debug testing to regression testing.

```
% note: the 'hdlDebugMode' variable is assumed to be defined in the workspace.
%       there are many ways to define the variable and control its value, including
%       some of the callback methods described above.  Another idea is to use a
%       Simulink Switch or Manual Switch block.
if (hdlDebugMode)
  tclPreCmds = 'puts { Starting cosimulation.}; probe /top -waveform';
else
  tclPreCmds = 'puts { Starting cosimulation.};'
end

set_param('MyModel/HDL Cosimulation', 'TclPreSimCommand', tclPreCmds);
```

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

- 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

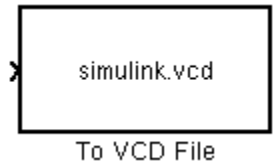
VCD files can provide data that you might not otherwise acquire unless you understood the details of a device's internal logic. In addition, they include data that can be graphically displayed or analyzed with postprocessing tools. For example, including the extraction of data pertaining to a particular section of a design hierarchy or data generated during a specific time interval.

The To VCD File block provided in the Link for Cadence Incisive block library serves as a VCD file generator during an HDL simulator and Simulink cosimulation session. 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 filename.

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,

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 Link for Cadence Incisive library. The browser displays four types of blocks, one of which is the To VCD File block.



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.

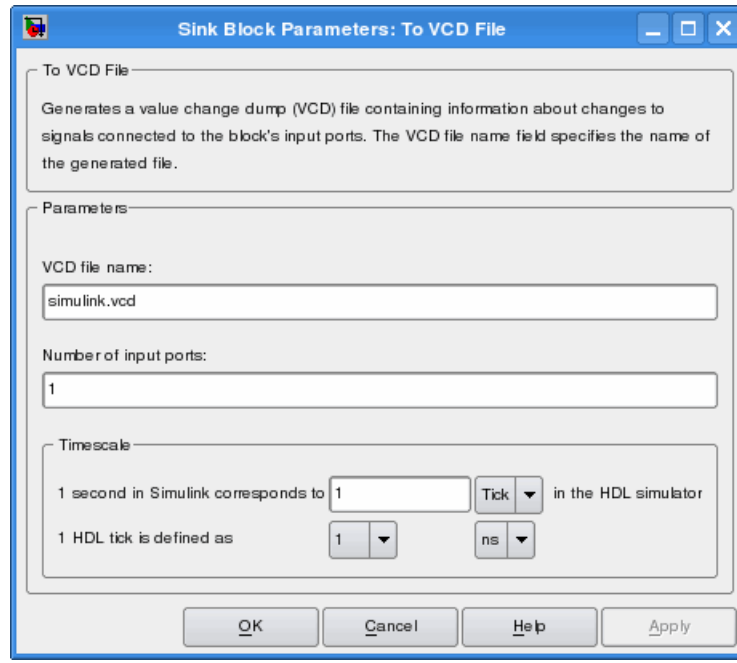
Note The To VCD File block does not support floating point signal types.

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.

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



b Specify a filename for the generated VCD file in the **VCD file name** text box. If you specify a filename only, Simulink places the file in your current MATLAB directory. Specify a complete pathname 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) bits, each of which maps to a unique symbol in the VCD file.

d Click **OK**.

7 Choose an optimal 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 To VCD File.

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 6-25.

Running Cosimulations

In this section...

Starting Cadence Incisive 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 `nclaunch`. Alternatively, you can start the HDL simulator manually and load the Link for Cadence Incisive libraries yourself. Either way, see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21.

Loading an HDL Module for Cosimulation

After you start the HDL simulator from MATLAB with a call to `nclaunch`, load an instance of an HDL module for cosimulation with the Link for Cadence Incisive HDL simulator command `hdlsimulink`. Issue the command for each instance of an HDL module in your model that you want to cosimulate. For example:

```
hdlsimulink work.manchester
```

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

Determining an Available Socket Port Number

To determine an available socket port number, use `hdldaemon` with the property name 'socket' set to value 0 at the MATLAB command prompt:

```
>> hdldaemon('socket',0)
HDLDaemon socket server is running on port 2274 with 0 connections
```

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 need to reset a clock during a cosimulation, you can do so by entering HDL simulator `force` commands at the HDL simulator command prompt or by specifying HDL simulator `force` commands in the **After simulation command** text field on the **Tcl** pane of your Link for Cadence Incisive block's parameters dialog.

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

MATLAB Functions — Alphabetical List

dec2mvl

Purpose

Convert decimal integer to binary string

Syntax

```
dec2mvl(d)
dec2mvl(d,n)
```

Description

`dec2mvl(d)` returns the binary representation of `d` as a multivalued logic string. `d` must be an integer smaller than 2^52.

`dec2mvl(d,n)` produces a binary representation with at least `n` bits.

Examples

The following function call returns the string '10111':

```
dec2mvl(23)
```

The following function call returns the string '01001':

```
dec2mvl(-23)
```

The following function call returns the string '11101001':

```
dec2mvl(-23,8)
```

See Also

`mvl2dec`

Purpose

Start MATLAB server component of Link for Cadence Incisive software

Syntax

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

Description

Server Activation

`hdldaemon` starts the Link for Cadence Incisive MATLAB server component 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 Incisive simulator), using shared memory communication when your application configuration consists of a single system can result in increased performance.

Only one `hldaemon` 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 `matlabtb` , `matlabtbeval`, or `matlabcp` command in the Incisive 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 Incisive simulator must match the socket being used by the server.

For more information on modes of communication, see “Modes of Communication” on page 1-7. For more information on establishing the Incisive simulator end of the communication link, see “Associating the HDL Module Component with the MATLAB Link Function” on page 2-35.

`hdldaemon('PropertyName', 'PropertyValue'...)` starts the Link for Cadence Incisive MATLAB server component with property-value pair settings that specify the communication mode for the link between MATLAB and the Incisive simulator, the time resolution for the MATLAB simulation function output ports, and, optionally, a Tcl command to be executed immediately in the HDL simulator. See "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 Incisive simulator:

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

You can also use this function to check on the mode of communication 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 Incisive 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.

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 Incisive 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. Some valid `tcp_spec` examples follow:

Option	Examples
Port number	`'4449'` or `4449`
Alias or service name	`'MATLAB Service'`
Operating system assigned	`'0'` or `0`

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page D-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`, `matlabtbeval`, or `matlabcp` command in the Incisive simulator.

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

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

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

`'tclcmd', 'command'`
Passes a Tcl command string, to be executed immediately in the Incisive simulator, from MATLAB to the Incisive simulator. You may use a compound command and separate the commands with semicolons.

Note The Tcl command string you specify cannot include commands that load an Incisive simulator project or modify simulator state. For example, the string cannot include commands such as `run`, `stop`, or `reset`.

