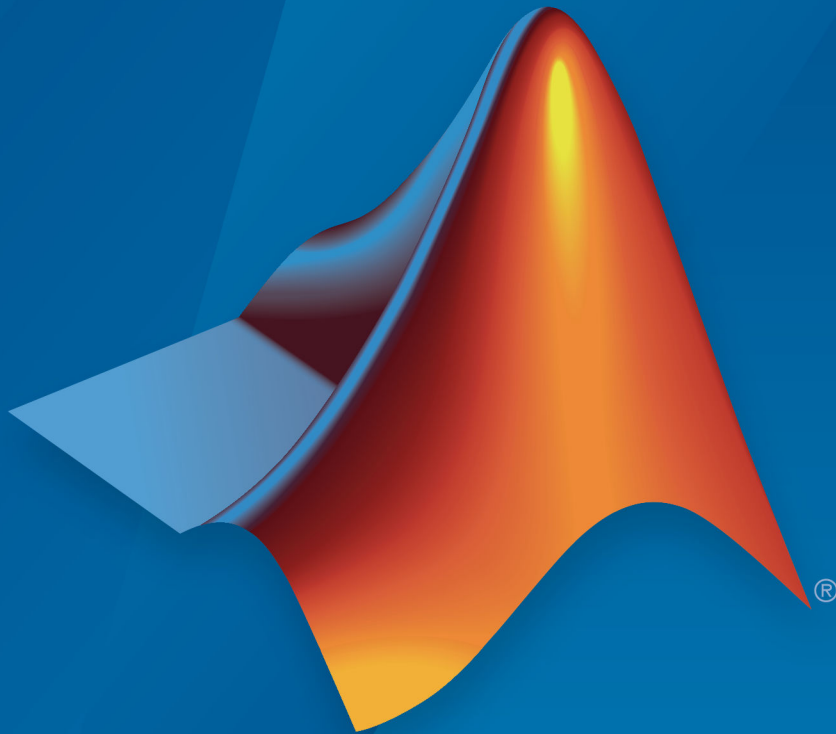


HDL Verifier™

Reference



MATLAB® & SIMULINK®

R2017b



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *HDL Verifier™ Reference*

© COPYRIGHT 2003–2017 by The MathWorks, Inc.

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

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

### **Trademarks**

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

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

|                |             |   |
|----------------|-------------|---|
| August 2003    | Online only | New for Version 1 (Release 13SP1)         |
| February 2004  | Online only | Revised for Version 1.1 (Release 13SP1)   |
| June 2004      | Online only | Revised for Version 1.1.1 (Release 14)    |
| October 2004   | Online only | Revised for Version 1.2 (Release 14SP1)   |
| December 2004  | Online only | Revised for Version 1.3 (Release 14SP1+)  |
| March 2005     | Online only | Revised for Version 1.3.1 (Release 14SP2) |
| September 2005 | Online only | Revised for Version 1.4 (Release 14SP3)   |
| March 2006     | Online only | Revised for Version 2.0 (Release 2006a)   |
| September 2006 | Online only | Revised for Version 2.1 (Release 2006b)   |
| March 2007     | Online only | Revised for Version 2.2 (Release 2007a)   |
| September 2007 | Online only | Revised for Version 2.3 (Release 2007b)   |
| March 2008     | Online only | Revised for Version 2.4 (Release 2008a)   |
| October 2008   | Online only | Revised for Version 2.5 (Release 2008b)   |
| March 2009     | Online only | Revised for Version 2.6 (Release 2009a)   |
| September 2009 | Online only | Revised for Version 3.0 (Release 2009b)   |
| March 2010     | Online only | Revised for Version 3.1 (Release 2010a)   |
| September 2010 | Online only | Revised for Version 3.2 (Release 2010b)   |
| April 2011     | Online only | Revised for Version 3.3 (Release 2011a)   |
| September 2011 | Online only | Revised for Version 3.4 (Release 2011b)   |
| March 2012     | Online only | Revised for Version 4.0 (Release 2012a)   |
| September 2012 | Online only | Revised for Version 4.1 (Release 2012b)   |
| March 2013     | Online only | Revised for Version 4.2 (Release 2013a)   |
| September 2013 | Online only | Revised for Version 4.3 (Release 2013b)   |
| March 2014     | Online only | Revised for Version 4.4 (Release 2014a)   |
| October 2014   | Online only | Revised for Version 4.5 (Release 2014b)   |
| March 2015     | Online only | Revised for Version 4.6 (Release 2015a)   |
| September 2015 | Online only | Revised for Version 4.7 (Release 2015b)   |
| March 2016     | Online only | Revised for Version 5.0 (Release 2016a)   |
| September 2016 | Online only | Revised for Version 5.1 (Release 2016b)   |
| March 2017     | Online only | Revised for Version 5.2 (Release 2017a)   |
| September 2017 | Online only | Revised for Version 5.3 (Release 2017b)   |



|          |   |
|----------|---|
| <b>1</b> | <b>Blocks — Alphabetical List</b>         |
| <b>2</b> | <b>System Objects — Alphabetical List</b> |
| <b>3</b> | <b>Functions — Alphabetical List</b>      |



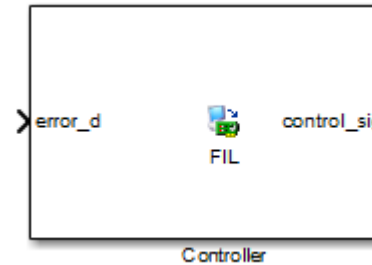
# Blocks — Alphabetical List

---

## FIL Simulation

Simulate HDL code on FPGA hardware from Simulink

**Library:** generated



## Description

The generated FPGA-in-the-loop (FIL) simulation block is the communication interface between the FPGA and your Simulink® model. It integrates the hardware into the simulation loop and allows it to participate in simulation as any other block.

You can generate a FIL Simulation block from existing HDL code using the **FPGA-in-the-Loop Wizard**, or, generate HDL code and an accompanying FIL Simulation block using HDL Workflow Advisor. Generating HDL code requires an HDL Coder™ license.

For the generation and simulation workflow, see “Block Generation with the FIL Wizard”. If you encounter any issues during FIL simulation, refer to “Troubleshooting FIL” for help in diagnosing the problem.

You can use the FIL Simulation block in models running in Normal, Accelerator, or Rapid Accelerator simulation modes. The FIL Simulation parameters are not tunable in any of the simulation modes. For more information about these modes, see “How Acceleration Modes Work” (Simulink).



## Ports

The ports of the block correspond to the interface of your HDL design running on your FPGA. You can configure the data types of the signals that the FIL Simulation block returns to Simulink.

## Input

**HDL\_input\_port\_name** — Signal passed from Simulink to FPGA

scalar | vector

The ports on the block correspond with ports on your HDL design. You can configure the **Sample time** and **Data type**

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 |  
Boolean | Fixed-point

## Output

**HDL\_output\_port\_name** — Signal passed from the FPGA to Simulink

scalar | vector

The ports on the block correspond with ports on your HDL design. You can configure the **Sample time** and **Data type**

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 |  
Boolean | Fixed-point

## Parameters

The parameters displayed in the **Hardware Information** section reflect your selections when you generated the FIL Simulation block from a subsystem. These parameters are informational only.

- **Connection:** Either Ethernet or PCI Express®. Some boards can use only one connection type or the other; with other boards, you may have the option of using either connection. You configure the **MAC address** and **IP address** of the board when you generate the block.

- **Board:** The make and model of FPGA board. For supported boards, see “Supported FPGA Devices for FIL Simulation”.
- **FPGA part:** Chip identification number.
- **FPGA project file:** The location of the FPGA project file generated for your design.

To download the generated FPGA programming file onto the FPGA, set the parameters in **FPGA Programming File**. This step is required before you can run a FIL simulation. See “Load Programming File onto FPGA”.

To configure data rate parameters, set options in the **Runtime Options** group.

On the **Signal Attributes** pane, you can configure **Sample time** and **Data type** for each output port. The direction and bit width of the signals, and the sample time and data type of the input ports, are informational only.

### FPGA Programming File

#### **File name — Location of programming file**

string

Location of the FPGA programming file generated for your design. To load this design to the FPGA for simulation, click **Load**.

### Runtime Options

#### **Overclocking factor — FPGA sample rate relative to Simulink clock**

1 (default) | integer

Ratio of FPGA clock rate to the Simulink clock rate. The FPGA clock samples inputs to the FPGA this many times for each Simulink timestep.

#### **Output frame size — Amount of data returned to Simulink**

Inherit: auto (default)

Output signals are returned as **Output frame size**-by-1 column vectors. Increasing the frame size can speed up your simulation by reducing the communication time between Simulink and the FPGA board.

Note these limitations on the frame size :

- The input frame size must be an integer multiple of the output frame size.

- The output frame size must be less than the input frame size.
- The input frame size and output frame size cannot vary during simulation.

### Signal Attributes

#### **Sample Time** — Sample time of each port

`Inherit: Inherit via internal rule` (default)

Explicitly set sample times for the output signals, or use `Inherit: Inherit via internal rule`. The internal rule is to set the output sample times to the input base sample time divided by the scaling factor.

#### **Data type** — Data type of each port

`fixdt(0,N,0)` (default) | data type expression

How Simulink interprets the bits in the output signal from the FPGA. You can explicitly set output data types, use the default unscaled and unsigned type, or specify `Inherit: auto` to inherit a data type from context.

## Model Examples

## See Also

### Topics

“FPGA-in-the-Loop Simulation”

“FPGA-in-the-Loop Simulation Workflows”

“FIL Simulation with HDL Workflow Advisor for Simulink”

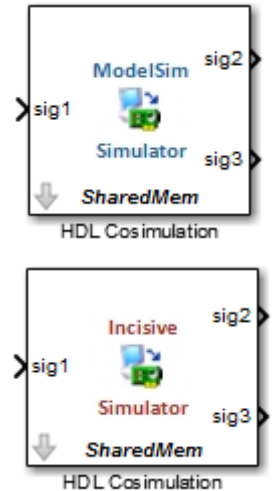
“Block Generation with the FIL Wizard”

**Introduced in R2012b**

## HDL Cosimulation

Cosimulate HDL design by connecting Simulink with HDL simulator

**Library:** HDL Verifier / For Use with Mentor Graphics  
Modelsim  
HDL Verifier / For Use with Cadence Incisive



## Description

The HDL Cosimulation block cosimulates a hardware component by applying input signals to and reading output signals from an HDL model under simulation in the HDL simulator. You can use this block to model a source or sink device by configuring the block with input or output ports only.

You can configure these options on the block:

- Mapping of the input and output ports of the block to correspond with signals (including internal signals) of an HDL module. You must specify a sample time for each output port. You can optionally specify a data type for each output port.
- Type of communication and communication settings used to exchange data between simulators.
- The timing relationship between units of simulation time in Simulink and the HDL simulator.

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

---

### Compatibility with Simulink Code Generation

- This block participates in HDL code generation with HDL Coder. The coder generates an interface to your manually written or legacy HDL code. It does not participate in C code generation with Simulink Coder™.
- 

## Ports

The ports shown on the block correspond with signals from your HDL design running in the HDL simulator. You can add and remove ports, and configure their data types and sample times, by changing the block parameters. The **Ports** tab displays the HDL signals that correspond to the ports. You can add, remove, and change the order of the ports. Use the **Auto Fill** button to fill the table via a port information request to the HDL simulator. This request returns port names and information from your HDL design running in the HDL simulator. See “Get Signal Information from HDL Simulator” for a detailed description of this feature.

All signals that you specify when you configure the HDL Cosimulation block must have read/write access in the HDL simulator. Refer to the HDL simulator product documentation for details.

When you import VHDL® signals from the HDL simulator, HDL Verifier returns the signal names in all capitals.

## Input

**HDL\_input\_port\_name** — Signal passed from Simulink to HDL simulator

scalar | vector

The ports on the block correspond with ports on your HDL design. Add or remove ports on the **Ports** tab.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

## Output

**HDL\_output\_port\_name** — Signal passed from HDL simulator to Simulink

scalar | vector

The ports on the block correspond with ports on your HDL design. Add or remove ports on the **Ports** tab.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `Fixed-point`

## Parameters

### Ports

**Enable direct feedthrough** — Work around algebraic loop warnings

`true` (default) | `false`

Eliminates the one output-sample delay difference between the cosimulation and Simulink that occurs when your model contains purely combinational paths. Clear this check box if the HDL Cosimulation block is in a feedback loop and generates algebraic loop warnings or errors. When you simulate a sequential circuit that has a register on the data path, specifying direct feedthrough does not affect the timing of that data path.

**Full HDL Name** — Signal path name

string

Specify the signal path name using the HDL simulator path name syntax. For example, `manchester.samp` for Incisive@HDL simulators. The signal can be at any level of the HDL design hierarchy. The HDL Cosimulation block port corresponding to the signal is labeled with this name.

For rules on specifying port and module path names in Simulink, see “Specify HDL Signal/Port and Module Paths for Cosimulation”.

You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field. Use the `Path.Name` view and not `Db::Path.Name` view. After pasting a signal path name into **Full HDL Name**, click **Apply** to complete the paste operation and update the signal list.

**I/O Mode — Port direction**

Input | Output

To add a bidirectional port, add the port to the list twice, as both input and output.

**Input** — HDL signals that Simulink drives. Simulink deposits values on the specified HDL simulator signal at the specified 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, avoid defining an input port that has multiple instances. If multiple sources drive input to a single signal, your simulation model produces unexpected results.

---

**Output** — HDL signals that Simulink reads. For output signals, you must specify an explicit sample time. You can also specify the data type, but the width must match the width of the signal in HDL. For details on specifying a data type, see the **Data Type** and **Fraction Length** parameters.

Simulink signals do not have a tristate semantic because there is no 'z' value. To interface with bidirectional signals, connect to the input and enable signals of both the output driver and the output signal of the input driver. This approach leaves the actual tristate buffer in HDL, where resolution functions can handle interfacing with other tristate buffers.

**Sample Time — Time between reading samples on an output port**

1 (default) | integer

Time interval between consecutive samples applied to an output port.

Simulink deposits an input port signal on an HDL simulator signal at the specified sample rate. Conversely, Simulink reads an output port signal from a specified HDL simulator signal at the specified sample rate.

