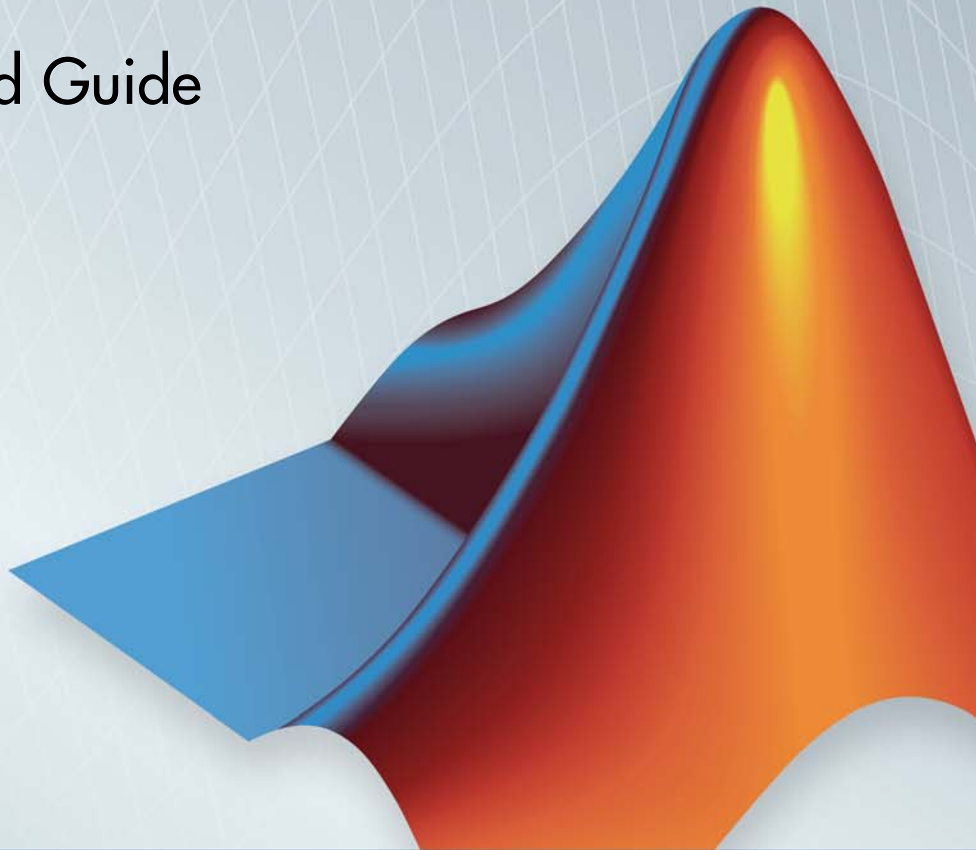
HDL Verifier™

Getting Started Guide

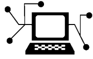
R2013a



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

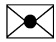


www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)



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

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

HDL Verifier™ Getting Started Guide

© COPYRIGHT 2003–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see 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)

Contents

1 Introduction

Product Description **1-2**
Key Features **1-2**

2 About HDL Verifier

About HDL Cosimulation **2-2**
HDL Cosimulation with MATLAB or Simulink **2-2**
Communications for HDL Cosimulation **2-7**
Hardware Description Language (HDL) Support **2-7**
HDL Cosimulation Workflows **2-7**
Product Features and Platform Support **2-8**

About FPGA Verification **2-9**
FPGA Verification with HDL Verifier and HDL Coder . . . **2-9**
Product Features and Platform Support **2-10**

About TLM Component Generation **2-12**
Generating TLM Components for Virtual Platform Development **2-12**
Typical Users and Applications **2-13**
Product Feature and Platform Support **2-14**

3 Third-Party Product Requirements

Supported EDA Tools **3-2**
Cosimulation Requirements **3-2**

FPGA Verification Requirements 3-3
TLM Generation System Requirements 3-7

4 System Objects

What Is a System Toolbox? **4-2**

What Are System Objects? **4-3**

When to Use System Objects Instead of MATLAB Functions **4-5**
System Objects vs. MATLAB Functions 4-5
Process Audio Data Using Only MATLAB Functions Code 4-5
Process Audio Data Using System Objects 4-6

System Design and Simulation in MATLAB **4-8**

System Objects in MATLAB Code Generation **4-9**
System Objects in Generated Code 4-9
System Objects in codegen 4-16
System Objects in the MATLAB Function Block 4-16
System Objects and MATLAB Compiler Software 4-16

System Objects in Simulink **4-17**

System Object Methods **4-18**
What Are System Object Methods? 4-18
The Step Method 4-18
Common Methods 4-20

System Design Using System Objects **4-22**
Create Components for Your System 4-22
Configure Components for Your System 4-23
Assemble Components to Create Your System 4-26
Run Your System 4-29

Reconfigure Your System During Runtime 4-29

Index

1

Introduction

Product Description

Verify VHDL® and Verilog® using HDL simulators and FPGA-in-the-loop test benches

HDL Verifier™ automates Verilog and VHDL design verification using HDL simulators and FPGA hardware-in-the-loop. It provides interfaces that link MATLAB® and Simulink® with Cadence Incisive®, Mentor Graphics® ModelSim®, and Mentor Graphics Questa® HDL simulators. It also supports FPGA-in-the-loop verification with Xilinx® and Altera® FPGA boards.

HDL Verifier automates verification by using MATLAB or Simulink to stimulate your HDL code and analyze its response. This approach eliminates the need to author standalone Verilog or VHDL test benches.

Key Features

- Cosimulation support for Cadence Incisive and for Mentor Graphics ModelSim and Questa
- FPGA-in-the-loop verification using Xilinx and Altera FPGA boards
- MATLAB functions and Simulink blocks
- Generation of IEEE® 1666 SystemC TLM 2.0 compatible transaction-level models
- Interactive or batch-mode cosimulation and debugging
- Single-machine, multiple-machine, and cross-network cosimulation

2

About HDL Verifier

- "About HDL Cosimulation" on page 2-2
- "About FPGA Verification" on page 2-9
- "About TLM Component Generation" on page 2-12

About HDL Cosimulation

In this section...