`'quiet', 'true'`
: Suppresses printing messages to the standard queue. Errors are still shown.

Examples

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 D-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'` .

If Your Application Is to...	Do the Following...
Execute Tcl command immediately upon simulator connection	Specify the `'tclcmd', '`*`command`*`'` property name and value pair. Command must be a valid Tcl command but cannot include commands that load an Incisive simulator project or modify the simulator state.
Suppress server shutdown message when using `hdldaemon` to get an unused socket number (message can appear confusing)	Specify `'quiet', 'true'`.

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 output ports:

```
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')
```

The following function call causes the string `This is a test` to be displayed at the Incisive simulator prompt:

```
hdldaemon('tclcmd','puts {This is a test}')
```

The following is an example of a compound Tcl command used with `hdldaemon`:

```
hdldaemon('tclcmd','{force filter2d_v.clk_enable 1
          -after 0ns;
          force filter2d_v.reset 1 -after 0 ns 0 -after 1 ns;
          puts {Running Simulink Cosimulation block};
          puts [clock format [clock seconds]]}')
```

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 2's 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

Start and configure Incisive simulators for use with Link for Cadence Incisive software

Syntax

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

Description

`nclaunch('PropertyName', 'PropertyValue'...)` starts the Incisive simulator for use with the MATLAB and Simulink features of the Link for Cadence Incisive software. The initial directory in the Incisive simulator matches your MATLAB current directory if no explicit `rundir` parameter is specified.

After you call this function, you can use HDL Simulator Tcl Commands to do interactive debug setup.

The property name/property value pair settings allow you to customize the Tcl commands used to start the Incisive simulator, the `ncsim` executable to be used, the path and name of the Tcl script that stores the start commands, and for Simulink applications, details about the mode of communication to be used by the applications. You must use a property name/property value pair with `nclaunch`.

Property Name/Property Value Pairs

`'hdlsimdir', 'pathname'`
: Specifies the pathname to the Incisive simulator executable to be started. By default, the function uses the first version of the simulator that it finds on the system path (defined by the `path` variable) . Use this option to start different versions of the Incisive simulator or if the version of the simulator you want to run does not reside on the system path.

`'hdlsimexe', 'simexename'`
: Specifies the name of an Incisive simulator executable. By default, this function uses `'ncsim'`. You can specify a custom-built simulator executable with `'simexename.'`

`'libdir', 'directory'`
: Specifies the directory containing MATLAB shared libraries. This property creates an entry in the startup Tcl file that points to the directory with the shared libraries needed for the Incisive

simulator to communicate with MATLAB when the Incisive simulator is running on a machine that does not have MATLAB.

`'rundir', 'dirname'`
: Specifies where to run the HDL simulator. By default, the function uses the current working directory.

  - If `dirname` is specified and the directory exists, the HDL simulator is run in the specified directory.
  - If no `rundir` property/value pair is specified or if `dirname` is empty, the HDL simulator is run in the current working directory.
  - If the value of `dirname` is "TEMPDIR", the function creates a temporary directory in which it runs the HDL simulator.
  - If `dirname` is specified and the directory does *not* exist, you will get an error.

`'startupfile', 'pathname'`
: Specifies a Tcl script that defines the behavior of the Incisive simulator commands `hdlsimmatlab` and `hdlsimulink`. The Tcl script consists of some general-purpose Tcl commands for launching the Incisive simulator and any commands you specify with the `'tclstart'` property. If you omit this property, the function creates a temporary file each time the Incisive simulator starts. If you specify a name for the Tcl script, later you can use the file to start the Incisive simulator from a system shell as shown in the following syntax:

  ```
  tclsh tcl_scriptname
  ```

`'socketsimulink', 'tcp_spec'`
: Specifies TCP/IP socket communication for links between the Incisive simulator and Simulink. 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`

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

If the Incisive simulator and Simulink are running on the same computing system, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you omit `socketsimulink` *`tcp_spec`* from the function call.

`'starthdlsim', ['yes' | 'no']`
: Determines whether the Incisive simulator is launched. The default is `yes`, which launches the Incisive simulator and creates a startup Tcl file. If `starthdlsim` is set to `no`, the Incisive simulator is not launched, but a startup Tcl file is still created.

  This startup Tcl file contains pointers to MATLAB and Simulink shared libraries. To run the Incisive simulator manually, see "Starting the HDL simulator for Use with Link for Cadence Incisive" on page 1-21.

`'tclstart', 'tcl_commands'`
: Specifies one or more Tcl commands to execute before the Incisive simulator launches. Specify a command string or a cell array of command strings. You must specify at least one command; otherwise, no action occurs.

Note You must put "exec" in front of non-Tcl system shell commands. For example:

```
exec -ncverilog -c +access+rw +linedebug top.v
hdlsimulink -gui work.top
```

Examples

The following function call sequence compiles the design and starts Simulink with a GUI from the "proj" directory with the model loaded. Simulink is instructed to communicate with the Link for Cadence Incisive interface on socket port 4449. All of these commands are specified in a single string as the property value to `tclstart`.

```
nclaunch(...
'tclstart',...
{'exec ncverilog -c +access+rw +linedebug top.v',...
'hdlsimulink -gui work.top'},...
'socketsimulink','4449',...
'rundir', '/proj');
```

In this next example, `tclcmd` is used to build the sequence of Tcl commands that are executed in a Tcl shell after calling `nclaunch` from MATLAB.

- `tclcmd{1}` compiles `vlogtestbench_top`.
- `tclcmd{2}` elaborates the model.
- `tclcmd{3}` calls `hdlsimmatlab` in `gui` mode and loads the elaborated `vlogtestbench_top` in the simulator.

The arguments being passed with `input` (`matlabtb` and `run`) are executed in the ncsim Tcl shell. In this example, `matlabcp` associates the m-function `vlogmatlabc` to the module instance `u_matlab_component`. It assumes that the `hdldaemon` in MATLAB is listening on port 32864. `run` will run 50 resolution units (ticks).

```
tclcmd{1} = 'exec ncvlog vlogtestbench_top.v'
tclcmd{2} = 'exec ncelab -access +wc vlogtestbench_top'
tclcmd{3} = ['hdlsimmatlab -gui vlogtestbench_top ' ...
           '-input "{@matlabcp vlogtestbench_top.u_matlab_component...
                      -mfunc vlogmatlabc -socket 32864}" '...
           '-input "{@run 50}"']
nclaunch('hdlsimdir', 'local.IUS.glnx.tools.bin', 'tclstart',tclcmd);
```

The following example demonstrates using the property `startupfile` to designate a Tcl script that is then used to start the HDL simulator from the Tcl shell.