In general, Simulink handles port sample periods as follows:

- If you connect an input port to a signal that has an explicit sample period, based on forward propagation, Simulink applies that rate to the port.
- If you connect an input port to a signal that does not have an explicit sample period, Simulink assigns a sample period that is equal to the least common multiple (LCM) of all identified input port sample periods in the model.

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

The exact interpretation of the output port sample time depends on the settings of the **Timescales** parameters of the HDL Cosimulation block. See also “Simulation Timescales”.

### Dependencies

To enable this parameter, set **I/O Mode** to Output.

### Data Type — Data type for output signal

`Inherit` (default) | `Fixedpoint` | `Double` | `Single`

Select `Inherit` to automatically determine the data type. The block checks that the inherited word length matches the word length queried from the HDL simulator. If they do not match, Simulink generates an error message. For example, if you connect a Signal Specification block to an output, `Inherit` forces the data type specified by the Signal Specification block onto the output port.

If Simulink cannot determine the data type of the signal connected to the output port, it queries the HDL simulator for the data type of the port. As an example, if the HDL simulator returns the VHDL data type `STD_LOGIC_VECTOR` for a signal of size `N` bits, the data type `ufixN` is forced on the output port. The implicit fraction length is 0.

You can also assign an explicit data type, with optional **Fraction Length**. By explicitly assigning a data type, you can force fixed-point data types on output ports of the HDL Cosimulation block. For example, for an 8-bit output port, setting the **Sign** to `Signed` and setting the **Fraction Length** to 5 forces the data type to `sfix8_En5`. You cannot force width. The width is always inherited from the HDL simulator.

### Dependencies

To enable this parameter, set **I/O Mode** to Output.

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

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



**Sign — Sign component of output data type**

Unsigned (default) | Signed

Sign designation for explicit output port data type.

**Dependencies**

To enable this parameter, set **I/O Mode** to Output, and set **Data Type** to Fixedpoint.

**Fraction Length — Number of fractional bits in output data type**

integer

Size, in bits, of the fractional part of a fixed-point output signal. For example, for an 8-bit output port, setting the **Sign** to Signed and setting the **Fraction Length** to 5 forces the data type to `sfix8_En5`. You cannot force width; the width is always inherited from the HDL simulator.

**Dependencies**

To enable this parameter, set **I/O Mode** to Output, and **Data Type** property to Fixedpoint.

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

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

## Clocks

Create optional rising-edge and falling-edge clocks that apply stimuli to your cosimulation model. The scrolling list displays HDL clocks that drive values to the HDL signals that you are modeling, using the deposit method. The clock signals must be single-bit signals. Vector signals are not supported. For instructions on adding and editing clock signals, see “Creating Optional Clocks with the Clocks Pane of the HDL Cosimulation Block”.

**Full HDL Name — Signal path name**

string

Specify each clock as a signal path name, using the HDL simulator path name syntax. For example: `/manchester/clock` or `manchester.clk`.

For information about and requirements for path specifications in Simulink, see “Specify HDL Signal/Port and Module Paths for Cosimulation”.

You can copy signal path names directly from the HDL simulator **wave** window and paste them into the **Full HDL Name** field. Use the *Path.Name* view and not *Db::Path.Name* view. After pasting a signal path name into **Full HDL Name**, click **Apply** to complete the paste operation and update the signal list.

### **Active Clock Edge — HDL clock edge used to sample signals**

*Rising* (default) | *Falling*

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

### **Period — Clock period**

2 (default) | integer

To specify an explicit clock period, enter a sample time equal to or greater than two resolution units (ticks).

If the clock period is not an even integer, Simulink cannot create a 50% duty cycle. Instead, the HDL Verifier software creates the falling edge at  $\text{clockperiod}/2$  (rounded down to the nearest integer).

## **Timescales**

Choose a timing relationship between Simulink and the HDL simulator, either manually or automatically. These parameters 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:

- *Relative* timing relationship (Simulink seconds correspond to an HDL simulator-defined tick interval)
- *Absolute* timing relationship (Simulink seconds correspond to an absolute unit of HDL simulator time)

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

For detailed information on the relationship between Simulink and the HDL simulator during cosimulation, and on the operation of relative and absolute timing modes, see “Simulation Timescales”.

### Automatically determine timescale at start of simulation — When to calculate automatic timescale

true (default) | false

If you select this option, HDL Verifier calculates the timescale when you start the Simulink simulation. If this option is not selected, click **Determine Timescale Now** to calculate the timescale immediately without starting a simulation. Alternatively, you can manually select a timescale. For guidance through the automatic timescale calculation, see “Specify Timing Relationship Automatically” on page 1-16.

### 1 second in Simulink corresponds to {} in the HDL simulator — Timing relationship between Simulink and HDL simulator

integer and time units

This parameter consists of a *Time* value and a *TimeUnit* value.

To configure relative timing mode for a cosimulation:

- 1 Verify that `TICK`, the default setting for *TimeUnit*, is selected. If it is not, then select it from the list on the right.
- 2 Enter a scale factor in the *Time* text box on the left. The default scale factor is 1.

To configure absolute timing mode for a cosimulation:

- 1 Set *TimeUnit* to a unit of absolute time: `fs` (femtoseconds), `ps` (picoseconds), `ns` (nanoseconds), `us` (microseconds), `ms` (milliseconds), or `s` (seconds).
- 2 Enter a scale factor in the *Time* text box on the left. The default scale factor is 1.

## Connection

### Connection mode — Connection between Simulink and HDL simulator

Full Simulation (default) | Confirm Interface Only | No Connection

Type of connection between Simulink and the HDL simulator.

- `Full Simulation`: Confirm interface and run HDL simulation.
- `Confirm Interface Only`: Connect to the HDL simulator and check for signal names, dimensions, and data types, but do not run HDL simulation. During Simulink simulation, there is no contact with the HDL simulator.

- **No Connection:** Do not communicate with the HDL simulator. The HDL simulator does not need to be started.

### **HDL simulator is running on this computer — Same host for HDL simulator and Simulink**

`true (default) | false`

When both applications run on the same computer, you can choose shared memory or TCP sockets for the communication channel between the applications. If you do not select this option, only TCP/IP socket mode is available, and the **Connection method** list becomes unavailable.

### **Connection method — Connection between HDL simulator and Simulink**

`Socket (default) | Shared memory`

- **Socket:** Simulink and the HDL simulator communicate via a designated TCP/IP socket. TCP/IP socket mode is more versatile. You can use it for single-system and network configurations. This option offers the greatest scalability. For more on TCP/IP socket communication, see “TCP/IP Socket Ports”.
- **Shared memory:** Simulink and the HDL simulator communicate via shared memory. Shared memory communication provides optimal performance and is the default mode of communication.

### **Dependencies**

This parameter shows when you select **HDL Simulator is running on this computer**.

### **Host name — HDL simulator host machine**

`string`

This parameter applies if you run Simulink and the HDL simulator on different computers.

### **Port number or service — Socket port number**

`string`

Indicate a valid TCP socket port number or service for your computer system, if you are not using shared memory. For information on choosing TCP socket ports, see “TCP/IP Socket Ports”.

### **Show connection info on icon — Add connection parameters on block icon**

`true (default) | false`

When you select this option, the HDL Cosimulation block icon displays the current communication parameter settings. If you select shared memory, the icon displays `SharedMem`. If you select TCP socket communication, the icon displays `Socket` and displays the host name and port number in the format `hostname:port`.

This information can help you distinguish between multiple HDL Cosimulation blocks, where each block is communicating to a different instance of the HDL simulator.

## Simulation

### **Time to run HDL simulator before cosimulation starts — Offset that aligns Simulink with HDL simulator**

integer and time unit

Specifies the amount of time to run the HDL simulator before beginning simulation in Simulink. Specifying this time properly aligns the signal of the Simulink block and the HDL signal so that they can be compared and verified directly without additional delays.

This setting consists of a *PreRunTime* value and a *PreRunTimeUnit* value.

- *PreRunTime*: Any valid time value. The default is 0.
- *PreRunTimeUnit*: Specifies the units of time for *PreRunTime*.
  - Tick
  - s
  - ms
  - us
  - ns
  - ps
  - fs

### **Pre-simulation Tcl commands — Commands to run in HDL simulator before cosimulation**

string

The cosimulation tool executes these commands in the HDL simulator, before simulating the HDL component of your Simulink model. If you enter multiple commands on one line, append each command with a semicolon (;), the standard Tcl concatenation operator.

For example, use this parameter to generate a one-line echo command to confirm that a simulation is running, or a complex script that performs an extensive simulation initialization and startup sequence. You cannot use these commands to change simulation state.

You can specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the character vector cannot include commands such as `start`, `stop`, or `restart` (for ModelSim®) or `run`, `stop`, or `reset` (for Incisive).

### **Post-simulation Tcl commands — Commands to run in HDL simulator after cosimulation**

`string`

The cosimulation tool executes these commands in the HDL simulator, after simulating the HDL component of your Simulink model.

You can specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the string cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Incisive).

---

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

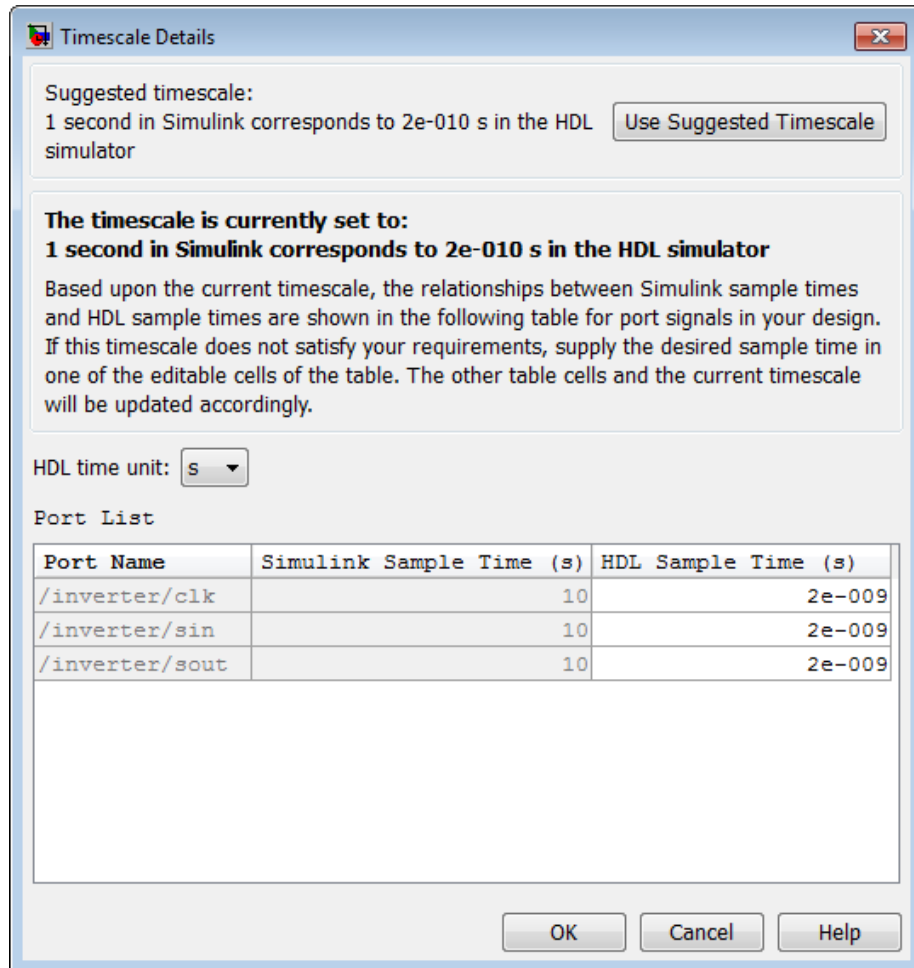
---

## **Specify Timing Relationship Automatically**

To have the HDL Verifier software calculate the timing relationship for you:

- 1 Start the HDL simulator. HDL Verifier software can obtain the resolution limit of the HDL simulator only when that simulator is running.
- 2 Choose to have HDL Verifier software suggest a timescale, immediately, or calculate the timescale when you start the Simulink simulation.
  - To have the calculation performed while you are configuring the block, on the **Timescale** tab, click **Determine Timescale Now**. The software connects Simulink with the HDL simulator so that Simulink can use the HDL simulator resolution to calculate the best timescale. The link then displays those results to you in the **Timescale Details** dialog box.

**Note** For the results to display, make sure that the HDL simulator is running and the design is loaded for cosimulation. The simulation does not have to be running.



You can accept the suggested timescale, or change the port list directly:

- To revert to the originally calculated settings, click **Use Suggested Timescale**.

- To view sample times for all ports in the HDL design, select **Show all ports and clocks**.
- To have the calculation performed when the simulation begins, select **Automatically determine timescale at start of simulation**, and click **Apply**. You obtain the same **Timescale Details** dialog box when the simulation starts in Simulink.

---

**Note** For the results to display, make sure that the HDL simulator is running and the design is loaded for cosimulation. The simulation does not have to be running.

---

HDL Verifier software analyzes all the clock and port signal rates from the HDL Cosimulation block when the software calculates the scale factor.

---

**Note** HDL Verifier software cannot automatically calculate a sample timescale based on any signals driven via Tcl commands or in the HDL simulator. The link software cannot perform such calculations because it cannot know the rates of these signals.

---

The link software returns the sample rate in either seconds or ticks:

- If the results are in seconds, then the link software was able to resolve the timing differences in favor of fidelity (absolute time).
- If the results are in ticks, then the link software was best able to resolve the timing differences in favor of efficiency (relative time).

Each time you select **Determine Timescale Now** or **Automatically determine timescale at start of simulation**, an interactive display opens. This display explains the results of the timescale calculations. If the link software cannot calculate a timescale for the given sample times, adjust your sample times in the **Port List**.

- 3 Click **Apply** to commit your changes.

---

**Note** HDL Verifier does not support timescales calculated automatically from frame-based signals.

---



## Set Parameters from a Tcl Script

You can create a Tcl script that lists the Tcl commands you want to execute on the HDL simulator, either pre- or post-simulation.