"HDL Cosimulation with MATLAB or Simulink" on page 2-2
"Communications for HDL Cosimulation" on page 2-7
"Hardware Description Language (HDL) Support" on page 2-7
"HDL Cosimulation Workflows" on page 2-7
"Product Features and Platform Support" on page 2-8

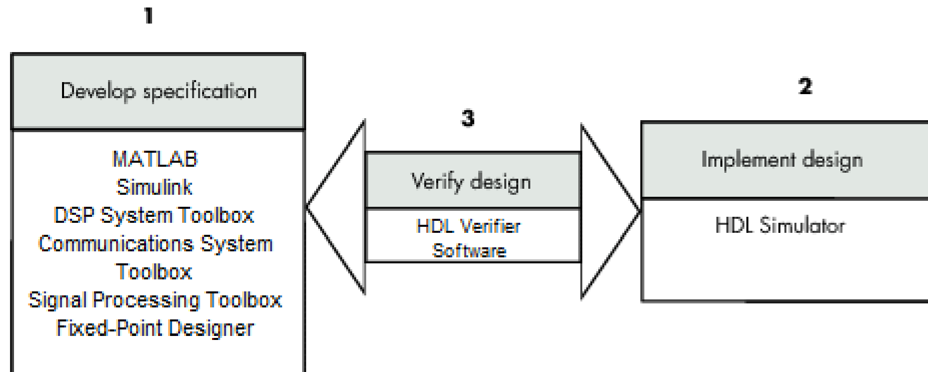
HDL Cosimulation with MATLAB or Simulink

The HDL Verifier software consists of MATLAB functions, a MATLAB System object™, and a library of Simulink blocks, all of which establish communication links between the HDL simulator and MATLAB or Simulink.

HDL Verifier software streamlines FPGA and ASIC development by integrating tools available for these processes:

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

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



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

In addition to the preceding design scenario, HDL Verifier software enables you to work with tools in the following ways:

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

---

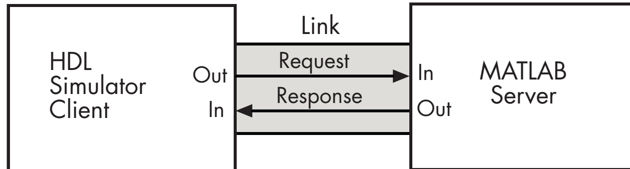
Note You can cosimulate a module using SystemVerilog, SystemC or both with MATLAB or Simulink using the HDL Verifier software. Write simple wrappers around the SystemC and make sure that the SystemVerilog cosimulation connections are to ports or signals of data types supported by the link cosimulation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

More discussion on how cosimulation works can be found in the following sections:

- "Linking with MATLAB and the HDL Simulator" on page 2-4
- "Linking with Simulink and the HDL Simulator" on page 2-5
- "The HDL Cosimulation Wizard" on page 2-7

Linking with MATLAB and the HDL Simulator

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

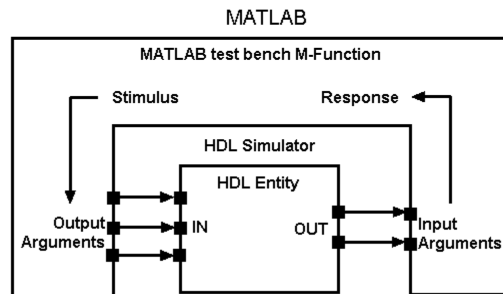


In this scenario, a MATLAB server function waits for service requests that it receives from an HDL simulator session. After receiving a request, the server establishes a communication link and invokes a specified MATLAB function that computes data for, verifies, or visualizes the HDL module (coded in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

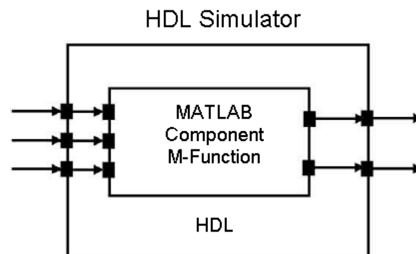
After the server is running, you can start and configure the HDL simulator or use with MATLAB with the supplied HDL Verifier function:

- `nclaunch` (Incisive)
- `vsim` (ModelSim)

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



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

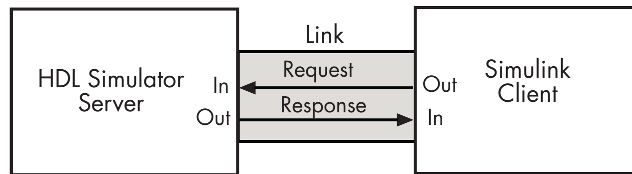


When you begin a specific test bench or component session, you specify parameters that identify the following information:

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

Linking with Simulink and the HDL Simulator

When linked with Simulink, the HDL simulator functions as the server, as shown in the following figure.



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

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

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

HDL Verifier software equips the HDL simulator with a set of customized functions. For ModelSim, when you use the function `vsimulink`, you execute the HDL simulator with an instance of an HDL module for cosimulation with Simulink. After the module is loaded, you can start the cosimulation session from Simulink. Incisive users can perform the same operations with the function `hdlsimulink`.

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

- View Simulink simulation waveforms in your HDL simulation environment

- Compare results of multiple simulation runs, using the same or different simulation environments
- Use as input to post-simulation analysis tools

The HDL Cosimulation Wizard

HDL Verifier contains the Cosimulation Wizard feature, which uses existing HDL code to create a customized MATLAB function (test bench or component), MATLAB System object, or Simulink HDL Cosimulation block. For more information, see "Import HDL Code With the HDL Cosimulation Wizard".

Communications for HDL Cosimulation

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

TCP/IP socket mode is more versatile. You can use it for single-system and network configurations. This option offers the greatest scalability. For more on TCP/IP socket communication, see "Choosing TCP/IP Socket Ports".