In MATLAB:

```
nclaunch (`tclstart', `xxx', `startupfile', `mytclscript',...
          `starthdlsim', `no')
```

In Tcl shell:

```
shell> tclsh mytclscript
```

Purpose

Blocks 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')
```

Purpose

Executes Tcl command in HDL simulator

Syntax

```
tclHdlSim(tclCmd)
tclHdlSim(tclCmd,'portNumber')
tclHdlSim(tclCmd, 'portnumber', 'hostname')
```

Description

`tclHdlSim(tclCmd)` executes a Tcl command on the HDL simulator using a shared connection.

`tclHdlSim(tclCmd,'portNumber')` executes a Tcl command on the HDL simulator by connecting to the local host on port *portNumber*.

`tclHdlSim(tclCmd, 'portnumber', 'hostname')` executes a Tcl command on the HDL simulator by connecting to the host *hostname* on port *portname*.

The Incisive or NC simulator must be connected to MATLAB using Link for Cadence Incisive for this function to work (see “Starting the HDL simulator for Use with Link for Cadence Incisive” on page 1-21. If you start from within MATLAB, you must use `hdlsimmatlab`.).

Examples

The following function call displays a message in the HDL simulator command window using port 5678 on hostname msuser:

```
tclHdlSim('puts "Done"', '5678', 'msuser')
```

5

HDL Simulator Tcl Commands — Alphabetical List

hdlsimmatlab

Purpose

Load instantiated HDL design for verification with MATLAB

Syntax

```
hdlsimmatlab <instance> [<ncsim_args>]
```

Arguments

`<instance>`
: Specifies the instance of an HDL design to load for verification.

`<ncsim_args>`
: Specifies one or more `ncsim` command arguments. For details, see the description of `ncsim` in the Incisive simulator documentation.

Description

The `hdlsimmatlab` command loads the specified instance of an HDL design for verification and sets up the Incisive simulator so it can establish a communication link with MATLAB. The Incisive simulator opens a simulation workspace as it loads the HDL design.

This command may be run from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

Examples

The following command loads the module instance `parse` from library `work` for verification and sets up the Incisive simulator so it can establish a communication link with MATLAB:

```
tclshell> hdlsimmatlab work.parse
```

Purpose

Load instantiated HDL design for cosimulation with Simulink

Syntax

```
hdlsimulink [<ncsim_args>] <instance>
[-socket <tcp_spec>]
```

Argument

`<ncsim_args>`
: Specifies one or more `ncsim` command arguments. At a minimum, either `-gui` or `-tcl` is required. If you specify `-gui`, the Simulink GUI will be launched when the HDL design is loaded. If you specify `-tcl`, a Tcl script shell is launched instead. If you do not specify either of these arguments, the HDL simulator runs the simulation without Simulink. Other valid `ncsim` arguments may be specified in addition to `-gui` or `-tcl`. For more information on `-gui`, `-tcl`, or other `ncsim` arguments, see the description of `ncsim` in the Incisive simulator documentation.

`<instance>`
: Specifies the instance of an HDL design to load for cosimulation.

`-socket <tcp_spec>`
: Specifies TCP/IP socket communication for the link between the Incisive simulator and MATLAB. This setting overrides the setting specified with the MATLAB `nclaunch` function. The `<tcp_spec>` can consist of a TCP/IP socket port number or service name (alias). For example, you might specify port number 4449 or the service name `matlabservice`.

  For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page D-2.

  If the Incisive 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 `hdlsimulink` command must match what you specify for the communication mode when you configure Link for Cadence Incisive blocks in your Simulink model.

For more information on modes of communication, see "Modes of Communication" on page 1-7. For more information on establishing the Simulink end of the communication link, see "Communicating Between Cadence Incisive and Simulink" on page 3-12.

Description

The `hdlsimulink` command loads the specified instance of an HDL design for cosimulation and sets up the Incisive simulator so it can establish a communication link with Simulink. The Incisive simulator opens a simulation workspace into which it loads the HDL design.

Examples

The following command loads the module instance `parse` from library `work` for cosimulation, sets up the Incisive simulator so it can establish a communication link with Simulink, and opens a Tcl script shell:

```
tclshell> hdlsimulink -gui work.parse
```

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`.

Do not specify an instance of an HDL design that has already been associated with a MATLAB test bench function (via `matlabtb`).

`<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.
`-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 initially 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 initially 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.

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the Incisive 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`). 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`

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

If the Incisive 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 "Modes of Communication" on page 1-7. For more information on establishing the MATLAB end of the communication link, see "Starting the MATLAB Server" on page 2-47.

`-rising <signal>[, <signal>...]`
: Indicates that the specified MATLAB function is called on the rising edge (transition from `'0'` to `'1'`) of any of the specified

signals. Specify `-rising` with the pathnames 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 pathnames 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 pathnames 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 attached to the HDL instance you specify for `instance`. If you omit this argument, `matlabcp` attaches the HDL instance to a MATLAB function that has the same name as the HDL instance. For example, if the HDL instance is `myfirfilter`, `matlabcp` associates the HDL 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 Incisive simulator client component of the Link for Cadence Incisive software.
- Associates a specified instance of an HDL design created in the Incisive 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.

Note For the Incisive 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.

MATLAB component functions simulate the behavior of the HDL model. A stub entity or module (providing port definitions only) in the HDL design 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 entity or module. A MATLAB component typically provides some functionality (such as a filter) that is not yet implemented in the HDL code. See “Coding MATLAB Link Functions” on page 2-11.

Examples

The following command starts the Incisive simulator client component of the Link for Cadence Incisive software. The `'-mfunc'` option specifies the m-function to connect to and `'-socket'` option specifies the port number for socket connection mode.

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

Purpose

Initiate MATLAB test bench session for instantiated HDL design

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 design that attaches to a MATLAB test bench function. By default, `matlabtb` attaches the instance to 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 Do not specify an instance of an HDL design that has already been associated with a MATLAB component function (via `matlabcp`). 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.
`-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 initially 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 initially 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.

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the Incisive 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`). 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`

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

If the Incisive 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 "Modes of Communication" on page 1-7. For more information on establishing the MATLAB end of the communication link, see "Starting the MATLAB Server" on page 2-47.

`-rising <signal>[, <signal>...]`
: Indicates that the specified MATLAB function is called on the rising edge (transition from `'0'` to `'1'`) of any of the specified

signals. Specify `-rising` with the pathnames 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 pathnames 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 pathnames 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 MATLAB function that is attached to the HDL instance you specify for `instance`. If you omit this argument, `matlabtb` attaches the HDL instance to a MATLAB function that has the same name as the HDL instance. For example, if the HDL instance is `myfirfilter`, `matlabtb` associates the HDL 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 Incisive simulator client component of the Link for Cadence Incisive software.
- Associates a specified instance of an HDL design created in the Incisive 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.

Note For the Incisive 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 starts the Incisive simulator client component of the Link for Cadence Incisive software, associates an instance of the module `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:

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

The following command starts the Incisive simulator client component of the Link for Cadence Incisive software, and initiates a remote TCP/IP socket-based session using remote MATLAB host `compa` and TCP/IP port 4449. Based on the specified test bench stimuli, `myfirfilter.m` executes 10 nanoseconds from the current time, each time signal `work.fclk` experiences a rising edge, and each time signal `work.din` changes state.