You can create a ModelSim DO file that lists Tcl commands and then specify that file with the ModelSim `do` command as follows:

```
do mycosimstartup.do
```

or

```
do mycosimcleanup.do
```

You can include the `quit -f` command in an after-simulation Tcl command or DO file to force ModelSim to shut down at the end of a cosimulation session. Specify all after-simulation Tcl commands in a single cosimulation block, and place `quit` at the end of the command or DO file.

Except for `quit`, the command or DO file that you specify cannot include commands that load a ModelSim project or modify simulator state. For example, they cannot include commands such as `start`, `stop`, or `restart`.

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

```
source mycosimstartup.script_extension
```

or

```
source mycosimcleanup.script_extension
```

You can include the `exit` command in an after-simulation Tcl script to force the HDL simulator to shut down at the end of a cosimulation session. Specify all after-simulation Tcl commands in a single cosimulation block and place `exit` at the end of the command or Tcl script.

Except for `exit`, the command or Tcl script that you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, they cannot include commands such as `run`, `stop`, or `reset`.

This example shows a Tcl script when the `-gui` argument was used with `hdlsimmatlab` or `hdlsimulink`:

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

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

```
after 1000 {exit}
```

## Model Examples

## See Also

### Topics

“Import HDL Code for HDL Cosimulation Block”

“Create Simulink Model for Component Cosimulation”

“Create a Simulink Cosimulation Test Bench”

“Simulation Timescales”

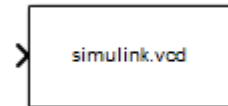
“Clock, Reset, and Enable Signals”

**Introduced in R2008a**

## To VCD File

Generate value change dump (VCD) file

**Library:** HDL Verifier / For Use with Mentor Graphics  
 Modelsim  
 HDL Verifier / For Use with Cadence Incisive



## Description

The To VCD File block generates a VCD file that logs changes to its input ports. You can use VCD files during design verification in these ways:

- Compare results of multiple simulation runs, using the same or different simulator environments.
- Provide input to postsimulation analysis tools.
- Porting areas of an existing design to a new design.

You can specify the following parameters:

- Name of the generated VCD file
- Number of block input ports
- Timescale, that relates Simulink sample times with HDL simulator ticks

VCD files can grow large for large designs or small designs with long simulation runs. The maximum number of signals supported in a generated VCD file is  $94^3$  (830,584).

You can use the To VCD File block in models running in normal, accelerator, or rapid accelerator simulation modes. The To VCD File parameters are not tunable in any of the simulation modes. For more information about these modes, see “How Acceleration Modes Work” (Simulink).

The To VCD File block is integrated into the Simulink Signal and Scope Manager. When you add a VCD block to a model using the manager, the signal name that appears in the VCD file may not be the one you specified. After simulation, open the VCD file and check the signal name. If you cannot find the signal name you specified, look for an automatic

signal name such as `In_1`. When you use the VCD block directly from the HDL Verifier library, the signal names match correctly.

---

**Note** The To VCD File block does not support framed signals.

---

## VCD File Format

The format of generated VCD files adheres to IEEE® Std 1364-2001. The table describes the format.

| VCD File Content   | Description  |
|--|--|
| <pre>\$date 23-Sep-2003 14:38:11 \$end</pre>   | Date and time the file was generated.  |
| <pre>\$version HDL       Verifier version 1.0 \$ end</pre>   | Version of the To VCD File block that generated the file.  |
| <pre>\$timescale 1 ns \$ end</pre>   | Timescale used during the simulation.  |
| <pre>\$scope module manchestermodel \$end</pre>  | Scope of module being dumped.  |
| <pre>\$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</pre> | <p>Variable definitions. Each definition associates a signal with a character identification code (symbol).</p> <p>The symbols are derived from printable characters in the ASCII character set from ! to ~.</p> <p>Variable definitions also include the variable type (wire) and size in bits.</p> |
| <pre>\$upscope \$end</pre>   | Marks a change to the next highest level in the HDL design hierarchy.  |

| VCD File Content                             | Description  |
|--|--|
| \$enddefinitions \$end                       | Marks the end of the header and definitions section.   |
| #0   | Simulation start time.   |
| \$dumpvars<br>0!<br>0"<br>0#<br>0\$<br>\$end | Lists the values of all defined variables at time 0.   |
| #630<br>1!                                   | Starting point of logged value changes from checks of variable values made at each simulation time increment.<br><br>This entry indicates that at 63 nanoseconds, the value of signal Original Data changed from 0 to 1. |
| .<br>. .<br>#1160<br>1#<br>1\$               | At 116 nanoseconds, the values of signals Recovered Data and Data Validity changed from 0 to 1.  |
| \$dumpoff<br>x!<br>x"<br>x#<br>x\$<br>\$end  | Marks the end of the file by dumping the values of all variables as the value x.   |

## Display VCD File Data

You can display VCD file data graphically or analyze the data with postprocessing tools. For example, the ModelSim `vcd2wlf` tool converts a VCD file to a WLF file, which you can view in a ModelSim **wave** window. Other 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.

## Ports

Specify the number of signals to log using **Number of input ports**. The block has no output ports.

## Input

**Port\_1, Port\_2, . . . , Port\_N** — Signal to log to VCD file

scalar | vector | matrix

Multi-dimensional signals are flattened to 1-D vectors in the VCD file.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | Fixed-point

## Parameters

**VCD file name** — Name of generated VCD file

string

Name of the generated VCD file. If you specify a file name only, Simulink places the file in your current MATLAB® folder. To place the generated file in a different location, specify a complete path name. If you specify the same name for multiple To VCD File blocks, Simulink automatically adds a numeric postfix to identify each instance uniquely.

---

**Note** To save the generated file with the .vcd file extension, you must specify it explicitly.

---

**Number of input ports** — Number of input signals to log

integer

Number of input signals to log data from. The block can log up to  $94^3$  (830,584) signals, each of which maps to a unique symbol in the VCD file.

In some cases, a single input port maps to multiple symbols. This multiple mapping occurs when the input port receives a multidimensional signal. Because the VCD specification does not include multidimensional signals, Simulink flattens them to a 1-D vector in the file.

## Timescale — Timing relationship between Simulink and the HDL simulator integer and time units

Timing relationship, defined as the correspondence between one second of Simulink time and some quantity of HDL simulator time. You can express this quantity of HDL simulator time in one of the following ways:

- In *relative* terms, that is, as some number of HDL simulator ticks. In this case, the cosimulation operates in *relative timing mode*, which is the timing mode default.

To use relative mode, in the **1 second in Simulink corresponds to {value} {unit} in the HDL simulator** parameter, set the unit to `Tick`, and the value to the number of ticks you want. The default value is 1 tick.

- In *absolute* units, such as milliseconds or nanoseconds. In this case, the cosimulation operates in *absolute timing mode*.

To use absolute mode, in the **1 second in Simulink corresponds to {value} {unit} in the HDL simulator** parameter, set the number of resolution units and the type of unit (`fs`, `ps`, `ns`, `us`, `ms`, `s`). Then, in the **1 HDL Tick is defined as** parameter, set the value of the HDL simulator tick to 1, 10, or 100, and choose a resolution unit.

## Model Examples

## See Also

### Topics

“Add a Value Change Dump (VCD) File”

“Visually Compare Simulink Signals with HDL Signals”

“Simulation Timescales”

Introduced in R2008a





## System Objects — Alphabetical List

---

## hdlverifier.FILSimulation System object

**Package:** hdlverifier

Construct System object for FIL simulation with MATLAB

### Description

The `FILSimulation` System object™ creates, launches, and controls FPGA execution from MATLAB.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

### Construction

`hdlverifier.FILSimulation` is a virtual class and cannot be instantiated directly. To use it, launch the FPGA-in-the-Loop Wizard and generate your own custom `FILSimulation` derived class. Then you can instantiate your own FIL simulation object with the following function:

`filobj = toplevel_fil` creates a new instance of the derived class generated by the FPGA-in-the-Loop Wizard. `toplevel` is the name of the top-level module in your HDL code.

You can adjust writable properties on the System object by using the `get` and `set` methods or by setting the property directly. See “Properties” on page 2-2.

### Properties

#### Connection

Parameters for the connection with the FPGA board

**R/W Access:** Read only

**Default:** `char('UDP','192.168.0.2','00-0A-35-02-21-8A')`

**Attributes:**

|                                 |                  |                              |
|---------------------------------|------------------|------------------------------|
| Connection type                 | character vector | Example: 'UDP'               |
| Board IP address                | character vector | Example: '192.168.0.2'       |
| Board MAC address<br>(optional) | character vector | Example: '00-0A-35-02-21-8A' |

**DUTName**

DUT top level name

**R/W Access:** Read only

**Default:** ''

**Attributes:**

|                       |                  |                         |
|-----------------------|------------------|-------------------------|
| Name of DUT top level | character vector | Example: 'inverter_top' |
|-----------------------|------------------|-------------------------|

**FPGABoard**

FPGA board name

**R/W Access:** Read only

**Default:** ''

**FPGAProgrammingFile**

Path to the programming file for the FPGA

**R/W Access:** Read and write

**Default:** ''

**Attributes:**

|           |                  |                             |
|-----------|------------------|-----------------------------|
| Path name | character vector | Example: 'c:\work\filename' |
|-----------|------------------|-----------------------------|

**FPGAVendor**

Name of the FPGA chip vendor

**R/W Access:** Read only

**Default:** 'Xilinx'

**Attributes:**

|                  |                  |                              |
|------------------|------------------|------------------------------|
| Chip vendor name | character vector | Examples: 'Altera', 'Xilinx' |
|------------------|------------------|------------------------------|

**InputBitWidths**

Input widths, in bits

**R/W Access:** Read only

**Default:** 0

**Attributes:**

|   |                               |                            |
|---|-------------------------------|----------------------------|
| Integer or vector of integer specifying the bit widths of the inputs. | integer or vector of integers | Examples:<br>10<br>[12, 6] |
|---|-------------------------------|----------------------------|

If you provide only a scalar, the inputs each have the same bit width; otherwise you should provide a vector of the same size as the number of inputs.

**InputSignals**

Input paths in the HDL code

**R/W Access:** Read only

**Default:** ''

**Attributes:**

input port name of each input in the HDL character vector or cell array of N character vectors Examples: 'in1', char('in1','in2')

### **OutputBitWidths**

Output widths, in bits

**R/W Access:** Read only

**Default:** 0

### **Attributes:**

|  |                               |                           |
|--|-------------------------------|---------------------------|
| Integer or vector of integer specifying the bit widths of the outputs. | integer or vector of integers | Examples:<br>10<br>[12,6] |
|--|-------------------------------|---------------------------|

If you provide only a scalar, the outputs each have the same bit width. Otherwise you should provide a vector of the same size as the number of outputs.

### **OutputDataTypes**

Output data types

**R/W Access:** Read and write

**Default:** fixedpoint

### **Attributes:**

Character vector or cell array of N character vectors specifying the data type of the output. If you only provide one data type, each of the outputs has the same type. Otherwise, you should provide a cell array of the same size as the number of outputs.

|   |  |
|---|--|
| character vector or cell array of N character vectors | Examples: 'logical', 'integer', 'fixedpoint', 'single', 'double'<br>char('integer','fixedpoint') |
|---|--|

### **OutputDownsampling**

Downsampling factor and phase of the outputs

**R/W Access:** Read and write

**Default:** [1, 0]

#### **Attributes:**

Vector of 2 integers: vector  
The first integer specifies the downsampling factor and is positive. The second integer specifies the phase and is null or positive and inferior to the downsampling factor.

Examples:

[3, 1]

### **OutputFractionLengths**

Output fraction lengths

**R/W Access:** Read and write

**Default:** 0

**Attributes:**

|   |                               |                           |
|---|-------------------------------|---------------------------|
| Integer or vector of integer specifying the fraction length of the outputs. | integer or vector of integers | Examples:<br>10<br>[12,6] |
|---|-------------------------------|---------------------------|

If you provide only a scalar, each output has the same fraction length. Otherwise you should provide a vector of the same size as the number of outputs.

**OutputSignals**

Output port name in the HDL top level

**R/W Access:** Read only

**Default:** ''

**Attributes:**

|  |   |  |
|--|---|--|
| Character vector or cell array of N character vectors containing the output port name of each output in HDL. | character vector or cell array of N character vectors | Examples: 'out1',<br>char('out1','out2') |
|--|---|--|

**OutputSigned**

Sign of the outputs

**R/W Access:** Read and write

**Default:** false

**Attributes:**

|   |                              |  |
|---|------------------------------|--|
| Boolean or vector of boolean specifying the sign of the outputs. If you provide only a scalar, each output has the same sign. Otherwise, you should provide a vector of the same size as the number of outputs. | Boolean of vector of Boolean | Examples: true (signed), false (unsigned)<br>[true, true, false] |
|---|------------------------------|--|

### **OverclockingFactor**

Hardware overclocking factor

**R/W Access:** Read and write

**Default:** 1

#### **Attributes:**

|   |         |               |
|---|---------|---------------|
| Positive integer specifying the overclocking factor for the hardware. | integer | Example:<br>3 |
|---|---------|---------------|

### **ScanChainPosition**

Position of the FPGA in the JTAG scan chain

**R/W Access:** Read only

**Default:** 1

#### **Attributes:**

|  |         |               |
|--|---------|---------------|
| Positive integer specifying the position of the FPGA in the JTAG scan chain. | integer | Example:<br>1 |
|--|---------|---------------|

### **SourceFrameSize**

Frame size of the source (only for HDL source block)



**R/W Access:** Read and write

**Default:** 1

**Attributes:**

|  |         |               |
|--|---------|---------------|
| Integer specifying the frame size of the source when the HDL is a source block (no input). | integer | Example:<br>1 |
|--|---------|---------------|

## Methods

|             |  |
|-------------|--|
| programFPGA | Load programming file onto FPGA                        |
| step        | Run FIL simulation for set of inputs and return output |

| Common to All System Objects |  |
|------------------------------|--|
| clone                        | Create System object with same property values   |
| getNumInputs                 | Expected number of inputs to a System object     |
| getNumOutputs                | Expected number of outputs of a System object    |
| isLocked                     | Check locked states of a System object (logical) |
| release                      | Allow System object property value changes       |

## See Also

### Topics

“FPGA-in-the-Loop simulation using MATLAB System Object”

Introduced in R2012b

## programFPGA

**System object:** hdlverifier.FILSimulation

**Package:** hdlverifier

Load programming file onto FPGA

### Syntax

```
programFPGA(filobj)
```

### Description

`programFPGA(filobj)` loads the FPGA through the JTAG cable using the `FILSimulation` property information from `ProgrammingFile`, `ScanChainPosition` and `BoardName`.