Hardware Description Language (HDL) Support

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

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

HDL Cosimulation Workflows

The HDL Verifier User Guide provides instruction for using the verification software with supported HDL simulators for the following workflows:

- Simulating an HDL Component in a MATLAB Test Bench Environment
- Replacing an HDL Component with a MATLAB Component Function
- Simulating an HDL Component in a Simulink Test Bench Environment
- Replacing an HDL Component with a Simulink Algorithm
- Recording Simulink Signal State Transitions for Post-Processing

Product Features and Platform Support

Product Feature	Required Products	Recommended Products	Supported Platforms
MATLAB and HDL simulator cosimulation (function)	MATLAB	Fixed-Point Designer™, Signal Processing Toolbox™	Windows® 32- and 64-bit; Linux® 64-bit
MATLAB and HDL simulator cosimulation (System object)	MATLAB and Fixed-Point Designer	Communications System Toolbox™, DSP System Toolbox™	Windows 32- and 64-bit; Linux 64-bit
Simulink and HDL simulator cosimulation	Simulink, Fixed-Point Designer, and Fixed-Point Designer	Signal Processing Toolbox, DSP System Toolbox	Windows 32- and 64-bit; Linux 64-bit

About FPGA Verification

In this section...
“FPGA Verification with HDL Verifier and HDL Coder” on page 2-9
“Product Features and Platform Support” on page 2-10

FPGA Verification with HDL Verifier and HDL Coder

HDL Verifier works with Simulink or MATLAB andHDL Coder™ and the supported FPGA development environment to prepare your automatically generated HDL Code for implementation in an FPGA. FPGA-in-the-Loop simulation allows you to run a Simulink or MATLAB simulation with an FPGA board strictly synchronized with this software. This process lets you get real world data into your design while accelerating your simulation with the speed of an FPGA.

You can generate a FIL programming file in one of the following ways:

- With the HDL Verifier FIL Wizard.
- With the HDL Coder Workflow Advisor (see HDL Coder documentation for instruction).

The FIL Wizard uses any synthesizable HDL code including code automatically generated from Simulink models by HDL Coder software. When you use FIL in the Workflow Advisor, HDL Coder uses the loaded design to create the HDL code. Either way, this HDL code is then augmented by customized code for FIL communication with your design and assembled into an FPGA project. The applicable downstream tools are used to process that project to create a programming file that is automatically downloaded to the FPGA device on a development board for verification.

HDL Verifier supports the use of a FIL block in a model reference block and a System object in conjunction with a MATLAB program.

See HDL Verifier product page for a list of currently supported devices and boards. You download board definitions files for use with FIL with the Support Package Installer. See “Performing FPGA-in-the-Loop Simulation”.

Product Features and Platform Support

- "Preregistered FPGA Devices for FIL Simulation" on page 2-10
- "Supported FPGA Device Families for Clock Module Generation" on page 2-11

Product Feature	Required Products	Recommended Products	Supported Platforms
FPGA-in-the-Loop	For FIL simulation with MATLAB: MATLAB, Fixed-Point Designer For FIL simulation with Simulink: Simulink, Fixed-Point Designer, Fixed-Point Designer	HDL Coder	Windows 32- and 64-bit; Linux 64-bit

Preregistered FPGA Devices for FIL Simulation

HDL Verifier supports FIL simulation on the devices as described in "Supported FPGA Device Families for Board Customization" on page 3-6. The board definition files for these boards are in the "FIL Support Package Download". You may also add other FPGA boards for use with FIL with FPGA board customization ().

FIL Support Package. The FIL Support Package contains the definition files for all the supported boards. You can download a Xilinx-only package or an Altera-only package, or you may download both packages. You must download one of the packages before you can use FIL, or you can customize your own board definition file using the FPGA Board Manager.

For instructions on how to download the FIL Support Package, see "FIL Support Package Download".

Supported FPGA Device Families for Clock Module Generation

For project generation with Filter Design HDL Coder™, see Xilinx documentation for a full list of supported FPGA families in ISE.

With the current release, clock module generation is supported for the following device families:

- Spartan-3
- Spartan-3A and Spartan-3AN
- Spartan-3A DSP
- Spartan-3E
- Spartan-6
- Virtex-4
- Virtex-5

About TLM Component Generation

In this section...

"Generating TLM Components for Virtual Platform Development" on page 2-12

"Typical Users and Applications" on page 2-13

"Product Feature and Platform Support" on page 2-14

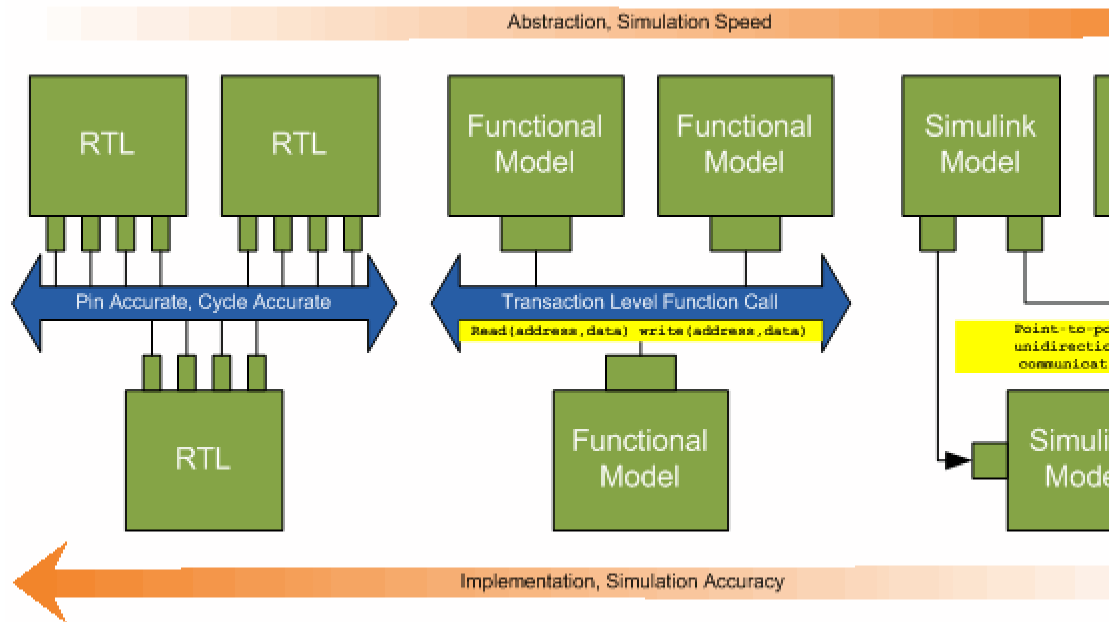
Generating TLM Components for Virtual Platform Development

HDL Verifier lets you create a SystemC Transaction Level Model (TLM) that can be executed in any OSCI-compatible TLM 2.0 environment, including a commercial virtual platform.