```
ncsim> matlabtb myfirfilter 10 ns -rising top.fclk
-sensitivity top.din -socket 4449@compa
```

The following command starts the Incisive simulator client component of the Link for Cadence Incisive software. The `'-mfunc'` option specifies the m-function to connect to and `'-socket'` option specifies the port number for socket connection mode. `'-sensitivity'` indicates that the test bench session is sensitized to the signal `sine_out`.

```
ncsim>matlabtb osc_top -sensitivity osc_top.sine_out
      -socket 4448 -mfunc hosctb
```

Purpose

Call specified MATLAB function for immediate execution on behalf of instantiated HDL design

Syntax

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

Arguments

`<instance>`

Specifies the instance of an HDL design that attaches to a MATLAB function. By default, `matlabtbeval` attaches the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabtbeval` associates the 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 Incisive 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. The following table lists different ways of specifying `<tcp_spec>`.

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

For more information on choosing TCP/IP socket ports, see “Choosing TCP/IP Socket Ports” on page D-2.

If the Incisive 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 modes of communication, see "Modes of Communication" on page 1-7. For more information on establishing the MATLAB end of the communication link, see "Starting the MATLAB Server" on page 2-47.

`-mfunc <name>`

The name of the MATLAB function that is attached to the HDL instance you specify for `instance`. If you omit this argument, `matlabtbeval` attaches the HDL instance to a MATLAB function that has the same name as the HDL instance. For example, if the HDL instance is `myfirfilter`, `matlabtbeval` associates the HDL 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 Incisive simulator client component of the Link for Cadence Incisive software.
- Associates a specified instance of an HDL design created in the Incisive simulator with a MATLAB function.

- Executes the specified MATLAB function once and immediately on behalf of the specified module instance.

Note For the Incisive 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

The following command starts the Incisive simulator client component of the Link for Cadence Incisive 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`:

```
ncsim> matlabtbeval myfirfilter -socket 4449
```

The following command starts the Incisive simulator client component of the Link for Cadence Incisive software, associates an instance of the module `filter` with the function `myfirfilter.m`, and uses a remote TCP/IP socket-based communication link to host `compa` and TCP/IP port 4449 to execute the function `myfirfilter.m`

```
ncsim> matlabtbeval myfirfilter -socket 4449@compa
```

nomatlabtb

Purpose

Terminate active MATLAB test bench and MATLAB component sessions

Syntax

```
nomatlabtb
```

Description

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

Examples

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