### Input Arguments

**filobj**

Instance of `FILSimulation`

---

## step

**System object:** hdlverifier.FILSimulation

**Package:** hdlverifier

Run FIL simulation for set of inputs and return output

## Syntax

```
[hdloutputs] = step(filobj, [hdlinputs])
```

## Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

`[hdloutputs] = step(filobj, [hdlinputs])` connects to the FPGA, writes `hdlinputs` to the FPGA and reads `hdloutputs` from the FPGA.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## Input Arguments

**filobj**

Instance of `FILSimulation`

**hdlinputs**

Set of inputs to run on FPGA

## Output Arguments

**hdloutputs**

Set of outputs returned by the FPGA

# hdlverifier.HDLCosimulation System object

**Package:** hdlverifier

Construct System object for HDL cosimulation with MATLAB

## Description

The `HDLCosimulation` System object cosimulates MATLAB and a hardware component. It does so by applying input signals to and reading output signals from an HDL model under simulation in the HDL simulator. You can use this object to model a source or sink device by configuring the object with input or output ports only.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`h = HDLCosimulation(Name,Value)` creates a new instance of `HDLCosimulation` with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name on page 2-14 and `Value` is the corresponding value. `Name` must appear inside single quotes (`'`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`h = hdlcosim(Name,Value)` creates a new instance of `HDLCosimulation` using a shortcut constructor.

The Cosimulation Wizard creates an HDL Cosimulation System object using existing HDL code. This workflow creates an HDL launch script for easier startup.

# Properties

### Connection

Parameters for the connection with the HDL simulator.

- The first element is the connection type ('SharedMemory', 'Socket'). If shared memory is used, then port number and host name are not applicable.
- The second element is the port number, which must be a positive integer. Optional. It is set to 4449 if not otherwise specified.
- The third element is the host name of the HDL session. Optional. Set to localhost if not specified.

Default: {'SharedMemory'}

Example values:

```
{ 'SharedMemory' }  
{ 'Socket' }  
{ 'Socket', 1234 }  
{ 'Socket', 1234, 'hostname' }
```

### FrameBasedProcessing

---

**Note** FrameBasedProcessing property will be removed in a future release. Sample mode or frame mode is automatically detected based on the size of the inputs during the step method execution.

---

Enable frame-based processing

Default: false

Example values:

```
true/false
```

### InputSignals

Input paths in the HDL code

Default: ''

Example values:

```
 '/top/in1'  
{ '/top/in1', '/top/in2' }
```

### **OutputDataTypes**

Data types of the output signals.

Default: ''

Example values:

```
 {'fixedpoint'}  
{ 'double', 'single' }
```

---

**Note** When `OutputDataTypes` is `fixedpoint`, and the size matches the size of a built-in type (8,16,32,64), and `OutputFractionLengths` is set to 0, the output signal is returned as the built-in type.

---

### **OutputFractionLengths**

Output fraction lengths. Must be an integer or vector of integer specifying the fraction length of the outputs. If you provide only a scalar, all the outputs are of the same type. Otherwise, provide a vector of the same size as the number of outputs.

Default: 0

Example values:

```
 10  
 [12, 6]
```

### **OutputSignals**

Output paths in the HDL code

Default: ''

Example values:

```
 '/top/out1'  
{ '/top/out1', '/top/out2' }
```

### **OutputSigned**

Output sign. Must be a boolean or vector of boolean specifying the sign of the outputs. If you provide only a scalar, all outputs are of the same type. Otherwise, provide a vector of the same size as the number of outputs.

Default: `false`

Example values:

```
true/false  
[true,true]
```

### **PreRunTime**

Delay in HDL simulator before the cosimulation starts

Default: `{0, 'ns'}`

Example values:

```
{10, 'fs'}  
{25, 'ps'}  
{4, 'ns'}  
{500, 'us'}  
{5, 'ms'}  
{1, 's'}
```

### **SampleTime**

Elapsed time in the HDL simulator between each call to step

Default: `{10, 'ns'}`

Example values:

```
{10, 'fs'}  
{25, 'ps'}  
{4, 'ns'}  
{500, 'us'}  
{5, 'ms'}  
{1, 's'}
```



**TclPostSimulationCommand**

Tcl post-simulation command executed by the HDL simulator during a call to release

Default: ''

Example value:

```
'echo "done"'
```

**TclPreSimulationCommand**

Tcl presimulation command executed by the HDL simulator during the first call to step or during the next call to step after a call to release

Default: ''

Example value:

```
'force /top/rst 1 0, 0 2 ns; force /top/clk 0 0, 1 1 ns -repeat 2 ns'
```

## Methods

`reset` Unlock object, release connection to HDL simulator, and reset internal state

`step` Run HDL simulator for set of inputs and return output

| Common to All System Objects |  |
|------------------------------|--|
| <code>clone</code>           | Create System object with same property values   |
| <code>getNumInputs</code>    | Expected number of inputs to a System object     |
| <code>getNumOutputs</code>   | Expected number of outputs of a System object    |
| <code>isLocked</code>        | Check locked states of a System object (logical) |
| <code>release</code>         | Allow System object property value changes       |

## Examples

See the following Featured Examples:

- “Verify Viterbi Decoder Using MATLAB System Object and Cadence Incisive”
- “Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim”

**Introduced in R2012b**

## reset

**System object:** hdlverifier.HDLCosimulation

**Package:** hdlverifier

Unlock object, release connection to HDL simulator, and reset internal state

## Syntax

```
reset(cosimobj)
```

## Description

`reset(cosimobj)` unlocks the System object, `cosimobj`, and releases its connection with the HDL simulator. It also resets all internal states.

## Input Arguments

**cosimobj**

Instance of HDLCosimulation.

## See Also

`hdlverifier.HDLCosimulation`

# step

**System object:** hdlverifier.HDLCosimulation

**Package:** hdlverifier

Run HDL simulator for set of inputs and return output

## Syntax

```
[hdloutputs] = step(cosimobj,hdlinputs)
```

## Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`[hdloutputs] = step(cosimobj,hdlinputs)` connects to the HDL simulator, writes `hdlinputs` to the HDL simulator and reads `hdloutputs` from the HDL simulator. The elapsed HDL simulator time between each call to `step` is defined by the `SampleTime` property.

---

**Note** `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

---

## Input Arguments

**cosimobj**

Instance of HDLCosimulation.

**hdlinputs**

Set of inputs for HDL simulator to run

**Default:**

## Output Arguments

**hdloutputs**

Set of outputs returned by HDL simulator

## Examples

See the following Featured Examples:

- “Verify Viterbi Decoder Using MATLAB System Object and Cadence Incisive”
- “Verify Viterbi Decoder Using MATLAB System Object and Mentor Graphics ModelSim”

## See Also

`hdlverifier.HDLCosimulation`



## Functions — Alphabetical List

---

## breakHdlSim

Execute `stop` command in HDL simulator from MATLAB

### Syntax

```
breakHdlSim()  
breakHdlSim('portNumber')  
breakHdlSim('portNumber', 'hostName')
```

### Description

`breakHdlSim()` executes a stop command on the HDL simulator on the local host. Use this function to unblock the HDL simulator after the HDL simulator has loaded the simulation but before Simulink starts the simulation. If, after starting the simulation, you decide to add more signals to the waveform window, use this function to unblock the HDL simulator first. When you use `breakHdlSim`, make sure that you specify the current connection information to the HDL simulator.

`breakHdlSim('portNumber')` executes a stop command on the HDL simulator on port *portNumber*.

`breakHdlSim('portNumber', 'hostName')` executes a stop command on the HDL simulator on host *hostName*.

### Examples

Stop the HDL simulator that is currently running on the local host.

```
>> breakHdlSim()
```

Stop the HDL simulator that is currently running on port 1234.

```
>> breakHdlSim('1234')
```

Stop the HDL simulator that is currently running on port 1234 and host "mylinux".



```
>> breakHdlSim('1234', 'mylinux')
```

## See Also

pingHdlSim

**Introduced in R2008a**

# Cosimulation Wizard

Generate a cosimulation block or System object from existing HDL files

## Description

Run your HDL design as part of a Simulink model, or MATLAB script. The Cosimulation Wizard generates a cosimulation block, System object, or callback function that compiles the HDL code and launches the HDL simulator.

## Open the Cosimulation Wizard App

- MATLAB command prompt: Enter `cosimWizard`.

## Examples

- “Verify Raised Cosine Filter Design Using MATLAB”
- “Verify Raised Cosine Filter Design Using Simulink”

## See Also

### Topics

“Verify Raised Cosine Filter Design Using MATLAB”  
“Verify Raised Cosine Filter Design Using Simulink”  
“Import HDL Code for MATLAB Function”  
“Import HDL Code for MATLAB System Object”  
“Import HDL Code for HDL Cosimulation Block”

**Introduced in R2012b**

## dec2mvl

Convert decimal integer to binary character vector

### Syntax

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

### Description

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

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

### Examples

#### Convert Decimal Integers to Binary Strings

The `dec2mvl` function returns strings of binary digits.

```
dec2mvl(23)
```

```
ans =  
    '10111'
```

```
dec2mvl(-23)
```

```
ans =
```

```
'101001'
```

The second argument specifies the minimum number of bits represented in the string.

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

```
ans =
```

```
'11101001'
```

## See Also

[mvl2dec](#)

**Introduced in R2008a**

# dpigen

Generate SystemVerilog DPI component from MATLAB function

## Syntax

```
dpigen fcn -args args
dpigen fcn -args args -testbench tb_name -options options files -c -
launchreport
```

## Description

`dpigen fcn -args args` generates a SystemVerilog DPI component shared library from MATLAB function `fcn` and all the functions that `fcn` calls.

The argument `-args args` specifies the type of inputs the generated code can accept. The generated DPI component is specialized to the class and size of the inputs. Using this information, `dpigen` generates a DPI component that emulates the behavior of the MATLAB function.

`fcn` and `-args args` are required input arguments. The MATLAB function must be on the MATLAB path or in the current folder.

`dpigen fcn -args args -testbench tb_name -options options files -c -launchreport` generates a SystemVerilog DPI component shared library according to the options specified. You can specify zero or more optional arguments, in any order.

- `-testbench tb_name` also generates a test bench for the SystemVerilog DPI component. The MATLAB test bench must be on the MATLAB path or in the current folder.
- `-options options` specifies additional options for the compiler and code generation.
- `files` specifies custom files to include in the generated code.
- `-c` generates C code only.
- `-launchreport` generates and opens a code generation report.

You cannot generate a DPI component for use on a different platform than the MATLAB host. For instance if you use 64-bit MATLAB on Windows®, you get a 64-bit DLL, which can be used only with a 64-bit HDL simulator in Windows.

## Examples

### Generate DPI Component and Test Bench

Generate a DPI component and test bench for the function `fun.m` and its associated test bench, `fun_tb.m`. The `dipgen` function compiles the component automatically using the default compiler. The `-args` option specifies that the first input type is a double and the second input type is an `int8`.

```
dipgen -testbench fun_tb.m -I E:\HDLTools\ModelSim\10.2c-mw-0\questa_sim\include fun.m
      -args {double(0),int8(0)}

### Generating DPI-C Wrapper fun_dpi.c
### Generating DPI-C Wrapper header file fun_dpi.h
### Generating SystemVerilog module fun_dpi.sv
### Generating makefiles for: fun_dpi
### Compiling the DPI Component
### Generating SystemVerilog test bench fun_tb.sv
### Generating test bench simulation script for Mentor Graphics ModelSim/Questasim run_tb_mq.do
### Generating test bench simulation script for Cadence Incisive run_tb_ncsim.sh
### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
```

### Generate DPI Component and Test Bench Without Compiling

Generate a DPI component and a test bench for the function `fun.m` and its associated test bench, `fun_tb.m`. To prevent the `dipgen` function from compiling the library, include the `-c` option. Send the source code output to 'MyDPIProject'.

```
dipgen -c -d MyDPIProject -testbench fun_tb.m fun.m -args {double(0),int8(0)}

### Generating DPI-C Wrapper fun_dpi.c
### Generating DPI-C Wrapper header file fun_dpi.h
### Generating SystemVerilog module fun_dpi.sv
### Generating makefiles for: fun_dpi
### Generating SystemVerilog test bench fun_tb.sv
### Generating test bench simulation script for Mentor Graphics ModelSim/Questasim run_tb_mq.do
```

```
### Generating test bench simulation script for Cadence Incisive run_tb_ncsim.sh
### Generating test bench simulation script for Synopsys VCS run_tb_vcs.sh
```

## Input Arguments

### **fcn** — Name of MATLAB function

character vector

Name of MATLAB function to generate the DPI component from, specified as a character vector. The MATLAB function must be on the MATLAB path or in the current folder.

### **-args args** — Data type and size of MATLAB function inputs

cell array

Data type and size of MATLAB function inputs, specified as a cell array. Specify the input types that the generated DPI component accepts. `args` is a cell array specifying the type of each function argument. Elements are converted to types using `coder.typeof`. This argument is required.

This argument has the same functionality as the `codegen` function argument `args`. `args` applies only to the function, `fcn`.

Example: `-args {double(0),int8(0)}`

### **-testbench tb\_name** — MATLAB test bench used to generate test bench for generated DPI component

character vector

MATLAB test bench used to generate test bench for generated DPI component, specified as a character vector. The `dpigen` function uses this test bench to generate a SystemVerilog test bench along with data files and execution scripts. The MATLAB test bench must be on the MATLAB path or in the current folder.

Example: `-testbench My_Test_bench.m`

### **-options options** — Compiler and code generation options

character vector

Compiler and `codegen` options, specified as character vectors. These options are a subset of the options for `codegen`. The `dpigen` function gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the right-most option prevails.