When used with virtual platforms, HDL Verifier joins two different modeling environments: Simulink for high-level algorithm development and virtual platforms for system architectural modeling. The Simulink modeling typically dispenses with implementation details of the hardware system such as processor and operating system, system initialization, memory subsystems, device configuration and control, and the particular hardware protocols for transferring data both internally and externally.

The virtual platform is a simulation environment that is concerned about the hardware details: it has components that map to hardware devices such as processors, memories, and peripherals, and a means to model the hardware interconnect between them.

Although many goals could be met with a virtual platform model, the ideal scenario for virtual platforms is to allow for software development—both high level application software and low-level device driver software—by having fairly abstract models for the hardware interconnect that allow the virtual platform to run at near real-time speeds, as demonstrated in the following diagram.



The functional model provides a sort of halfway point between the speed you can achieve with abstraction and the accuracy you get with implementation.

Typical Users and Applications

Using HDL Verifier and Simulink, you can create a TLM-2.0-compliant SystemC Transaction Level Model (TLM) that can be executed in any OSCI-compatible TLM 2.0 environment, including a commercial virtual platform.

Typical users and applications include:

- System-level engineers designing electronic system models that include architectural characteristics
- Software developers who want to incorporate an algorithm into a virtual platform without using an instruction set simulator (ISS).

- Hardware functional verification engineers. In this case, the algorithm represents a piece of hardware going into a chip.

Product Feature and Platform Support

Product Feature	Required Products	Recommended Products	Supported Platforms
TLM Generator	Simulink Coder™ OR Embedded Coder®		Windows 32-bit and 64–bit; Linux 64-bit

Third-Party Product Requirements

Supported EDA Tools

In this section...
"Cosimulation Requirements" on page 3-2
"FPGA Verification Requirements" on page 3-3
"TLM Generation System Requirements" on page 3-7

Cosimulation Requirements

- "Cadence Incisive Requirements" on page 3-2
- "Mentor Graphics Questa and ModelSim Usage Requirements" on page 3-2

Cadence Incisive Requirements

MATLAB and Simulink support Cadence® verification tools using HDL Verifier. Use one of these recommended versions, which have been fully tested against the current release:

- ES 10.2-s040
- IES 9.2-s014
- IUS 8.2-s009
- IUS 11.10-s005
- e12.1-s004

The HDL Verifier shared libraries (`liblfihdls*.so`, `liblfihdlc*.so`) are built using the gcc included in the Cadence Incisive simulator platform distribution. Before you link your own applications into the HDL simulator, first try building against this gcc. See the HDL simulator documentation for more details about how to build and link your own applications.

Mentor Graphics Questa and ModelSim Usage Requirements

MATLAB and Simulink support Mentor Graphics verification tools using HDL Verifier. Use one of the following recommended versions. Each version has been fully tested against the current release:

- ModelSim SE 10.1a, 10.0c, 6.6d, 6.5f
- ModelSim PE 10.1a, 10.0c, 6.6d, 6.5f
- ModelSim DE 10.1a, 10.0c
- Questa 10.0a

TheLinux platform requires that HDL Verifier software run gcc c++ libraries (4.1 or later). You should install a recent version of the gcc c++ library on your computer. To determine which libraries are installed on your computer, type the command:

```
gcc -v
```

FPGA Verification Requirements

- "Xilinx ISE Usage Requirements" on page 3-3
- "Altera Quartus II Usage Requirements" on page 3-4
- "Supported FPGA Devices for FIL Simulation" on page 3-4
- "Supported FPGA Device Families for Board Customization" on page 3-6
- "Supported FPGA Device Families for Clock Module Generation" on page 3-6

Xilinx ISE Usage Requirements

MATLAB and Simulink support Xilinx design tools using HDL Verifier.

- FPGA-in-the-Loop and FPGA Automation are tested with Xilinx ISE 14.2.
- ISE 11.1 or newer is recommended
- Additional requirements for clock module generation using FPGA Automation:
  - 12.1 or later: Windows only
  - 11.4: Windows 32-bit only
- Consult Xilinx user documentation for compatibility of ISE tools with various Linux distributions.

Note Xilinx does not ship the Digilent plugin with ISE 14.2. To get the plugin, see the Digilent plugin and related software download page on the Digilent web site.

Altera Quartus II Usage Requirements

MATLAB and Simulink support Altera design tools using HDL Verifier.

- FPGA-in-the-Loop is tested with Altera Quartus™ II 12.0.

Supported FPGA Devices for FIL Simulation

HDL Verifier supports FIL simulation on the devices shown in the following table. The board definition files for these boards are in the “FIL Support Package Download”. You may also add other FPGA boards for use with FIL with FPGA board customization ().

Device Family	Board	Comments
Xilinx Spartan-6	Spartan-6 SP605 Spartan-6 SP601 XUP Atlys Spartan-6	
Xilinx Virtex-6	Virtex-6 ML605	
Xilinx Virtex-5	Virtex-5 ML505 Virtex-5 ML506 Virtex-5 ML507 Virtex-5 XUPV5–LX110T	
Xilinx Virtex-4	Virtex-4 ML401 Virtex-4 ML402 Virtex-4 ML403	
Altera Arria II	Arria II GX FPGA development kit	

Device Family	Board	Comments
Altera Cyclone IV	Cyclone IV GX FPGA development kit DE2-115 development and education board	The Altera DE2-115 FPGA development board has two Ethernet ports. FPGA-in-the-Loop uses only Ethernet 0 port. Make sure that you connect your host computer with the Ethernet 0 port on the board via an Ethernet cable.
Altera Cyclone III	Cyclone III FPGA development kit Altera Nios II Embedded Evaluation Kit, Cyclone III Edition	

Limitations.