```
ncsim> nomatlabtb
```

See Also

`matlabcp`, `matlabtb`

6

Simulink Blocks — Alphabetical List

HDL Cosimulation

Purpose

Cosimulate a hardware component by communicating with an HDL model executing in Incisive simulator

Library

Link for Cadence Incisive

Description

HDL Cosimulation

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 Incisive 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 model. 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 Incisive simulator.
- Rising-edge or falling-edge clocks to apply to your model. You can specify the period for each clock signal.
- Tcl commands to run before and after the simulation.

The **Ports** pane provides fields for mapping signals of your HDL design to input and output ports in your block. The signals can be at any level of the HDL design hierarchy. Simulink deposits an input port signal on an Incisive simulator signal at the signal's sample rate. Conversely, Simulink reads an output port signal from a specified Incisive 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 The **Data Type** and **Fraction Length** properties apply only to

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

Input/output ports can be used here as well; specify port as both input and output.

The **Timescales** pane lets you choose an optimal timing relationship between Simulink and the Incisive simulator. You can configure either a *relative* timing relationship (Simulink seconds correspond to an Incisive simulator-defined tick interval) or an *absolute* timing relationship (Simulink seconds correspond to an absolute unit of Incisive simulator time).

The **Connection** pane specifies the communications mode used between Simulink and the Incisive 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 Incisive simulator. The **Connection** pane also provides the option for bypassing the cosimulation block during Simulink simulation.

The **Clocks** pane lets you create optional rising-edge and falling-edge clocks that apply stimuli to your cosimulation model. You can either specify an explicit period for each clock, or accept a default period of `2`. Simulink attempts to create a clock that has a 50% duty cycle and a predefined phase that is inverted for the falling edge case.

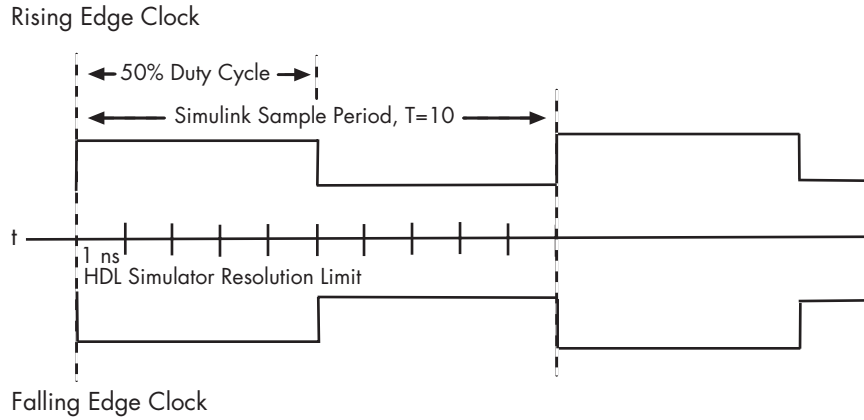
Whether you have configured the **Timescales** pane for relative timing mode or absolute timing mode, the following restrictions apply to clock periods:

- If you specify an explicit clock period, you must enter a sample time equal to or greater than 2 resolution units (ticks).
- If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle, and therefore the Link for Cadence Incisive interface creates the falling edge at

  clockperiod / 2

  (rounded down to the nearest integer).

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



For more information on calculating relative and absolute timing modes, see “Defining the Simulink and Cadence Incisive Timing Relationship” on page 3-16 .

The **Tcl** pane provides a way of specifying tools command language (Tcl) commands to be executed before and after the Incisive simulator simulates the HDL component of your Simulink model. The **Pre–simulation commands** field on this pane is particularly useful for simulation initialization and startup operations, but it cannot be used to change simulation state.

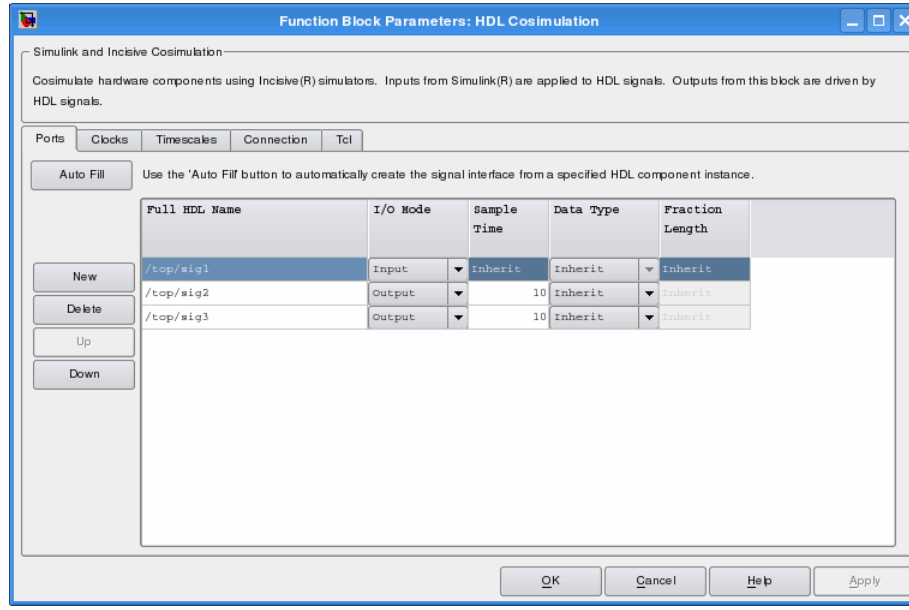
Note You must make sure that signals being used in cosimulation have read/write access (this is done through the HDL simulator – see product documentation for details). This rule applies to all signals on the **Ports**, **Clocks**, and **Tcl** panes.

Dialog Box

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

- “Ports Pane” on page 6-6
- “Connection Pane” on page 6-10

Ports Pane



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 Incisive simulator. The port information request returns port names and information from an HDL model under simulation in the Incisive simulator, and automatically enters this information into the ports list. See “Obtaining Signal Information Automatically from Cadence Incisive” on page 3-37 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**.

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 pathname, using the Incisive simulator pathname syntax. For example, a pathname 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 3-31.

Note You can copy signal pathnames directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field, using the standard copy and paste commands in the Incisive simulator and Simulink (as long as you use the 'Path.Name' view and not 'Db::Path.Name' view). After pasting a signal pathname into the **Full HDL Name** field, you must click the **Update** button to complete the paste operation and update the signal list.

I/O Mode

Select either `Input`, `Output`, or both.

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

To specify `Inout` ports of your HDL model, specify one entry for the signal in the Ports Pane as an input and another entry as an output.