You can specify zero or more optional arguments, in any order. For example:

```
dpigen -c -d MyDPIProject -testbench fun_tb.m fun.m -args
{double(0),int8(0)} -launchreport
```

| Option flag                         | Option value  |
|-------------------------------------|---|
| <p><code>-I include_path</code></p> | <p>Specifies the path to folders containing headers and library files needed for <code>codegen</code>, specified as a character vector. Add <code>include_path</code> to the beginning of the code generation path.</p> <p>For example:</p> <pre>-I E:\HDLTools\ModelSim\10.2c- mw-0\questa_sim\include</pre> <p><code>include_path</code> must not contain spaces, which can lead to code generation failures in certain operating system configurations. If the path contains non 7-bit ASCII characters, such as Japanese characters, <code>dpigen</code> might not find files on this path.</p> <p>When converting MATLAB code to C/C++ code, <code>dpigen</code> searches the code generation path first.</p> <p>Alternatively, you can specify the include path with the <code>files</code> input argument.</p> |



| Option flag    | Option value   |
|----------------|--|
| -config config | <p>Specify a custom configuration object using <code>coder.config('dll')</code>. The DPI component must be a shared library.</p> <p>To avoid using conflicting options, do not combine a configuration object with command-line options. Usually the <code>config</code> object offers more options than the command-line flags.</p> <hr/> <p><b>Note</b> Not all the options in the <code>config</code> object are compatible with the DPI feature. If you try to use an incompatible option, an error message informs you of which options are not compatible.</p> |
| -o output      | Specify the name of the generated component as a character vector. The <code>dpigen</code> function adds a platform-specific extension to this name for the shared library.  |
| -d dir         | <p>Specify the output folder. All generated files are placed in <i>dir</i>. By default, files are placed in <code>./codegen/dll/&lt;function&gt;</code>.</p> <p>For example, when <code>dpigen</code> compiles the function <code>fun.m</code>, the generated code is placed in <code>./codegen/dll/fun</code>.</p>  |

| Option flag                   | Option value   |
|-------------------------------|--|
| <code>-globals globals</code> | <p>Specify initial values for global variables in MATLAB files. The global variables in your function are initialized to the values in the cell array <code>GLOBALS</code>. The cell array provides the name and initial value of each global variable.</p> <p>If you do not provide initial values for global variables using the <code>-globals</code> option, <code>dpigen</code> checks for the variables in the MATLAB global workspace. If you do not supply an initial value, <code>dpigen</code> generates an error.</p> <p>MATLAB Coder and MATLAB each have their own copies of global data. For consistency, synchronize their global data whenever the two products interact. If you do not synchronize the data, their global variables might differ.</p> |

**files** — Custom files to include in the generated code

character vector

Custom files to include in the generated code, each file specified as a character vector. The files build along with the MATLAB function specified by `fcn`. List each file separately, separated by a space. The following extensions are supported.

| File Type         | Description  |
|-------------------|--|
| <code>.c</code>   | Custom C file  |
| <code>.cpp</code> | Custom C++ file                                      |
| <code>.h</code>   | Custom header file (included by all generated files) |
| <code>.o</code>   | Object file  |
| <code>.obj</code> | Object file  |
| <code>.a</code>   | Library file   |
| <code>.so</code>  | Library file   |
| <code>.lib</code> | Library file   |

In Windows, if your MATLAB function contains matrix or vector output or input arguments, use the `files` option to specify the library (`.lib`) that contains the

ModelSim DPI definitions. Otherwise, you must manually modify the generated Makefile (\* .mk) and then compile the library separately.

**-c — Option to generate C code only**

character vector

Option to generate C code without compiling the DPI component, specified as the character vector `-c`. If you do not use the `-c` option, `dpigen` tries to compile the DPI component using the default compiler. To select a different compiler, use the `-config` option and refer to the `codegen` documentation for instructions on specifying the different options.

**-launchreport — Option to generate and open a code generation report**

character vector

Option to generate and open a code generation report, specified as the character vector `-launchreport`.

## See Also

`codegen`

Introduced in R2014b

## FPGA-in-the-Loop Wizard

Generate an FPGA-in-the-loop (FIL) block or System object from existing HDL files

### Description

FPGA-in-the-loop (FIL) enables you to run a Simulink or MATLAB simulation that is synchronized with an HDL design running on an Xilinx® or Altera® FPGA board.

This link between the simulator and the board enables you to:

- Verify HDL implementations directly against algorithms in Simulink or MATLAB.
- Apply data and test scenarios from Simulink or MATLAB to the HDL design on the FPGA.
- Integrate existing HDL code with models under development in Simulink or MATLAB.

### Open the FPGA-in-the-Loop Wizard App

- MATLAB command prompt: Enter `filWizard`. You provide the HDL code and all related information for creating a FIL block for simulation with an FPGA device.

### Examples

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”

### Programmatic Use

`filWizard(filename)` relaunches the FIL Wizard using a configuration file from a previous session. At the end of each FIL Wizard session, the tool saves a MAT-file that contains the session information. You can use this MAT-file to restore the session later.

## See Also

### Topics

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”
- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”

**Introduced in R2012b**

## hdldaemon

Control MATLAB server that supports interactions with HDL simulator

### Syntax

```
hdldaemon
hdldaemon(Name, Value)
hdldaemon(Option)

s=hlddaemon( ___ )
```

### Description

`hdldaemon` starts the HDL Link MATLAB server using shared memory inter-process communication. Only one `hdldaemon` per MATLAB session can be running at any given time.

`hdldaemon(Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

- If you do not specify memory type, the server starts using shared memory.
- If you specify the `socket Name, Value` argument, the server starts using socket memory.

---

**Note** If server is already running, issuing `hdldaemon` with these arguments shuts down the current server and then starts a new server session using shared memory (unless socket is specified).

---

`hdldaemon(Option)` accepts a single optional input. Only one option may be specified in a single call. You must establish the server connection before calling `hdldaemon` with one of these options.

`s=hlddaemon( ___ )` returns the server status connection in structure `s`, using any of the input arguments in the previous syntaxes.

## Examples

### Start MATLAB Server With Shared Memory

Start the MATLAB server using shared memory communication and use an integer representation of time.

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

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

### Start MATLAB Server With Socket Communication

Start MATLAB server and specify socket communication on port 4449.

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

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

### Check Server Status

With one or more connections:

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

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

With no connections:

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

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

Server has not been started:

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

```
HDLDaemon is NOT running
```

#### Check Connection Information

Check connection information for communication mode, number of existing connections, and the interprocess communication identifier (`ipc_id`) the MATLAB server is using for a link.

Returned message for a socket connection:

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

Returned message for a shared memory connection:

```
x=hlddaemon('status')  
  
x =  
      comm: 'shared memory'  
 connections: 0  
   ipc_id: '\\.\pipe\E505F434-F023-42a6-B06D-DEFD08434C67'
```

You can examine `ipc_id` by entering it at the MATLAB command prompt:

```
x.ipc_id  
  
'\\.\pipe\E505F434-F023-42a6-B06D-DEFD08434C67'
```

#### Shut Down Server

Shut down server without shutting down MATLAB.

```
hlddaemon('kill')  
  
HLDaemon server was shutdown
```

#### Issue Tcl Commands

Issue simple or complex Tcl commands.



Simple example:

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

Complex example:

```
tclcmd = {[ 'cd ',unixprojdir],...
          'vlib work',... % create library (if applicable)
          ['vcom -performdefaultbinding ' unixsrcfile1],...
          ['vcom -performdefaultbinding ' unixsrcfile2],...
          ['vcom -performdefaultbinding ' unixsrcfile3],...
          'vsimmatlab work.osc_top ',...
          'matlabcp u_osc_filter -mfunc oscfilter',...
          'add wave sim:/osc_top/clk',...
          'add wave sim:/osc_top/clk_enable',...
          'add wave sim:/osc_top/reset',...
          ['add wave -height 100 -radix decimal -format analog-step...
           -scale 0.001 -offset 50000 ', 'sim:/osc_top/osc_out'],...
          ['add wave -height 100 -radix decimal -format analog-step...
           -scale 0.00003125 -offset 50000 ', 'sim:/osc_top/filter1x_out'],...
          ['add wave -height 100 -radix decimal -format analog-step...
           -scale 0.00003125 -offset 50000 ', 'sim:/osc_top/filter4x_out'],...
          ['add wave -height 100 -radix decimal -format analog-step...
           -scale 0.00003125 -offset 50000 ', 'sim:/osc_top/filter8x_out'],...
          'force sim:/osc_top/clk_enable 1 0',...
          'force sim:/osc_top/reset 1 0, 0 120 ns',...
          'force sim:/osc_top/clk 1 0 ns, 0 40 ns -r 80ns',...
        ];
```

This example is taken from "Implementing the Filter Component of an Oscillator in MATLAB". See the full example for use of this complex Tcl command in context.

- Implementing the Filter Component of an Oscillator in MATLAB

## Input Arguments

**Option** — Server option to shut down MATLAB server or display server status

```
'kill' | 'stop' | 'status'
```

Server option to shut down MATLAB server or display server status, specified as one of these character vectors:

'kill'

Shuts down the MATLAB server without shutting down MATLAB.

'stop'

Shuts down the MATLAB server without shutting down MATLAB. There is no difference between using 'kill' and 'stop'.

`'status'`

Displays status of the MATLAB server. You can also use `s=hdldaemon('status')`, which displays MATLAB server status and returns status in structure `s`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'time', 'int64', 'quiet', 'true'` specifies time values are returned as 64-bit integers and output messages are suppressed.

### **time** — Instruction to MATLAB server on how it should send and return time values

`'sec'` (default) | `'int64'`

Instruction to MATLAB server on how it should send and return time values, specified as the comma-separated pair consisting of `'time'` and one of these values:

- |                      |  |
|----------------------|--|
| <code>'int64'</code> | Specifies that the MATLAB server send and return time values in the MATLAB function callbacks as 64-bit integers representing the number of simulation steps.<br>See the <code>matlabcp/matlabtb_tnow</code> parameter reference (“MATLAB Function Syntax and Function Argument Definitions”). |
| <code>'sec'</code>   | Specifies that the MATLAB server sends and returns time values in the MATLAB function callbacks as <code>double</code> values that HDL Verifier scales to seconds based on the current HDL simulation resolution.  |

If server is already running, issuing `hdldaemon` with the `time` parameter alone will shut down the current server and start the server up again using shared memory.

Example: `'time', 'int64'`

### **quiet** — Indicator to suppress printing diagnostic messages

`'false'` (default) | `'true'`

Indicator to suppress printing diagnostic messages, specified as the comma-separated pair consisting of `'quiet'` and one of the following values:

---

|                      |   |
|----------------------|---|
| <code>'true'</code>  | Suppress printing diagnostic messages.        |
| <code>'false'</code> | Do not suppress printing diagnostic messages. |

Errors still appear. Use this option to suppress the MATLAB server shutdown message when using `hdldaemon` to get an unused socket number. If server is already running, issuing `hdldaemon` with the `quiet` parameter alone will shut down the current server and start the server up again using shared memory.

Example: `'quiet', 'true'`

### **socket** — TCP/IP port used for communication

0 | port number | character vector alias

TCP/IP port used for communication, specified as the comma-separated pair consisting of `'socket'` and a value. The value can be either 0, indicating that the host automatically chooses a valid TCP/IP port, an explicit port number ( $1024 < \text{port} < 49151$ ) or a service (alias) name from `/etc/services` file.

If you specify the operating system option (0), use `hdldaemon('status')` to acquire the assigned socket port number.

Example: `'socket',4449`

### **tclcmd** — Tcl command transmitted to all connected clients

character vector

Tcl command transmitted to all connected clients, specified as any valid Tcl command character vector.

The Tcl command character vector you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the character vector cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Incisive).

---

**Note** You can issue this command only after the software establishes a server connection.

---



---

**Caution** Do not call `hdldaemon('tclcmd', 'Tcl command')` from inside a `matlabtb` or `matlabcp` function. Doing so results in a race condition, and the simulator hangs.

---

Example: `'tclcmd','puts' '"done"'`

## Output Arguments

**s** — Structure containing information about the connection

`'comm' | 'connections' | 'ipc_id'`

Structure containing information about the connection. The structure contains the following variables:

|                            |  |
|----------------------------|--|
| <code>'comm'</code>        | Either <code>'shared memory'</code> or <code>'sockets'</code>  |
| <code>'connections'</code> | Number of open connections   |
| <code>'ipc_id'</code>      | If shared memory, file system name for the shared memory communication channel. If socket, the TCP/IP port number. |

## See Also

`nclaunch` | `vsim`

## Topics

Implementing the Filter Component of an Oscillator in MATLAB  
“Start the HDL Simulator from MATLAB”

**Introduced in R2008a**

# hdlsimmatlab

Load instantiated HDL design for verification with Cadence Incisive and MATLAB

## Syntax

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

## Description

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

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

This command is issued in the HDL simulator.

## 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 Cadence Incisive simulator documentation.

## Examples

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

```
tclshell> hdlstimmatlab work.parse
```

**Introduced in R2008a**

# hdlsimulink

Load instantiated HDL design for cosimulation with Cadence Incisive and Simulink

## Syntax

```
hdlsimulink <instance> [<ncsim_args>]
```

## Description

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

This command is issued in the HDL simulator. The communication mode is determined by the call to `nclaunch`, which must be issued before you call `hdlsimulink`.

## Argument

`<instance>`

Specifies the instance of an HDL design to load for cosimulation.

`<ncsim_args>`

Specifies one or more `ncsim` command arguments. Do not use `-GUI`, `-BATCH`, or `-TCL`. For more information on `ncsim` arguments, see the description of `ncsim` in the Cadence Incisive simulator documentation.

## Examples

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

```
tclshell> hdl simulink -gui work.parse
```

**Introduced in R2008a**



# matlabcp

Associate MATLAB component function with instantiated HDL design

## Syntax

```
matlabcp <instance>  
[<time-specs>]  
[-socket <tcp-spec>]  
[-rising <port>[,<port>...]]  
[-falling <port> [,<port>,...]]  
[-sensitivity <port>[,<port>,...]]  
[-mfunc <name>]  
[-use_instance_obj]  
[-argument]
```

## Description

The `matlabcp` command has the following characteristics:

- Starts the HDL simulator client component of the HDL Verifier software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous `matlabcp` command that specified the same instance. For example, if you issue the command `matlabcp` for instance `foo`, all previously scheduled events initiated by `matlabcp` on `foo` are canceled.

This command is issued in the HDL simulator.