- Ethernet PHY RGMII interface is not supported for Xilinx Spartan6 family when used with FPGA-in-the-Loop.
- For FPGA development boards that have more than one FPGA device, only one such device can be used with FIL.

FIL Support Package. The FIL Support Package contains the definition files for all the supported boards. You can download a Xilinx-only package or an Altera-only package, or you may download both packages. You must download one of the packages before you can use FIL, or you can customize your own board definition file using the FPGA Board Manager.

For instructions on how to download the FIL Support Package, see "FIL Support Package Download".

Supported FPGA Device Families for Board Customization

HDL Verifier supports the following FPGA device families for board customization; that is, when you create your own board definition file. See "FPGA Board Customization".

Device Family	
Xilinx	Virtex4
	Virtex5
	Virtex6
	Spartan6
	Kintex7
Altera	Cyclone III
	Cyclone IV
	Arria II
	Stratix IV
	Stratix V

Supported FPGA Device Families for Clock Module Generation

For project generation with Filter Design HDL Coder, see Xilinx documentation for a full list of supported FPGA families in ISE.

With the current release, clock module generation is supported for the following device families:

- Spartan-3
- Spartan-3A and Spartan-3AN
- Spartan-3A DSP
- Spartan-3E
- Spartan-6
- Virtex-4

- Virtex-5

TLM Generation System Requirements

With the current release, TLMG includes support for:

- Compilers:
  - Visual Studio®: VS2005, VS2008, VS2010
  - gcc 4.4.6
- SystemC:
  - SystemC 2.2.0 with TLM 2.0.1
  - SystemC 2.3.0 (TLM included)

You can download SystemC and TLM libraries at http://accellera.org. Consult the Accellera site for information about how to build these libraries after downloading.

4

System Objects

- "What Is a System Toolbox?" on page 4-2
- "What Are System Objects?" on page 4-3
- "When to Use System Objects Instead of MATLAB Functions" on page 4-5
- "System Design and Simulation in MATLAB" on page 4-8
- "System Objects in MATLAB Code Generation" on page 4-9
- "System Objects in Simulink" on page 4-17
- "System Object Methods" on page 4-18
- "System Design Using System Objects" on page 4-22

What Is a System Toolbox?

System Toolbox products provide algorithms and tools for designing, simulating, and deploying dynamic systems in MATLAB and Simulink. These toolboxes contain MATLAB functions, System objects, and Simulink blocks that deliver the same design and verification capabilities across MATLAB and Simulink, enabling more effective collaboration among system designers. Available System Toolbox products include:

- DSP System Toolbox
- Communications System Toolbox
- Computer Vision System Toolbox
- Phased Array System Toolbox

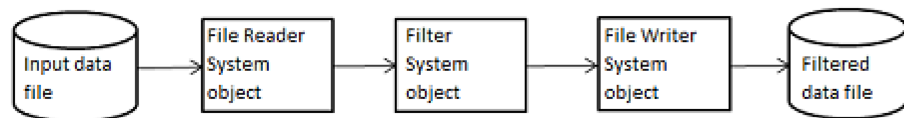
System Toolboxes support floating-point and fixed-point streaming data simulation for both sample- and frame-based data. They provide a programming environment for defining and executing code for various aspects of a system, such as initialization and reset. System Toolboxes also support code generation for a range of system development tasks and workflows, such as:

- Rapid development of reusable IP and test benches
- Sharing of component libraries and systems models across teams
- Large system simulation
- C-code generation for embedded processors
- Finite wordlength effects modeling and optimization
- Ability to prototype and test on real-time hardware

What Are System Objects?

A System object is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer license)
- C code generation (requires a MATLAB Coder or Simulink Coder license)
- HDL code generation (requires an HDL Coder license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

When to Use System Objects Instead of MATLAB Functions

In this section...

"System Objects vs. MATLAB Functions" on page 4-5
"Process Audio Data Using Only MATLAB Functions Code" on page 4-5
"Process Audio Data Using System Objects" on page 4-6

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, your can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows code using only MATLAB functions to read audio data from a file, filter it, and then play the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);
maxSamples = audioInfo.TotalSamples;
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160,.15);
```

Initialize the filter states.

```
z = zeros(1,numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1
    audio = audioread(fname,[nIdx nIdx+frameSize-1]);
    [y,z] = filter(b,1,audio,z);
    sound(y,fs);
    nIdx = nIdx+frameSize;
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows code using System objects from the DSP System Toolbox software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname,'OutputDataType','single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

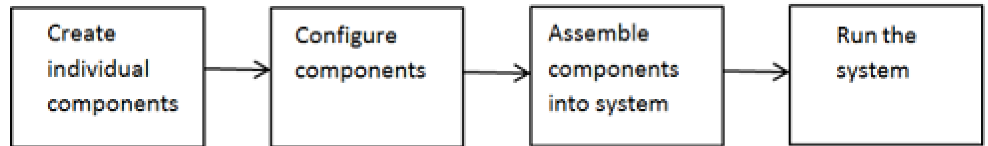
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects as shown in this diagram.



1. Create individual components — Create the System objects to use in your system. See "Create Components for Your System" on page 4-22 for information.

2. Configure components — If necessary, change the objects' property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See "Configure Components for Your System" on page 4-23 for information.

3. Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variable as inputs and outputs to simulate your system. See "Assemble Components to Create Your System" on page 4-26 for information.

4. Run the system — Run your program, which uses the `step` method to run your system's System objects. You can change tunable properties while your system is running. See "Run Your System" on page 4-29 and "Reconfigure Your System During Runtime" on page 4-29 for information.

System Objects in MATLAB Code Generation

In this section...

"System Objects in Generated Code" on page 4-9
"System Objects in codegen" on page 4-16
"System Objects in the MATLAB Function Block" on page 4-16
"System Objects and MATLAB® Compiler™ Software" on page 4-16

System Objects in Generated Code

You can generate C/C++ code from your system that contains System objects by using the MATLAB Coder product. Using this product, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms.

Note Most, but not all, System objects support code generation. Refer to the particular object's reference page for information.