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 6-14). See also "Representation of Simulation Time" on page 3-15.

Data Type
Fraction Length

These two related parameters apply only to output signals.

The **Data Type** property is enabled only for output signals. You can direct Simulink to determine the data type, or you can assign

an explicit data type (with option fraction length). By explicitly assigning a data type, you can force fixed point data types on output ports of an HDL Cosimulation block.

The **Fraction Length** property specifies the size, in bits, of the fractional part of the signal in fixed-point representation. **Fraction Length** 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

- 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 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 Incisive simulator for the data type of the port. As an example, if the Incisive

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**.

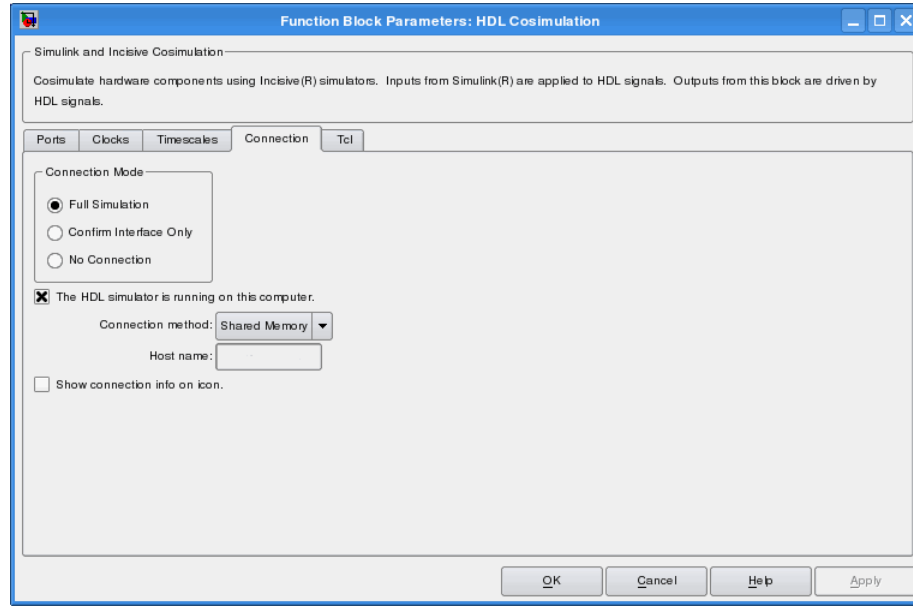
  For example, if you specify **Data Type** as `Signed` and a **Fraction Length** of `5` for a 16-bit signal, Simulink forces the data type to `sfix16_En5`. For the same signal with a **Data Type** set to `Signed` and **Fraction Length** of `-5`, Simulink forces the data type to `sfix16_E5`.

- 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**.

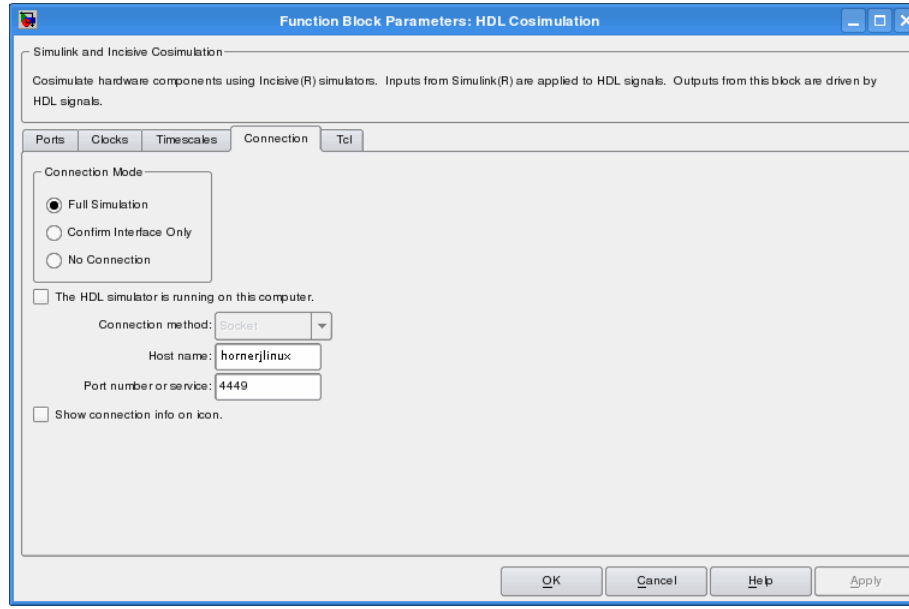
  For example, if you specify **Data Type** as `Unsigned` and a **Fraction Length** of `5` for a 16-bit signal, Simulink forces the data type to `ufix16_En5`. For the same signal with a **Data Type** set to `Unsigned` and **Fraction Length** of `-5` , Simulink forces the data type to `ufix16_E5`.

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 Incisive 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.



the HDL Simulator is running on this computer

Select this option if you want to run Simulink and the Incisive 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 Incisive simulator to communicate via a designated TCP/IP socket. Select `Shared memory` if you want Simulink and the Incisive simulator to communicate via shared memory. For more information on these connection methods, see “Modes of Communication” on page 1-7.

Host name

If Simulink and the Incisive 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 Incisive 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 D-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 Incisive simulator in different modes, this information helps to distinguish between different cosimulations.

Connection Mode

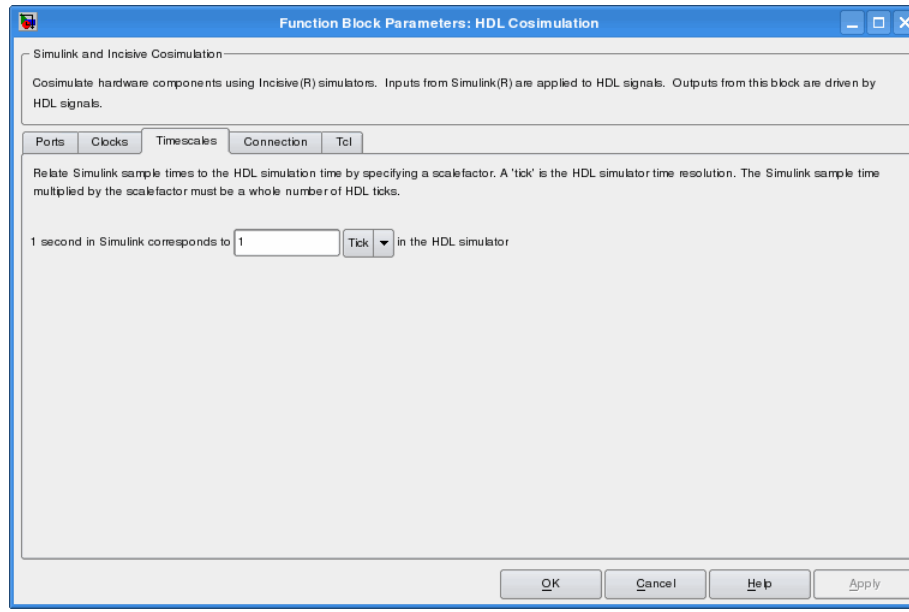
If you want to bypass the HDL simulator when running a Simulink simulation, use these options to specify what type of simulation connection you want. Select one of the following:

- **Full Simulation**: Confirm interface and run HDL simulation (default).
- **Confirm Interface Only**: 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, the Link for Cadence Incisive interface does not communicate with the HDL simulator during Simulink simulation.

Timescales Pane

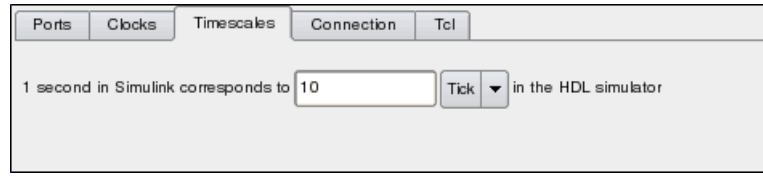
The **Timescales** pane of the HDL Cosimulation block parameters dialog lets you choose an optimal timing relationship between Simulink and the Incisive simulator. The following figure shows the default settings of the **Timescales** pane.



The **Timescales** pane specifies a correspondence between one second of Simulink time and some quantity of Incisive simulator time. This quantity of Incisive simulator time can be expressed in one of the following ways:

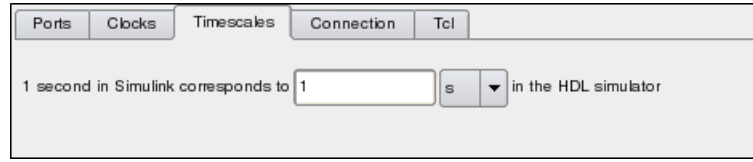
- In *relative* terms (i.e., as some number of Incisive simulator ticks). In this case, the cosimulation is said to operate in *relative timing mode*. Relative timing mode is the default.

  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` Incisive 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` Incisive simulator second to `1` Simulink second.



  To set the absolute time, you must know the value of the HDL simulator tick (resolution unit) to understand how Link for Cadence Incisive software handles the timing of the falling edge when the duty cycle does not fall at 50%. The following restrictions apply to clock periods:

  - You must enter a sample time equal to or greater than 2 resolution units (ticks) (no falling edge can occur in < 2 ticks).

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

  clockperiod / 2

  (rounded down to the nearest integer).

  You must know how many ticks your selected time represents so that you know how the falling edge will occur. This next example demonstrates how to calculate the number of HDL simulator ticks for an absolute clock period of 1 Simulink second = 3 HDL simulator seconds.

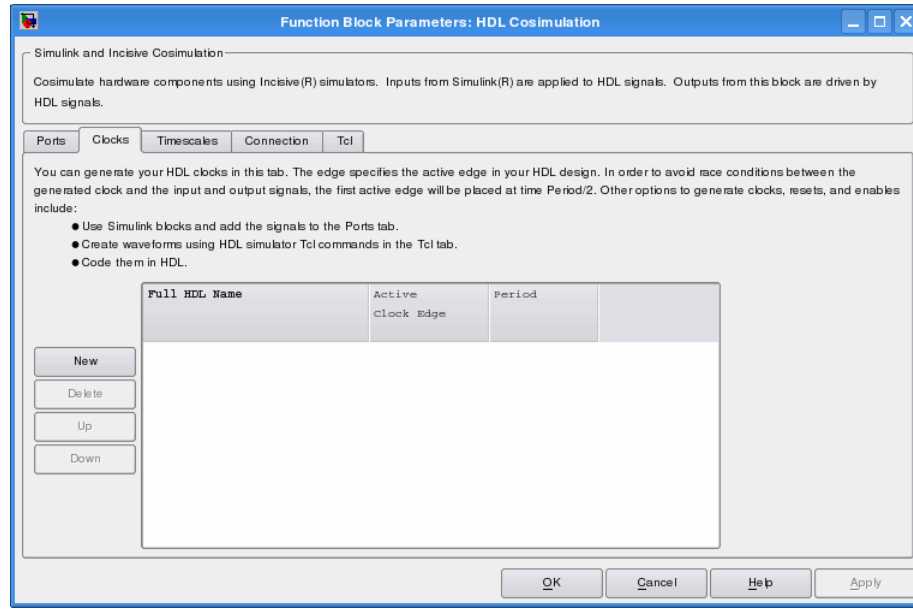
  ```
  1 HDL simulator second = 10^9 HDL simulator ns
  1 HDL simulator tick = 10 HDL simulator ns
  1 HDL simulator second = (10^9/10) or 10^8 HDL simulator ticks