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

---

**Notes** The communication mode that you specify for `matlabcp` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabcp` command in the HDL simulator.

---

## Arguments

`<instance>`

Specifies an instance of an HDL design that is associated with a MATLAB function. By default, `matlabcp` associates the instance to a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabcp` associates the instance with the MATLAB function `myfirfilter` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabcp` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

---

**Note** Do not specify an instance of an HDL module that has already been associated with a MATLAB function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

---

`<time-specs>`

Specifies a combination of time specifications consisting of any or all of the following:

|                |  |
|----------------|--|
| <timen>,...    | <p>Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space.</p> <p>For example:</p> <pre>matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec</pre> <p>The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero.</p> <hr/> <p><b>Note</b> 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.</p> |
| -repeat <time> | <p>Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified &lt;timen&gt;, . . . pattern. The time values are relative to the value of tnow at the time the HDL simulator first calls the MATLAB function.</p>   |
| -cancel <time> | <p>Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of tnow at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a nomatlabtb call.</p> <hr/> <p><b>Note</b> The -cancel option works only with the &lt;time-specs&gt; arguments. It does not affect any of the other scheduling arguments for matlabcp.</p>  |

---

**Note** Place time specifications after the matlabcp instance and before any additional command arguments; otherwise the time specifications are ignored.

---

All time specifications for the `matlabcp` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies that HDL Verifier use TCP/IP sockets to communicate between the HDL simulator and MATLAB. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp_spec>` on the command line. The communication mode that you specify with the `matlabcp` command must match the communication mode that you issued with the `hdldaemon` command.

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (STD\_LOGIC, BIT, X01, and so on).

For determining signal transition in:

- VHDL: Rising edge is {0 or L} to {1 or H}.
- Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.

---

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

---

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a falling edge—changes from '1' to '0'. Specify -

falling with the path names of one or more signals defined as a logic type (STD\_LOGIC, BIT, X01, and so on).

For determining signal transition in:

- VHDL: Falling edge is {1 or H} to {0 or L}.
- Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.

---

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

---

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level in the HDL model hierarchy.

---

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

---

`-mfunc <name>`

The name of the MATLAB function that is associated with the HDL module instance you specify for `instance`. By default, the HDL Verifier software invokes a MATLAB function that has the same name as the specified HDL instance. Thus, if the names are the same, you can omit the `-mfunc` option. If the names are not the same, use this argument when you call `matlabcp`. If you omit this argument and `matlabcp` does not find a MATLAB function with the same name, the command generates an error message.

`-use_instance_obj`

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by HDL Verifier to the function. This argument has the fields shown in the following table. See “Writing Functions Using the HDL Instance Object” for examples.

| Field     | Read/Write Access | Description   |
|-----------|-------------------|---|
| tnext     | Write only        | Used to schedule a callback during the set time value. This field is equivalent to old tnext. For example:<br><br><pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</pre> <p>will schedule a callback at time equals 5 nanoseconds from tnow.</p>  |
| userdata  | Read/Write        | Stores state variables of the current matlabcp instance. You can retrieve the variables the next time the callback of this instance is scheduled.   |
| simstatus | Read only         | Stores the status of the HDL simulator. The HDL Verifier software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. simstatus is a read-only property.<br><br><pre>&gt;&gt; hdl_instance_obj.simstatus</pre> <pre>ans=</pre> <pre>    Init</pre>  |
| instance  | Read only         | Stores the full path of the Verilog/VHDL instance associated with the callback. instance is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:<br><br><p>In the HDL simulator:</p> <pre>hdlsim&gt; matlabcp osc_top -mfunc oscfilter use_instance_obj</pre> <p>In MATLAB:</p> <pre>&gt;&gt; hdl_instance_obj.instance</pre> <pre>ans=</pre> <pre>    osc_top</pre> |

| Field    | Read/Write Access | Description  |
|----------|-------------------|--|
| argument | Read only         | <p>Stores the argument set by the <code>-argument</code> option of <code>matlabcp</code>. For example:</p> <pre>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</pre> <p>The link software supports the <code>-argument</code> option only when it is used with <code>-use_instance_obj</code>, otherwise the argument is ignored. <code>argument</code> is a read-only property.</p> <pre>&gt;&gt; hdl_instance_obj.argument ans=     foo</pre>   |
| portinfo | Read only         | <p>Stores information about the VHDL and Verilog ports associated with this instance. <code>portinfo</code> is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information".</p> <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <hr/> <p><b>Note</b> When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p> |

| Field      | Read/Write Access | Description   |
|------------|-------------------|---|
| tscale     | Read only         | <p>Stores the resolution limit (tick) in seconds of the HDL simulator. <code>tscale</code> is a read-only property.</p> <pre>&gt;&gt; hdl_instance_obj.tscale  ans=     1.0000e-009</pre> <p><b>Note</b> When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p> |
| tnow       | Read only         | <p>Stores the current time. <code>tnow</code> is a read-only property.</p> <pre>hdl_instance_obj.tnext = hld_instance_obj.tnow + fastestrate;</pre>   |
| portvalues | Read/Write        | <p>Stores the current values of and sets new values for the output and input ports for a <code>matlabcp</code> instance. For example:</p> <pre>&gt;&gt; hdl_instance_obj.portvalues  ans = Read Only Input ports:     clk_enable: []          clk: []          reset: [] Read/Write Output ports:     sine_out: [22x1 char]</pre>   |
| linkmode   | Read only         | <p>Stores the status of the callback. The HDL Verifier software sets this field to <code>'testbench'</code> if the callback is associated with <code>matlabtb</code> and <code>'component'</code> if the callback is associated with <code>matlabcp</code>. <code>linkmode</code> is a read-only property.</p> <pre>&gt;&gt; hdl_instance_obj.linkmode  ans=     component</pre>  |



-argument

Used to pass user-defined arguments from the `matlabcp` invocation on the HDL side to the MATLAB function callbacks. Supported with `-use_instance_obj` only. See the field listing under the `-use_instance_obj` property.

## Examples

The following examples demonstrate some ways you might use the `matlabcp` function.

### Using `matlabcp` with the `-mfunc` option to Associate an HDL Component with a MATLAB Function of a Different Name

This example explicitly associates the Verilog module `vlogtestbench_top.u_matlab_component` with the MATLAB function `vlogmatlabcp` using the `-mfunc` option. The `'-socket'` option specifies using socket communication on port 4449.

```
hdlsim>matlabcp vlogtestbench_top.u_matlab_component -mfunc vlogmatlabcp -socket 4449
```

### Using `matlabcp` with Explicit Times and the `-cancel` Option

This example includes explicit times with the `-cancel` option.

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

### Using `matlabcp` with Rising and Falling Edges

This example implicitly associates the Verilog module, `vlogtestbench_top`, with the MATLAB function `vlogtestbench_top`, and also uses rising and falling edges.

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

**Introduced in R2008a**

## matlabtb

Schedule MATLAB test bench session for instantiated HDL module

### Syntax

```
matlabtb <instance>  
[<time-specs>]  
[-socket <tcp-spec>]  
[-rising <port>[,<port>...]]  
[-falling <port> [,<port>,...]]  
[-sensitivity <port>[,<port>,...]]  
[-mfunc <name>]  
[-use_instance_obj]  
[-argument]
```

### Description

The `matlabtb` command has the following characteristics:

- Starts the HDL simulator client component of the HDL Verifier software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Creates a process that schedules invocations of the specified MATLAB function.
- Cancels any pending events scheduled by a previous `matlabtb` command that specified the same instance. For example, if you issue the command `matlabtb` for instance `foo`, all previously scheduled events initiated by `matlabtb` on `foo` are canceled.

This command is issued in the HDL simulator.

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

**Notes** The communication mode that you specify for `matlabtb` must match the communication mode you specified for `hdldaemon` when you established the server connection.

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabtb` command in the HDL simulator.

---

## Arguments

<instance>

Specifies the instance of an HDL module that the HDL Verifier software associates with a MATLAB test bench function. By default, `matlabtb` associates the instance with a MATLAB function that has the same name as the instance. For example, if the instance is `myfirfilter`, `matlabtb` associates the instance with the MATLAB function `myfirfilter` (note that hierarchy names are ignored; for example, if your instance name is `top.myfirfilter`, `matlabtb` would associate only `myfirfilter` with the MATLAB function). Alternatively, you can specify a different MATLAB function with `-mfunc`.

---

**Note** Do not specify an instance of an HDL module that has already been associated with a MATLAB function (via `matlabcp` or `matlabtb`). If you do, the new association overwrites the existing one.

---

<time-specs>

Specifies a combination of time specifications consisting of any or all of the following:

|                             |  |
|-----------------------------|--|
| <p>&lt;timen&gt;,...</p>    | <p>Specifies one or more discrete time values at which the HDL simulator calls the specified MATLAB function. Each time value is relative to the current simulation time. Even if you do not specify a time, the HDL simulator calls the MATLAB function once at the start of the simulation. Separate multiple time values by a space.</p> <p>For example:</p> <pre>matlabtb vlogtestbench_top 10 ns, 10 ms, 10 sec</pre> <p>The MATLAB function executes when time equals 0 and then 10 nanoseconds, 10 milliseconds, and 10 seconds from time zero.</p> <hr/> <p><b>Note</b> 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.</p> |
| <p>-repeat &lt;time&gt;</p> | <p>Specifies that the HDL simulator calls the MATLAB function repeatedly based on the specified &lt;timen&gt;, ... pattern. The time values are relative to the value of tnow at the time the HDL simulator first calls the MATLAB function.</p> <p>For example:</p> <pre>matlabtb vlogtestbench_top 5 ns -repeat 10 ns</pre> <p>The MATLAB function executes at time equals 0 ns, 5 ns, 15 ns, 25 ns, and so on.</p>  |
| <p>-cancel &lt;time&gt;</p> | <p>Specifies a time at which the specified MATLAB function stops executing. The time value is relative to the value of tnow at the time the HDL simulator first calls the MATLAB function. If you do not specify a cancel time, the application calls the MATLAB function until you finish the simulation, quit the session, or issue a nomatlabtb call.</p> <hr/> <p><b>Note</b> The -cancel option works only with the &lt;time-specs&gt; arguments. It does not affect any of the other scheduling arguments for matlabtb.</p>  |

---

**Note** Place time specifications after the `matlabtb` instance and before any additional command arguments; otherwise the time specifications are ignored.

---

All time specifications for the `matlabtb` functions appear as a number and, optionally, a time unit:

- fs (femtoseconds)
- ps (picoseconds)
- ns (nanoseconds)
- us (microseconds)
- ms (milliseconds)
- sec (seconds)
- no units (tick)

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. When you provide TCP/IP information for `matlabtb`, you can choose a TCP/IP port number or TCP/IP port alias or service name for the `<tcp_spec>` parameter. If you are setting up communication between computers, you must also specify the name or Internet address of the remote host that is running the MATLAB server (`hdldaemon`).

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp_spec>` on the command line.

---

**Note** The communication mode that you specify with the `matlabtb` command must match what you specify for the communication mode when you issue the `hdldaemon` command in MATLAB. For more information on modes of communication, see “Communications for HDL Cosimulation”. For more information on establishing the MATLAB end of the communication link, see “Start the HDL Simulator from MATLAB”.

---

`-rising <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function on the rising edge (transition from '0' to '1') of any of the specified signals. Specify `-rising` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Rising edge is {0 or L} to {1 or H}.
- Verilog: Rising edge is the transition from 0 to x, z, or 1, and from x or z to 1.

---

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

---

`-falling <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals experiences a falling edge—changes from '1' to '0'. Specify `-falling` with the path names of one or more signals defined as a logic type (`STD_LOGIC`, `BIT`, `X01`, and so on).

For determining signal transition in:

- VHDL: Falling edge is {1 or H} to {0 or L}.
- Verilog: Falling edge is the transition from 1 to x, z, or 0, and from x or z to 0.

---

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

---

`-sensitivity <signal>[, <signal>...]`

Indicates that the application calls the specified MATLAB function whenever any of the specified signals changes state. Specify `-sensitivity` with the path names of one or more signals. Signals of any type can appear in the sensitivity list and can be positioned at any level of the HDL design.

If you specify the option with no signals, the interface is sensitive to value changes for all signals.

---

**Note** Use of this option for INOUT ports can result in double calls.

---

For example:

```
-sensitivity /randnumgen/dout
```

The MATLAB function executes if the value of `dout` changes.

---

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

---

`-mfunc <name>`

The name of the associated MATLAB function. If you omit this argument, `matlabtb` associates the HDL module instance to a MATLAB function that has the same name as the HDL instance. If you omit this argument and `matlabtb` does not find a MATLAB function with the same name, the command generates an error message.