System Objects Code with Persistent Objects for Code Generation

This example shows how to use System objects to make MATLAB code suitable for code generation. The example highlights key factors to consider, such as passing property values and using extrinsic functions. It also shows that by using persistent objects, the object states are maintained between calls.

```
function w = lmssystem(x, d)
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.
    persistent hlms;

    % Initialize persistent System objects only once
    % Do this with 'if isempty(persistent variable).'
```

```
    % This condition will be false after the first time.

    if isempty(hlms)
        % Create LMS adaptive filter used for system
        % identification. Pass property value arguments
        % as constructor arguments. Property values must
        % be constants during compile time.

        hlms = dsp.LMSFilter(11,'StepSize',0.01);
    end

    [~,~,w] = step(hlms,x,d);      % Filter weights
end
```

This example shows how to compile the `lmssystem` function and produce a MEX file with the same name in the current directory.

```
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter

coefs = fir1(10,.25);
hfilt = dsp.FIRFilter('Numerator', coefs);

x = randn(1000,1);                     % Input signal
hSrc = dsp.SignalSource(x,100);        % Use x as input-signal with
                                       % 100 samples per frame

% Generate code for lmssystem
codegen lmssystem -args {ones(100,1),ones(100,1)}

while ~isDone(hSrc)
    in = step(hSrc);
    d  = step(hfilt,in) + 0.01*randn(100,1);  % Desired signal
    w  = lmssystem_mex(in,d);          % Call generated mex file
    stem([coefs.',w]);
end


function ex_system_codegen
% Find corresponding interest points between a pair of images using local
```

```
% neighborhoods.

%#codegen

% Declare System objects as persistent.
persistent cornerDetector colorSpaceConverter

% Initialize persistent System objects only once
% Do this with 'if isempty(persistent variable).'
% This condition will be false after the first time.
if isempty(cornerDetector)

    % Create system objects. Pass property value arguments as constructor
    % arguments. Property values must be constants during compile time.

    cornerDetector = vision.CornerDetector('Method',...
        'Harris corner detection (Harris & Stephens)');

    colorSpaceConverter = vision.ColorSpaceConverter('Conversion',...
            'RGB to intensity');
end

% Declare functions called into MATLAB that do not generate
% code as extrinsic.
coder.extrinsic('imread');

% The output of an extrinsic function is an mxArray - also called a MATLAB
% array. To use mxArrays returned by extrinsic functions, assign the
% mxArray to a variable whose type and size is defined.
imgLeft  = zeros([300 400 3],'uint8');
imgRight = zeros([300 400 3],'uint8');

% Call extrinsic function
imgLeft  = imread('viprectification_deskLeft.png');
imgRight = imread('viprectification_deskRight.png');

% Convert RGB to grayscale
I1 = step(colorSpaceConverter,imgLeft);
I2 = step(colorSpaceConverter,imgRight);
```

```
% Find corners
points1 = step(cornerDetector, I1);
points2 = step(cornerDetector, I2);

% Extract neighborhood features
[features1, valid_points1] = extractFeatures(I1, points1);
[features2, valid_points2] = extractFeatures(I2, points2);

% Match features
index_pairs = matchFeatures(features1, features2);

% Retrieve locations of corresponding points for each image
matched_points1 = valid_points1(index_pairs(:, 1), :);
matched_points2 = valid_points2(index_pairs(:, 2), :);

% Visualize corresponding points
coder.extrinsic('showMatchedFeatures')
figure; showMatchedFeatures(I1, I2, matched_points1, matched_points2);
```

For another detailed code generation example, see "Generate Code for MATLAB Handle Classes and System Objects" in the MATLAB Coder product documentation.

System Objects Code Without Persistent Objects for Code Generation

The following example, using System objects, does not use the persistent keyword because calling a persistent object with different data types causes a data type mismatch error. This example filters the input and then performs a discrete cosine transform on the filtered output.

```
function [out] = FilterAndDCTLib(in)
   hFIR = dsp.FIRFilter('Numerator',fir1(10,0.5));
   DCT = dsp.DCT;

   % Run the objects to get the filtered spectrum
   firOut = hFIR.step(in);
   out = hDCT.step(firOut);
```

```
function [out1, out2] = CompareRealInt(in1)
   % Call the library function, FilterAndDCTLib, which can
   % generate code for multiple calls each with a different data type.

   % Convert input data from double to int16
   in2 = int16(in1);

   % Call the library function for both data types, double and int16
   out1 = FilterAndDCTLib(in1);
   out2 = FilterAndDCTLib(in2);


function RunDCTExample
   % Execute everything needed at the command line to run the example

   warnState = warning('off','SimulinkFixedPoint:util:fxpParameterUnderflow

   % Create vector, length 256, of data containing noise and sinusoids
   dataLength = 256;
   sampleData = rand(dataLength,1) + 3*sin(2*pi*[1:dataLength]*.085)' ...
             + 2*cos(2*pi*[1:dataLength]*.02)';

   % Generate code and run generated file
   codegen CompareRealInt -args {sampleData}
   [out1,out2] = CompareRealInt_mex(sampleData);

   % Compare the the floating point results, in blue
   % with the int16 results, in red
   plot(out1,'b');
   hold on;
   plot(out2,'r');
   hold off

   warning(warnState.state,warnState.identifier);
end
```

Usage Rules and Limitations for System Objects in Generated MATLAB Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Create child System objects only in the parent constructor or in the `setup` method. Calls to the child constructor cannot be in a function called from the parent constructor or the `setup` method.
- Set arguments to System object constructors as compile-time constants.

Inputs and Outputs

- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note Variable-size properties in MATLAB Function block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the MATLAB Function block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a MATLAB Function block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that

automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
  - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
  - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation.
- Default values of tunable properties determine the dimension, data type, and complexity of those properties. Objects cannot be used as default values for properties.

Cell Arrays and Global Variables

- Do not use cell arrays.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

  ```
  f = coder.MEXConfig;
  f.GlobalSyncMethod='NoSync'
  ```

  Then, include `'-config f'` in your `codegen` command.

Methods Supported for Code Generation

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`

- `getNumOutputs`
- `isDone` (for sources only)
- `release`
- `reset`
- `set` (for tunable properties)
- `step`

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” and “MATLAB Classes” for more information.