  1 Simulink seconds = 3 HDL simulator seconds
  1 Simulink second = 3x10^8 HDL simulator ticks
  ```

  In this example, the number of ticks is greater than 2 and an even integer multiple, therefore the duty cycle will fall at 50%. If 1 HDL simulator tick was instead equal to 13 ns, the end result would have the falling edge occur at 1153846153 ticks, or a just under 50% duty cycle.

For more information on calculating relative and absolute timing modes, see “Defining the Simulink and Cadence Incisive Timing Relationship” on page 3-16.

For detailed information on the relationship between Simulink and the Incisive simulator during cosimulation, and on the operation of relative and absolute timing modes, see “Representation of Simulation Time” on page 3-15.

Clocks Pane



The scrolling list at the center of the pane displays HDL clocks that drive values to the HDL signals that you are modeling, using the deposit method.

Maintain the list of clock signals with the buttons on the left of the pane:

- **New** — Add a new clock signal to the list and select it for editing.
- **Delete** — Remove a clock signal from the list.
- **Up** — Move the selected clock signal up one position in the list.
- **Down** — Move the selected clock signal down one position in the list.

To commit edits to the Simulink model, you must also click **Apply**

To edit the name of a clock signal, double-click it and enter the correct name. To edit the properties of a clock signal, select the appropriate property in that signal row. The properties of a clock signal are

Full HDL Name

Specify each clock as a signal pathname, using the Incisive simulator pathname syntax. A sample pathname for a clock might be `manchester.clk`.

For information about and requirements for path specifications in Simulink, see "Specifying HDL Signal/Port and Module Paths for Cosimulation" on page 3-31.

Note You can copy signal pathnames directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field, using the standard copy and paste commands in the Incisive simulator and Simulink (as long as you use the 'Path.Name' view and not 'Db::Path.Name' view). After pasting a signal pathname into the **Full HDL Name** field, you must click the **Update** button to complete the paste operation and update the signal list.

Edge

Select `Rising` or `Falling` to specify either a rising-edge clock or a falling-edge clock.

Period

You must either specify the clock period explicitly, or accept the default period of `2`.

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

If the clock period (whether explicitly specified or defaulted) is not an even integer, Simulink cannot create a 50% duty cycle,

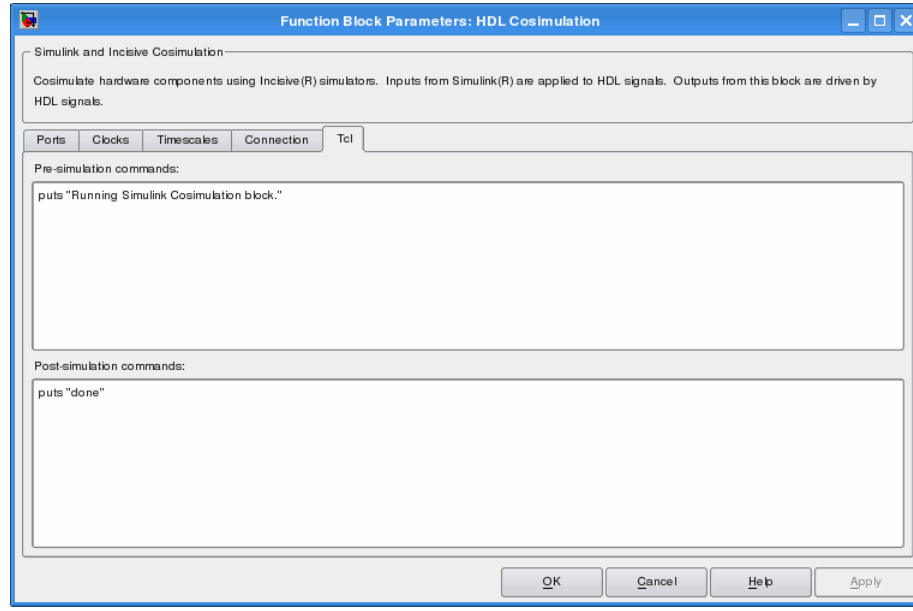
and therefore the Link for Cadence Incisive software creates the falling edge at

clockperiod / 2

(rounded down to the nearest integer).

Note Vectored signals in the **Clocks** pane are not supported. Signals must be logic types with '1' and '0' values.

Tcl Pane



Pre-simulation commands

Tcl commands to be executed before the Incisive simulator simulates the HDL component of your Simulink model. You can specify one Tcl command per line in the text box, or enter multiple

commands per line by appending each command with a semicolon (;), the standard Tcl concatenation operator.

Alternatively, you can create an Incisive simulator Tcl script that lists Tcl commands and then specify that file with the Incisive simulator `source` command as follows:

```
source mycosimstartup.script_extension
```

Use of this field can range from something as simple as a one-line echo command to confirm that a simulation is running to a complex script that performs an extensive simulation initialization and startup sequence.

Note The command string or Tcl script that you specify for this parameter cannot include commands that load an Incisive simulator project or modify simulator state. For example, they cannot include commands such as `run`, `stop`, or `reset`.

Post-simulation commands

Tcl commands to be executed after the Incisive simulator simulates the HDL component of your Simulink model. You can specify one Tcl command per line in the text box or enter multiple commands per line by appending each command with a semicolon (;), the standard Tcl concatenation operator.

Alternatively, you can create an Incisive simulator Tcl script that lists Tcl commands and then specify that file with the Incisive simulator `source` command as follows:

```
source mycosimcleanup.script_extension
```

Notes

- You can include the `exit` command in an after simulation Tcl script to force the Incisive simulator to shut down at the end of a cosimulation session. To ensure that all other after simulation Tcl commands specified for the model have an opportunity to execute, specify all after simulation Tcl commands in a single cosimulation block and place `exit` at the end of the command string or Tcl script.

  The following is an example of a Tcl script when the `-gui` argument was used with `hdlsimmatlab` or `hdlsimulink`:

  ```
  after 1000 {ncsim -submit exit}
  ```

  This next example is of a Tcl exit script to use when the -tcl argument was used with `hdlsimmatlab` or `hdlsimulink`:

  ```
  after 1000 {exit}
  ```

- With the exception of `exit`, the command string or Tcl script that you specify cannot include commands that load an Incisive simulator project or modify simulator state. For example, they cannot include commands such as `run`, `stop`, or `reset`.

To VCD File

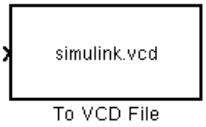
Purpose

Generate a value change dump (VCD) file

Library

Link for Cadence Incisive

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

- 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:

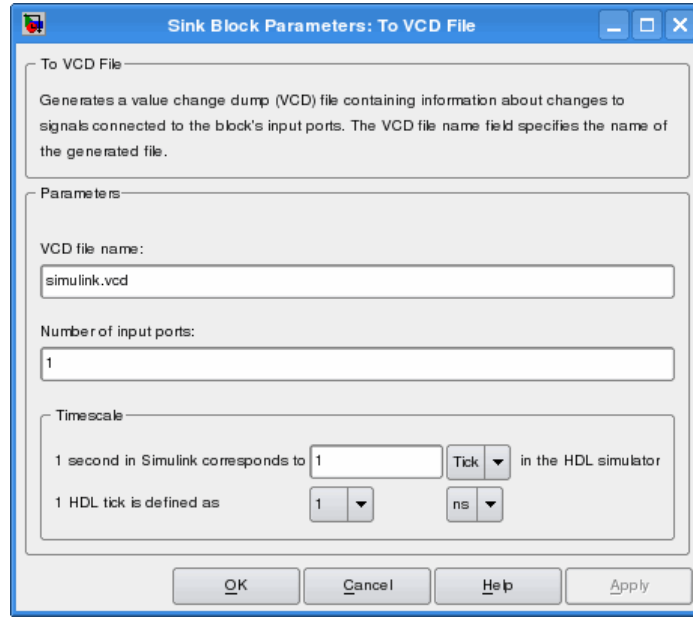
- The file name to be used for the generated file
- The number of block input ports that are to receive signal data

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). Each bit maps to one symbol.

For a description of the VCD file format, see "VCD File Format" on page 6-25.

Note The toVCD block is integrated into the Simulink Signal & Scope Manager. See the Simulink User's Guide, "Signal & Scope Manager" 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 pathname 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.

If you want the generated file to have a `.vcd` file type extension, you must specify it explicitly.

Caution 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.

Note The To VCD File block does not support floating point signal types.

Note 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 Incisive 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 Link for Cadence Incisive version` `1.0 $ end`	Version of the VCD block that generated the file.
`$timescale 1 ns $ end`	The time scale that was used during the simulation.
`$scope module manchestermodel $end`	The scope of the module being dumped.

Generated VCD File Format (Continued)

File Content	Description

`$var wire 1 ! Original Data [0] $end`
`$var wire 1 " Recovered Clock [0] $end`
`$var wire 1 # Recovered Data [0] $end`
`$var wire 1 $ Data Validity [0] $end`
	Variable definitions. Each definition associates a signal with character identification code (symbol). The symbols are derived from printable characters in the ASCII character set from ! to ~. Variable definitions also include the variable type (wire) and size in bits.

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

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

Generated VCD File Format (Continued)

File Content	Description
`#630` `1!`	The starting point of logged value changes. 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.

`.`
`.`
`.`
`#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`.

VCD files can grow very large for larger designs or smaller designs with longer simulation runs. 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).

To VCD File

A

Examples

Use this list to find examples in the documentation.

Starting the HDL Simulator Using the MATLAB Function nclaunch

"nclaunch Examples" on page 1-21

Coding MATLAB and Cadence Incisive Applications

"Sample MATLAB Test Bench Function" on page 2-28

B

VHDL and Verilog Language Support

VHDL and Verilog Language Support

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

Link for Cadence Incisive software also supports mixed-language HDL models (models with both Verilog and VHDL components), allowing you to cosimulate VHDL and Verilog signals simultaneously. However, only Simulink can access components in different languages at any level; MATLAB can access signals only with the language of the top-level module instance or component. .

Mixed-Language Model Limitation

The Cadence VHPI reports the incorrect simulator precision when simulating mixed Verilog/VHDL design. (It is correct when in a pure VHDL design.) In a mixed-HDL model, the VHPI always returns a precision of 1 fs. The actual simulator precision is properly modified by `-vhdl_time_precision`, but the returned value does not reflect that value in a mixed-HDL model.

You will get incorrect or non-running simulations if *both* the following conditions exist:

- You have a mixture of VHDL and Verilog in your design AND you have set `-vhdl_time_precision` to TP and TP != 1fs
- You also are cosimulating *either* of the following:
  - Only VHDL signals and there is a Simulink sample time finer than TP (after accounting for the cosimulation block timescale calculations)
  - Both VHDL and Verilog signals and TP is coarser than the Verilog time precision set by timescale or the -timescale command line and there is a Simulink sample time finer than TP (after accounting for the cosimulation block timescale calculations)

C

Link for Cadence Incisive Machine Configuration Requirements

Configuration Restrictions Using Link for Cadence Incisive with MATLAB

The following list provides samples of valid configurations for using the Cadence HDL simulator with MATLAB . The scenarios apply whether the HDL simulator is running on the same or different computing system as MATLAB. 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 Link for Cadence Incisive 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.

Configuration Restrictions Using Link for Cadence Incisive with Simulink

The following list provides samples of valid configurations for using the Cadence HDL simulator with Simulink. The scenarios apply whether the HDL simulator is running on the same or different computing system as MATLAB or Simulink. 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.

D

TCP/IP Socket Communication

Choosing TCP/IP Socket Ports

Depending on your particular configuration (for example, when MATLAB and the HDL simulator reside on separate machines), when creating a Link for Cadence Incisive 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 Link for Cadence Incisive 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 a Link for Cadence Incisive 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 Simulink 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 Link for Cadence Incisive 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 Link for Cadence Incisive to fail to make a connection.

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

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

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

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

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

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

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 Simulink as you cannot really automate the starting of an HDL simulator service, you might want to use a service with Simulink to reserve a TCP/IP socket port.

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

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

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z