`-use_instance_obj`

Instructs the function specified with the argument `-mfunc` to use an HDL instance object passed by HDL Verifier to the function. This argument has the fields shown in the following table. See “Writing Functions Using the HDL Instance Object” for examples.

| Field              | Read/Write Access | Description  |
|--------------------|-------------------|--|
| <code>tnext</code> | Write only        | Used to schedule a callback during the set time value. This field is equivalent to old <code>tnext</code> . For example:<br><br><pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + 5e-9</pre><br>will schedule a callback at time equals 5 nanoseconds from <code>tnow</code> . |

| Field     | Read/Write Access | Description  |
|-----------|-------------------|--|
| userdata  | Read/Write        | Stores state variables of the current <code>matlabcp</code> instance. You can retrieve the variables the next time the callback of this instance is scheduled.   |
| simstatus | Read only         | Stores the status of the HDL simulator. The HDL Verifier software sets this field to 'Init' during the first callback for this particular instance and to 'Running' thereafter. <code>simstatus</code> is a read-only property.<br><br><pre>&gt;&gt; hdl_instance_obj.simstatus ans=     Init</pre>  |
| instance  | Read only         | Stores the full path of the Verilog/VHDL instance associated with the callback. <code>instance</code> is a read-only property. The value of this field equals that of the module instance specified with the function call. For example:<br><br><b>In the HDL simulator:</b><br><pre>hdlsim&gt; matlabcp osc_top -mfunc oscfilter use_instance_obj</pre> <b>In MATLAB:</b><br><pre>&gt;&gt; hdl_instance_obj.instance ans=     osc_top</pre>                     |
| argument  | Read only         | Stores the argument set by the <code>-argument</code> option of <code>matlabcp</code> . For example:<br><pre>matlabtb osc_top -mfunc oscfilter -use_instance_obj -argument foo</pre> The link software supports the <code>-argument</code> option only when it is used with <code>-use_instance_obj</code> , otherwise the argument is ignored. <code>argument</code> is a read-only property.<br><br><pre>&gt;&gt; hdl_instance_obj.argument ans=     foo</pre> |



| Field    | Read/Write Access | Description  |
|----------|-------------------|--|
| portinfo | Read only         | <p>Stores information about the VHDL and Verilog ports associated with this instance. <code>portinfo</code> is a read-only property, which has a field structure that describes the ports defined for the associated HDL module. For each port, the <code>portinfo</code> structure passes information such as the port's type, direction, and size. For more information on port data, see "Gaining Access to and Applying Port Information".</p> <pre>hdl_instance_obj.portinfo.field1.field2.field3</pre> <p><b>Note</b> When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p> |
| tscale   | Read only         | <p>Stores the resolution limit (tick) in seconds of the HDL simulator. <code>tscale</code> is a read-only property.</p> <pre>&gt;&gt; hdl_instance_obj.tscale  ans=     1.0000e-009</pre> <p><b>Note</b> When you use <code>use_instance_obj</code>, you access <code>tscale</code> through the HDL instance object. If you do not use <code>use_instance_obj</code>, you can still access <code>tscale</code> through <code>portinfo</code>.</p>  |
| tnow     | Read only         | <p>Stores the current time. <code>tnow</code> is a read-only property.</p> <pre>hdl_instance_obj.tnext = hdl_instance_obj.tnow + fastestrate;</pre>  |

| Field      | Read/Write Access | Description   |
|------------|-------------------|---|
| portvalues | Read/Write        | Stores the current values of and sets new values for the output and input ports for a <code>matlabcp</code> instance. For example:<br><br><pre>&gt;&gt; hdl_instance_obj.portvalues  ans = Read Only Input ports:   clk_enable: []   clk: []   reset: [] Read/Write Output ports:   sine_out: [22x1 char]</pre>                                       |
| linkmode   | Read only         | Stores the status of the callback. The HDL Verifier software sets this field to 'testbench' if the callback is associated with <code>matlabtb</code> and 'component' if the callback is associated with <code>matlabcp</code> . <code>linkmode</code> is a read-only property.<br><br><pre>&gt;&gt; hdl_instance_obj.linkmode  ans=   component</pre> |

-argument

Used to pass user-defined arguments from the `matlabtb` instantiation on the HDL side to the MATLAB function callbacks. Supported with `-use_instance_obj` only. See the field listing for argument under the `-use_instance_obj` property.

## Examples

The following examples demonstrate some ways you might use the `matlabtb` function.

### Using `matlabtb` with the `-socket` Argument and Time Parameters

The following command starts the HDL simulator client component of HDL Verifier, associates an instance of the entity, `myfirfilter`, with the MATLAB function `myfirfilter`, and begins a local TCP/IP socket-based test bench session using TCP/IP

port 4449. Based on the specified test bench stimuli, `myfirfilter.m` executes 5 nanoseconds from the current time, and then repeatedly every 10 nanoseconds:

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

## Applying Rising Edge Clocks and State Changes with `matlabtb`

The following command starts the HDL simulator client component of HDL Verifier, and begins a remote TCP/IP socket-based session using remote MATLAB host computer named `computer123` and TCP/IP port 4449. Based on the specified test bench stimuli, `myfirfilter.m` executes 10 nanoseconds from the current time, each time the signal `/top/fclk` experiences a rising edge, and each time the signal `/top/din` changes state.

```
hdlsim> matlabtb /top/myfirfilter 10 ns -rising /top/fclk -sensitivity /top/din  
-socket 4449@computer123
```

## Specifying a MATLAB Function Name and Sensitizing Signals with `matlabtb`

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

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

**Introduced in R2008a**

## matlabtbeval

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

### Syntax

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

### Description

The `matlabtbeval` command has the following characteristics:

- Starts the HDL simulator client component of the HDL Verifier software.
- Associates a specified instance of an HDL design created in the HDL simulator with a MATLAB function.
- Executes the specified MATLAB function once and immediately on behalf of the specified module instance.

This command is issued in the HDL simulator.

---

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

---

---

**Notes** The communication mode that you specify for `matlabtbeval` must match the communication mode you specified for `hdldaemon` when you established the server connection.

---

For socket communications, specify the port number you selected for `hdldaemon` when you issue a link request with the `matlabtbeval` command in the HDL simulator.

---

## Arguments

<instance>

Specifies the instance of an HDL module that is associated with a MATLAB function. By default, `matlabtbeval` associates the HDL module instance with a MATLAB function that has the same name as the HDL module instance. For example, if the HDL module instance is `myfirfilter`, `matlabtbeval` associates the HDL module instance with the MATLAB function `myfirfilter`. Alternatively, you can specify a different MATLAB function with the `-mfunc` property.

`-socket <tcp_spec>`

Specifies TCP/IP socket communication for the link between the HDL simulator and MATLAB. For TCP/IP socket communication on a single computer, the `<tcp_spec>` can consist of just a TCP/IP port number or service name (alias). If you are setting up communication between computers, you must also specify the name or Internet address of the remote host.

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”.

If you run the HDL simulator and MATLAB on the same computer, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

---

**Note** The communication mode that you specify with the `matlabtbeval` command must match what you specify for the communication mode when you call the `hdldaemon` command to start the MATLAB server. For more information on communication modes, see “Communications for HDL Cosimulation”. For more information on establishing the MATLAB end of the communication link, see “Start the HDL Simulator from MATLAB”.

---

`-mfunc <name>`

The name of the associated MATLAB function. If you omit this argument, `matlabtbeval` associates the HDL module instance with a MATLAB function that has the same name as the HDL module instance. If you omit this argument and `matlabtbeval` does not find a MATLAB function with the same name, the command displays an error message.

### Examples

This example starts the HDL simulator client component of the link software, associates an instance of the module `myfirfilter` with the function `myfirfilter.m`, and uses a local TCP/IP socket-based communication link to TCP/IP port 4449 to execute the function `myfirfilter.m`:

```
hdlsim> matlabtbeval myfirfilter -socket 4449:
```

**Introduced in R2008a**

# mvl2dec

Convert multivalued logic to decimal

## Syntax

```
mvl2dec('mv_logic_char')  
mvl2dec('mv_logic_char', signed)
```

## Description

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

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

## Examples

### Convert Binary Strings to Decimal Integers

The `mvl2dec` function accepts strings of binary digits and returns decimal integer values.

```
mvl2dec('010111')
```

```
ans =
```

```
23
```

The following function call returns NaN because the binary digits are not either 1s and 0s.

```
mv12dec('xxxxxx')
```

```
ans =
```

```
NaN
```

The second argument indicates that the string is a signed number.

```
mv12dec('10111',true)
```

```
ans =
```

```
-9
```

## See Also

[dec2mv1](#)

**Introduced in R2008a**



# nclaunch

Start and configure Cadence Incisive simulators for use with HDL Verifier software

## Syntax

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

## Description

`nclaunch('PropertyName','PropertyValue'...)` starts the Cadence Incisive simulator for use with the MATLAB and Simulink features of the HDL Verifier software. The first folder in the Cadence Incisive simulator matches your MATLAB current folder if you do not specify an explicit `rundir` parameter.

After you call this function, you can use HDL Verifier functions for the HDL simulator (for example, `hdlsimmatlab`, `hdlsimulink`) to do interactive debug setup.

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

## Name-Value Pair Arguments

### `hdlsimdir`

Specifies the path name to the Cadence Incisive simulator executable to be started.

- `pathname`

Start a different version of the Cadence Incisive simulator or if the version of the simulator you want to run does not reside on the system path.

**Default:** The first version of the simulator that the function finds on the system path.

**hdlsimexe**

Specifies the name of a Cadence Incisive simulator executable.

- `simexename`

Custom-built simulator executable.

**Default:** `ncsim`

**libdir**

This property creates an entry in the startup Tcl file that points to the folder with the shared libraries for the Cadence Incisive simulator to communicate with MATLAB when the Cadence Incisive simulator runs on a machine that does not have MATLAB.

- `folder`

Folder containing MATLAB shared libraries.

**libfile**

Specifies the library file to use for HDL simulation. If the HDL simulator links other libraries, including SystemC libraries, that were built using a compiler supplied with the HDL simulator, you can specify an alternate library file with this property. See “Cosimulation Libraries” for versions of the library built using other compilers.

- `library_file_name`

The particular library file to use for HDL simulation.

**Default:** The version of the library file that was built using the same compiler that MATLAB itself uses.

**rundir**

Specifies the folder containing the HDL simulator executable.

- `dirname`

Where to run the HDL simulator.

The following conditions apply to this name/value pair:

- If the value of `dirname` is “TEMPDIR”, the function creates a temporary folder in which it runs the HDL simulator.
- If you specify `dirname` and the folder does *not* exist, you will get an error.

**Default:** The current working folder

### **runmode**

Specifies how to start the HDL simulator.

- `mode`

This property accepts the following valid values:

- 'Batch': Start the HDL simulator in the background with no window.
- 'Batch with Xterm': Run HDL simulator in a non-interactive Xterm window.
- 'CLI': Start the HDL simulator in an interactive terminal window.
- 'GUI': Start the HDL simulator with the graphical user interface.

**Default:** 'GUI'

### **socketsimulink**

Specifies TCP/IP socket communication between the Cadence Incisive simulator and Simulink. For shared memory, omit `-socket <tcp-spec>` on the command line.

- `tcp_spec`

TCP/IP port number or service name (alias)

**Default:** Shared memory

### **starthdlsim**

Determines whether the Cadence Incisive simulator is launched.

This function creates a startup Tcl file which contains pointers to MATLAB and Simulink shared libraries. To run the Cadence Incisive simulator manually, see “Start the HDL Simulator from MATLAB”.

- `yes`

Launches the Cadence Incisive simulator and creates a startup Tcl file.

- no

Does not launch the Cadence Incisive simulator , but still creates a startup Tcl file.

**Default:** yes

#### **startupfile**

Specify the name and location of the Tcl script generated by `nclaunch`. The generated Tcl script, when executed, compiles and launches the HDL simulator. You can edit and use the generated file in a regular shell outside of MATLAB. For example:

```
sh> tclsh compile_and_launch.tcl
```

- pathname

Filename and path for generated Tcl script. If the file name already exists on the specified path, that file's contents are overwritten.

**Default:** Generates a filename of `compile_and_launch.tcl` in the folder specified by `rundir`.

#### **tclstart**

Specifies one or more Tcl commands to execute before the Cadence Incisive simulator launches. You must specify at least one command; otherwise, no action occurs.

- tcl\_commands

A command character vector or a cell array of commands.

---

**Note** You must type `exec` in front of non-Tcl system shell commands. For example:

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

---

## Examples

## Start Cosimulation Session with Simulink

Compile design and start Simulink.

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

In this example, nclaunch performs the following:

- Compiles the design top.v: `exec ncverilog -c +access+rw +linedebug top.v`.
- Starts Simulink with the GUI from the `proj` folder with the model loaded: `hdlsimulink -gui work.top` and `'rundir', '/proj'`.
- Instructs Simulink to communicate with the HDL Verifier interface on socket port 4449: `'socketsimulink','4449'`.

All of these commands are specified in a single character vector as the property value to `tclstart`.

## Create Tcl Script to Start HDL Simulator

Create a Tcl script to start the HDL simulator from a Tcl shell using nclaunch.

Specify the name of the Tcl script and the command(s) it includes as parameters to nclaunch:

```
nclaunch('tclstart','xxx','startupfile','mytclscript','starthdlsim','yes')
```

In this example, a Tcl script is created and the command to start the HDL simulator is included. The startup Tcl file is named "mytclscript".

Execute the script in a Tcl shell:

```
shell> tclsh mytclscript
```

This starts the HDL simulator.

## Execute Multiple Tcl Commands When Launching Cosimulation Connection

Build a sequence of Tcl commands that are then executed in a Tcl shell, after calling `nclaunch` from MATLAB.

Assign Tcl command values to the `tclcmd` parameter of `nclaunch`:

```
tclcmd{1} = 'exec ncvtlog 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}""']

tclcmd =

    'exec ncvtlog vlogtestbench_top.v'    'exec ncelab -access +wc vlogtestbench_top'

tclcmd =

    'exec ncvtlog vlogtestbench_top.v'    'exec ncelab -access +wc vlogtestbench_top'

tclcmd =

    [1x31 char]    [1x41 char]    [1x145 char]
```

- `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.

Issue the `nclaunch` command, passing the `tclcmd` variable just set:

```
nclaunch('hdlsimdir','local.IUS.glnx.tools.bin','tclstart',tclcmd);
```

In this example, the `nclaunch` launches the following tasks through the Tcl commands assigned in `tclcmd`:

- Executes the arguments being passed with `-input` (`matlabtb` and `run`) in the `ncsim` Tcl shell.
- Issues a call to `matlabcp`, which associates the function `vlogmatlabc` to the module instance `u_matlab_component`.
- Assumes that the `hdldaemon` in MATLAB is listening on port 32864

- Instructs the `run` function to run 50 resolution units (ticks).

**Introduced in R2008a**

# nomatlabtb

End active MATLAB test bench and MATLAB component sessions

## Syntax

```
nomatlabtb
```

## Description

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

This command is issued in the HDL simulator.

---

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

---

## Examples

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

```
hdlsim> nomatlabtb
```

## See Also

`matlabcp` | `matlabtb`

**Introduced in R2008a**



# notifyMatlabServer

Send HDL simulator event and process IDs to MATLAB server

## Syntax

```
notifyMatlabServer EventID -socket tcp-spec
```

## Description

`notifyMatlabServer EventID -socket tcp-spec` sends the HDL simulator event ID and process identification (PID) to the MATLAB server (`hdldaemon`) using the specified connection methods (socket or shared memory). For MATLAB to receive this message, `hdldaemon` must be running with the same communication mode as specified with the `notifyMatlabServer` command. The event ID and the PID queue in `hdldaemon.notifyMatlabServer` is often used in conjunction with `waitForHdlClient` to make sure the HDL simulator is ready to begin or continue processing.