Note Most, but not all, System objects support code generation. Refer to the particular object’s reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see “What Is a MATLAB Function Block?” in the Simulink documentation.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

System Objects in Simulink

You can also include System object code in Simulink models using the MATLAB Function block. This ability to include MATLAB code in Simulink. However, portions of the system are easier to implement in the MATLAB environment. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Object Methods

In this section...

"What Are System Object Methods?" on page 4-18

"The Step Method" on page 4-18

"Common Methods" on page 4-20

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object.

For more information about the step method and other available methods, see the descriptions in "Common Methods" on page 4-20.

Calculate the Effect of Propagating a Signal in Free Space

This example uses two different `step` methods. The first `step` method is associated with the `phased.LinearFMWaveform` object and the second `step` method is associated with the `phased.Freespace` object.

Construct a linear FM waveform with a pulse duration of 50 microseconds, a sweep bandwidth of 100 kHz, an increasing instantaneous frequency, and a pulse repetition frequency (PRF) of 10 kHz..

```
hFM = phased.LinearFMWaveform('SampleRate',1e6,...
    'PulseWidth',5e-5,'PRF',1e4,...
    'SweepBandwidth',1e5,'SweepDirection','Up',...
    'OutputFormat','Pulses','NumPulses',1);
```

Obtain the waveform using the `step` method. Note that the input to the `step` method is a handle to a `phased.LinearFMWaveform` object.

```
Sig = step(hFM);
```

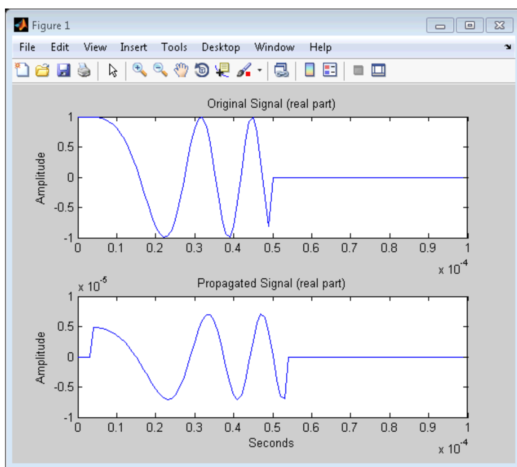
Construct a free space object with a propagation speed equal to the speed of light, an operating frequency of 3 GHz, and a sample rate of 1 MHz. The free space object is constructed to model one way propagation.

```
hFS = phased.FreeSpace(...
    'PropagationSpeed',physconst('LightSpeed'),...
    'OperatingFrequency',3e9,'TwoWayPropagation',false,...
    'SampleRate',1e6);
```

Calculate the effect on the waveform of one-way propagation in free space from coordinates `[0;0;0]` to `[500; 1e3; 20]` and plot the results for comparison.

```
PropSig = step(hFS,Sig,[0; 0; 0],[500; 1e3; 20],...
   [0;0;0],[0;0;0]);
% compare the original signal to the propagated waveform
t = unigrid(0,1/hFS.SampleRate,length(Sig)*1/hFS.SampleRate,'[)');
subplot(211)
plot(t,real(Sig)); title('Original Signal (real part)');
ylabel('Amplitude');
```

```
subplot(212)
plot(t,real(PropSig)); title('Propagated Signal (real part)');
xlabel('Seconds'); ylabel('Amplitude');
```



Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks

Method	Description
	the object. For System objects, use the `release` method instead of a destructor.
`reset`	Resets the internal states of the object to the initial values for that object
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, `getDiscreteState` returns an empty structure.
`clone`	Creates another object of the same type with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked.
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design Using System Objects

In this section...

"Create Components for Your System" on page 4-22
"Configure Components for Your System" on page 4-23
"Assemble Components to Create Your System" on page 4-26
"Run Your System" on page 4-29
"Reconfigure Your System During Runtime" on page 4-29

Create Components for Your System

A System object is a component you can use to create your system in MATLAB. System objects support fixed- or variable-size data. *Variable-size data* is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at initialization time, and therefore, cannot change at run time.

Many System objects are predefined in the software. You can also define your own System objects (see "Define New System Objects""Define New System Objects""Define New System Objects""Define New System Objects").

This example shows the first step in designing a system that processes a long stream of audio data. The data is read from a file, filtered, and then played. The particular predefined components you need are:

- `dsp.AudioFileReader` — Read the file of audio data
- `dsp.FIRFilter` — Filter the audio data
- `dsp.AudioPlayer` — Play the filtered audio data

This example shows the first step in designing a system that finds the edges of objects in a video stream. The particular predefined components you need are:

- `vision.VideoFileReader` — Read the file of video data
- `vision.EdgeDetector` — Detect the edges in the video data
- `vision.AlphaBlender` — Overlay edges onto the original video images

- `vision.VideoPlayer` — Play the video

First, you create the component objects, using default property settings:

```
audioIn = dsp.AudioFileReader;
filtLP = dsp.FIRFilter;
audioOut = dsp.AudioPlayer;
```

First, you create the component objects, using default property settings. You create three `VideoPlayer` objects to play the original video, the edges, and the edges overlayed on the original video.

```
hVideoSrc = vision.VideoFileReader;
hEdge = vision.EdgeDetector;
hAB = vision.AlphaBlender;
hVideoOrig = vision.VideoPlayer;
hVideoEdges = vision.VideoPlayer;
hVideoOverlay = vision.VideoPlayer
```

Next, you configure each System object for your system. See "Configure Components for Your System" on page 4-23.

Note Alternately, if desired, you can "Create and Configure Components at the Same Time" on page 4-25.

Configure Components for Your System

When to Configure Components

If you did not set an object's properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See "Reconfigure Your System During Runtime" on page 4-29 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object's handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure the components for your system by setting the component objects' properties. Use this procedure if you have created your components as described in "Create Components for Your System" on page 4-22.

Note If you have not yet created your components, use the procedure in "Create and Configure Components at the Same Time" on page 4-25.