This command issues in the HDL simulator.

## Input Arguments

### **EventID**

Specifies the event ID to be sent to `hdldaemon`. The ID requires a positive number less than the maximum value of 32-bit signed integer. This parameter contains the event ID expected by the command `waitForHdlClient` in MATLAB.

**Default:** 1

### **socket tcp\_spec**

Specifies that TCP/IP socket communication be used for the link between the HDL simulator and MATLAB. For TCP/IP socket communication on a single computer,

*tcp\_spec* requires either a TCP/IP port number or service name (alias). To set up communication between computers, you must also specify the name or Internet address of the remote host that is running the MATLAB server (*hdldaemon*).

When you omit the *socket* option, MATLAB and the HDL simulator use shared memory communication.

## Examples

In MATLAB, use the function *waitForHdlClient* to verify whether the HDL simulator event ID has been received. In the following example, the function returns the HDL Simulator PID if *EventID* = 5 is received within 100 seconds. If a time-out occurs, the function returns -1.

```
>> hlddaemon('socket',5002);  
...  
>> hdlpid = waitForHdlClient(100,5);
```

In the HDL simulator, issue the *notifyMatlabServer* command to send event ID 5 to *hdldaemon* running on the same machine using TCP/IP socket port 5002.

```
>> notifyMatlabServer 5 -socket 5002
```

## See Also

*waitForHdlClient*

**Introduced in R2012b**

# pingHdlSim

Block cosimulation until HDL simulator is ready for simulation

## Syntax

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

## Description

`pingHdlSim(timeout)` blocks cosimulation by not returning until the HDL server loads or until the specified time-out occurs. `pingHdlSim` returns the process ID of the HDL simulator or -1 if a time-out occurs. You must enter a time-out value. You may find this function useful if you are trying to automate a cosimulation and need to know that the HDL server has loaded before your script continues the simulation.

`pingHdlSim(timeout, portnumber)` tries to connect to the local host on port `portnumber` and times out after `timeout` seconds you specify.

`pingHdlSim(timeout, portnumber, hostname)` tries to connect to the host `hostname` on port `portnumber`. It times out after `timeout` seconds you specify.

## Examples

The following function call blocks further cosimulation until the HDL server loads or until 30 seconds have passed:

```
>>pingHdlSim(30)
```

If the server loads within 30 seconds, `pingHdlSim` returns the process ID. If it does not, `pingHdlSim` returns -1.

The following function call blocks further cosimulation on port 5678 until the HDL server loads or until 20 seconds have passed:

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

The following function call blocks further cosimulation on port 5678 on host name `msuser` until the HDL server loads or until 20 seconds pass:

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

**Introduced in R2008a**

## tclHdlSim

Execute Tcl command in Incisive or ModelSim simulator

### Syntax

```
tclHdlSim(tclCmd)
tclHdlSim(tclCmd,portNumber)
tclHdlSim(tclCmd, portname, hostname)
```

### Description

`tclHdlSim(tclCmd)` executes a Tcl command on the Incisive or ModelSim simulator using a shared connection during a Simulink cosimulation session.

`tclHdlSim(tclCmd,portNumber)` executes a Tcl command on the Incisive or ModelSim simulator by connecting to the local host on port `portNumber`.

`tclHdlSim(tclCmd, portname, hostname)` executes a Tcl command on the Incisive or ModelSim simulator by connecting to the host `hostname` on port `portname`.

The Incisive or ModelSim simulator must be connected to MATLAB and Simulink using the HDL Verifier software for this function to work (see either `vsimulink` or `hdlsimulink`).

You may specify any valid Tcl command. The Tcl command you specify cannot include commands that load an HDL simulator project or modify simulator state. For example, the character vector cannot include commands such as `start`, `stop`, or `restart` (for ModelSim) or `run`, `stop`, or `reset` (for Incisive).

To execute a Tcl command on the Incisive or ModelSim simulator during a MATLAB cosimulation session, use `hdldaemon('tclcmd','command')`.

### Examples

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

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

### See Also

hdldaemon | nclaunch | vsim

**Introduced in R2008a**

## vsim

Start and configure ModelSim for use with HDL Verifier

### Syntax

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

### Description

`vsim('PropertyName', 'PropertyValue'...)` starts and configures the ModelSim simulator (`vsim`) for use with the MATLAB and Simulink features of HDL Verifier. The first folder in ModelSim matches your MATLAB current folder.

`vsim` creates a startup (or `.do`) file that adds the following Tcl commands to ModelSim:

- `vsimmatlab`: link to MATLAB from ModelSim
- `vsimulink`: link to Simulink from ModelSim
- `vmatlabsysobj`: link to MATLAB System object from ModelSim.

You can use these new ModelSim commands in place of the ModelSim `vsim` command. These commands are used to load instances of VHDL entities or Verilog modules for simulations that use MATLAB or Simulink for verification

The property name/property value pair settings allow you to customize the Tcl commands used to start ModelSim, the `vsim` executable to be used, the path and name of the DO file that stores the start commands, and for Simulink applications, details about the mode of communication to be used by the applications.

---

**Tip** Use `pingHdlSim` to add a pause between the call to `vsim` and the call to actually run the simulation when you are attempting to automate the cosimulation.

---

## Property Name/Property Value Pairs

### **libdir**

Specifies the path to HDL Verifier HDL libraries

- `folder`

Folder containing the libraries for ModelSim to communicate with MATLAB when ModelSim runs on a machine that does not have MATLAB.

If this property is not set, the default path in the MATLAB installation is used.

### **libfile**

Specifies a particular library file

- `library_file_name`

. This value defaults to the version of the library file that was built using the same compiler that MATLAB itself uses. If the HDL simulator links other libraries, including SystemC libraries, that were built using a compiler supplied with the HDL simulator, you can specify an alternate library file with this property. See “Cosimulation Libraries” for versions of the library built using other compilers.

Do not include the OS-specific library extension in `library_file_name`.

### **pingTimeout**

Time to wait, in seconds, for the HDL simulator to start.

- `seconds`

Specify 0 (the default) to immediately return without waiting.

### **rundir**

Specifies where to run the HDL simulator

- `dirname`

By default, the function uses the current working folder.



The following conditions apply to this name/value pair:

- If the value of `dirname` is “TEMPDIR”, the function creates a temporary folder in which it runs ModelSim.
- If you specify `dirname` and the folder does *not* exist, you will get an error.

### **runmode**

Specifies how to start the HDL simulator.

- `mode`

You can set run mode to one of the following values:

- 'Batch': Start the HDL simulator in the background with no window (Linux®) or in a non-interactive command window (Windows).
- 'CLI': Start the HDL simulator in an interactive terminal window.
- 'GUI': Start the HDL simulator with the ModelSim graphical user interface.

This value defaults to 'GUI'.

### **socketmatlabsobj**

Specifies TCP/IP socket communication for links between ModelSim and MATLAB.

- `tcp_spec`

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.

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”

If ModelSim and MATLAB run on the same computing system, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

---

**Note** The function applies the communication mode specified by this property to all invocations of MATLAB from ModelSim.

---

#### **socketsimulink**

Specifies TCP/IP socket communication for links between ModelSim and Simulink.

- `tcp_spec`

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.

For more information on choosing TCP/IP socket ports, see “TCP/IP Socket Ports”

If ModelSim and Simulink run on the same computing system, you have the option of using shared memory for communication. Shared memory is the default mode of communication and takes effect if you do not specify `-socket <tcp-spec>` on the command line.

---

**Note** The function applies the communication mode specified by this property to all invocations of Simulink from ModelSim.

---

#### **startms**

Determines whether ModelSim is launched from `vsim`.

- `yes | no`

This property defaults to `yes`, which launches ModelSim and creates a startup Tcl file. If `startms` is set to `no`, ModelSim does not launch, but the HDL simulator still creates a startup Tcl file.

This startup Tcl file contains pointers to MATLAB libraries. To run ModelSim on a machine without MATLAB, copy the startup Tcl file and MATLAB library files to the remote machine and start ModelSim manually. See “Cosimulation Libraries”.

#### **startupfile**

Specifies Tcl script for startup

- `pathname`

Each invocation of `vsim` creates a Tcl script that is applied during HDL simulator startup. By default, this function generates the filename of

`compile_and_launch.tcl` in the folder specified by `rundir`. With this property, you can specify the name and location of the generated Tcl script. If the file name already exists, that file's contents are overwritten. You can edit and use the generated file in a regular shell outside of MATLAB. For example:

```
sh> vsim -gui -do compile_and_launch.tcl
```

### **tclstart**

Specifies one or more Tcl commands to execute during ModelSim startup

- `tcl_commands`

Specify a command character vector or a cell array of commands, with each entry being a Tcl command. These commands are appended to the startup file.

### **vsimdir**

Specifies the path name to the folder with the ModelSim simulator executable (`vsim.exe`) to be started.

- `pathname`

By default, the function uses the first version of `vsim.exe` that it finds on the system path (defined by the `path` variable) . Use this option to start different versions of the ModelSim simulator or if the version of the simulator you want to run does not reside on the system path.

## **Examples**

The following function call sequence changes the folder location to `VHDLproj` and then calls the function `vsim`. Because the call to `vsim` omits the `'vsimdir'` and `'startupfile'` properties, `vsim` uses the default `vsim` executable and creates a temporary DO file in a temporary folder. The `'tclstart'` property specifies a Tcl command that loads an instance of a VHDL entity for MATLAB verification:

- The `vsimmatlab` command loads an instance of the VHDL entity `parse` in the library `work` for MATLAB verification.

- The `matlabtb` command begins the test bench session for an instance of entity `parse`, using TCP/IP socket communication on port 4449 and a test bench timing value of 10 ns.

```
>>cd VHDLproj % Change folder to ModelSim project folder
>>vsim('tclstart','vsimmatlab work.parse; matlabtb parse 10 ns -socket 4449')
```

The following function call sequence changes the folder location to `VHDLproj` and then calls the function `vsim`.

- Because the call to `vsim` omits the `'vsimdir'` and `'startupfile'` properties, `vsim` uses the default `vsim` executable and creates a DO file in a temporary folder.
- The `'tclstart'` property specifies a Tcl command that loads the VHDL entity `parse` in the library `work` for cosimulation between `vsim` and Simulink.
- The `'socketsimulink'` property specifies that TCP/IP socket communication on the same computer is to be used for links between Simulink and ModelSim, using socket port 4449.

```
>>cd VHDLproj % Change folder to ModelSim project folder
>>vsim('tclstart','vsimulink work.parse','socketsimulink','4449')
```

**Introduced in R2008a**

## vsimmatlab

Load instantiated HDL module for verification with ModelSim and MATLAB

### Syntax

```
vsimmatlab <instance> [<vsim_args>]
```

### Description

The `vsimmatlab` command loads the specified instance of an HDL module for verification and sets up ModelSim so it can establish a communication link with MATLAB. ModelSim opens a simulation workspace and displays a series of messages in the command window as it loads the HDL module's packages and architectures.

This command is generally issued in the HDL simulator. It also may be run from the HDL simulator prompt or from a Tcl script shell (`tclsh`).

### Arguments

`<instance>`

Specifies the instance of an HDL module to load for verification.

`<vsim_args>`

Specifies one or more ModelSim `vsim` command arguments. For details, see the description of `vsim` in the ModelSim documentation.

### Examples

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

```
ModelSim> vsimmatlab work.parse
```

**Introduced in R2008a**

## vsimulink

Load instantiated HDL module for cosimulation with ModelSim and Simulink

### Syntax

```
vsimulink instance> [<vsim_args>]
```

### Description

The `vsimulink` command loads the specified instance of an HDL module for cosimulation and sets up ModelSim so it can establish a communication link with Simulink. ModelSim opens a simulation workspace and displays a series of messages in the command window as it loads the HDL module's packages and architectures.

This command is issued in the HDL simulator. The communication mode is determined by the call to `vsim`, which must be issued before you call `vsimulink`.

### Argument

<instance>

Specifies the instance of an HDL module to load for cosimulation.

<vsim\_args>

Specifies one or more ModelSim `vsim` command arguments. For details, see the description of `vsim` in the ModelSim documentation. Do not issue a command such as `vsim < command.do` with this parameter.

### Examples

The following command loads the HDL module instance `parse` from library `work` for cosimulation and sets up ModelSim so it can establish a communication link with Simulink:

```
ModelSim> vsimulink work.parse
```

**Introduced in R2008a**



# waitForHdlClient

Wait until specified event ID is obtained or time-out occurs

## Syntax

```
waitForHdlClient (TimeOut, EventID)
waitForHdlClient (TimeOut)
waitForHdlClient
output = waitForHdlClient (TimeOut, EventID)
```

## Description

`waitForHdlClient (TimeOut, EventID)` waits for the expected HDL simulator event ID to arrive at the MATLAB server (`hdldaemon`) before processing continues. If the expected event ID arrives before the number of seconds specified by the `TimeOut` parameter, the value returned by the HDL simulator is the HDL simulator process identification (PID). Otherwise, the returned value is `-1`.

`waitForHdlClient (TimeOut)` waits for `EventID = 1` for `TimeOut` seconds.

`waitForHdlClient` waits for `EventID = 1` for 60 seconds.

`output = waitForHdlClient (TimeOut, EventID)` returns the process identification (PID) in `output`. Although you are not required to provide an output variable, MATLAB returns an error if a time-out occurs and the output argument is not specified.

## Input Arguments

### **TimeOut**

Number of seconds to wait for a response from the HDL simulator

### **EventID**

The HDL simulator event ID. *EventID* must be a positive number less than the maximum value of a 32-bit signed integer. The value should match the event ID sent by the `notifyMatlabServer` command in the HDL simulator.

*EventID* can be either a scalar or vector value. If *EventID* is a vector, the function return a value only if all elements of the vector have been collected or if a time-out occurs. The returned output value is the same size as the event ID, and each element of the output variable is the detected PID of the HDL simulator that sends the corresponding event ID element.

## Output Arguments

### **output**

Output variable for holding returned value from call to `waitForHdlClient`. Contains either the HDL simulator process identification (PID) or `-1` if an error occurs.

## Examples

Wait for event ID 2 for 120 seconds.

```
>> hdlpid = waitForHdlClient(120, 2);
```

## See Also

`notifyMatlabServer`

**Introduced in R2012b**