For the file reader object, specify the file to read and set the output data type.

```
audioIn.Filename = 'speech_dft_8kHz.wav';
audioIn.OutputDataType = 'single';
```

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

```
filtLP.Numerator = fir1(160,.15);
```

For the audio player object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut.SampleRate = audioIn.SampleRate;
```

For the video file reader object, specify the file to read and set the image color space.

```
hVideoSrc.Filename = 'vipmen.avi';
hVideoSrc.ImageColorSpace = 'Intensity';
```

For the edge detector object, specify the edge detection method, the threshold and threshold source, and whether to use edge thinning.

```
hEdge.Method = 'Prewitt';
```

```
hEdge.ThresholdSource = 'Property';
hEdge.Threshold = 15/256;
hEdge.EdgeThinning = true;
```

For the alpha blender object, specify the type of operation to use.

```
hAB.Operation = 'Highlight selected pixels';
```

For the video player objects, specify the names, the window size, and the window position.

```
WindowSize = [190 150];

hVideoOrig.Name = 'Original';
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges.Name = 'Edges';
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay.Name = 'Overlay';
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
    'OutputDataType','single')
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

For the video file reader object, specify the file to read and set the image color space.

```
hVideoSrc = vision.VideoFileReader('vipmen.avi', ...
     'ImageColorSpace', 'Intensity');
```

For the edge detector object, specify the edge detection method, the threshold and threshold source, and whether to use edge thinning.

```
hEdge = vision.EdgeDetector('Method','Prewitt',...
     'ThresholdSource','Property', ...
     'Threshold',15/256,'EdgeThinning',true);
```

For the alpha blender object, specify the type of operation to use.

```
hAB = vision.AlphaBlender('Operation', 'Highlight selected pixels');
```

For the video player objects, specify the names, the window size, and the window position.

```
WindowSize = [190 150];
hVideoOrig = vision.VideoPlayer('Name', 'Original');
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges = vision.VideoPlayer('Name', 'Edges');
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay = vision.VideoPlayer('Name', 'Overlay');
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];
```

After you create the components, you can assemble them in your system. See "Assemble Components to Create Your System" on page 4-26.

Assemble Components to Create Your System

- "Connect Inputs and Outputs" on page 4-27

- "Code for the Whole System" on page 4-27
- "Code for the Whole System" on page 4-28

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object's `step` method as the input to another object's `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see "System Object Methods" on page 4-18.

Code for the Whole System

This example shows the full code for reading, filtering, and playing a file of audio data.

You can type this code on the command line or put it into a program file.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
   'OutputDataType','single');
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);

while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
```

```
end
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system. See "Run Your System" on page 4-29.

Code for the Whole System

This example shows the full code for edge detection.

You can type this code on the command line or put it into a program file.

```
hVideoSrc = vision.VideoFileReader('vipmen.avi', ...
     'ImageColorSpace','Intensity');
hEdge = vision.EdgeDetector('Method','Prewitt', ...
      'ThresholdSource','Property', ...
      'Threshold',35/256,'EdgeThinning',true);
hAB = vision.AlphaBlender('Operation','Highlight selected pixels');

WindowSize = [190 150];
hVideoOrig = vision.VideoPlayer('Name','Original');
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges = vision.VideoPlayer('Name','Edges');
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay = vision.VideoPlayer('Name','Overlay');
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];

while ~isDone(hVideoSrc)
    frame     = step(hVideoSrc);          % Read input video
    edges     = step(hEdge, frame);       % Edge detection
    composite = step(hAB, frame, edges)   % AlphaBlender

    step(hVideoOrig,frame);              % Display original
    step(hVideoEdges,edges);             % Display edges
    step(hVideoOverlay,composite);       % Display edges overlayed
end
release(hVideoSrc);
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system. See "Run Your System" on page 4-29.

Run Your System

- "How to Run Your System" on page 4-29
- "What You Cannot Change While Your System Is Running" on page 4-29

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see "Reconfigure Your System During Runtime" on page 4-29.

Reconfigure Your System During Runtime

- "When Can You Change Component Properties?" on page 4-30

- "Change a Tunable Property in Your System" on page 4-30
- "Change Input Complexity or Dimensions" on page 4-31

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object's reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change a Tunable Property in Your System

You can change the filter type to a high-pass filter as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as at the next iteration of the while loop).

```
reset(audioIn);               % Reset audio file
filtLP.Numerator = fir1(160,0.15,'high');
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

You can change the threshold value as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as, at the next iteration of the while loop).

```
while ~isDone(hVideoSrc)
    frame     = step(hVideoSrc);          % Read input video
    edges     = step(hEdge, frame);       % Edge detection
    composite = step(hAB, frame, edges)   % AlphaBlender
```

```
    hEdge.Threshold = hEdge.Threshold-0.0005; % Tune threshold

    step(hVideoOrig,frame);               % Display original
    step(hVideoEdges,edges);              % Display edges
    step(hVideoOverlay,composite);        % Display edges overlayed
end
```

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

Index

A

application specific integrated circuits
  (ASICs) 1-1
ASICs (application specific integrated
  circuits) 1-1

C

clone method 4-21
communication
  modes of 2-7

E

EDA (Electronic Design Automation) 1-1
Electronic Design Automation (EDA) 1-1

F

field programmable gate arrays (FPGAs) 1-1
FPGAs (field programmable gate arrays) 1-1

G

getDiscreteState method 4-21
getNumInputs method 4-21
getNumOutputs method 4-21

H

hardware description language (HDL). *See* HDL
HDL (hardware description language) 1-1
HDL Verifier™ software
  definition of 1-1

I

info method 4-21
isDone method 4-21
isLocked method 4-21

R

release method 4-21
reset method 4-21

S

shared memory communication 2-7
sockets 2-7
  See also TCP/IP socket communication
step method 4-20
streaming data
  using System objects 4-18
System object
  clone method 4-21
  getDiscreteState method 4-21
  getNumInputs method 4-21
  getNumOutputs method 4-21
  info method 4-21
  isDone method 4-21
  isLocked 4-21
  methods 4-18
  release method 4-21
  reset method 4-21
  step method 4-20
  using with MATLAB code generation 4-9

T

TCP/IP networking protocol 2-7
  See also TCP/IP socket communication
TCP/IP socket communication
  mode 2-7