Simulink® HDL Coder

For Use with MATLAB® and Simulink®

- Computation
- Visualization
- Programming
- Simulation

User's Guide

Version 1

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Simulink HDL Coder User's Guide

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

ModelSim is a registered trademark of Mentor Graphics Corporation.

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

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

September 2006	Online only	New for Version 1.0 (Release 2006b)

Contents

1 Getting Started

2 Introduction to HDL Code Generation

3 Code Generation Options in the Simulink HDL Coder GUI

4 Code Generation Control Files

5 Generating Bit-True Cycle-Accurate Models

6 HDL Compatibility, Code Tracing, and Block Support Reports

7 Interfacing Subsystems and Models to HDL Code

8 Stateflow HDL Code Generation Support

9 Generating Scripts for HDL Simulators and Synthesis Tools

Properties — By Category

10

1

Getting Started

What Is Simulink HDL Coder?

Simulink® HDL Coder lets you generate hardware description language (HDL) code based on models developed in Simulink and finite-state machines developed in Stateflow®. Simulink HDL Coder brings the Simulink Model-Based Design approach into the domain of application-specific integrated circuit (ASIC) and field programmable gate array (FPGA) development. Using Simulink HDL Coder, system architects and designers can spend more time on fine-tuning algorithms and models through rapid prototyping and experimentation and less time on HDL coding.

Simulink HDL Coder in the Hardware Development Process

Typically, you use Simulink to model a design intended for realization as an ASIC or FPGA. Once satisfied that the model meets design requirements, you run the Simulink HDL Coder compatibility checker utility to examine model semantics and blocks for HDL code generation compatibility. You then invoke the Simulink HDL Coder code generator, using either the MATLAB® command line or the Simulink graphical user interface. Simulink HDL Coder generates VHDL or Verilog code that implements the design embodied in the model.

Usually, you also generate a corresponding test bench. You can use the test bench with HDL simulation tools such as ModelSim® to drive the generated HDL code and evaluate its behavior. Simulink HDL Coder generates scripts that automate the process of compiling and simulating your code in these tools. You can also use the MathWorks Link for ModelSim to cosimulate generated HDL entities within a Simulink model.

The test bench feature increases confidence in the correctness of the generated code and saves time spent on test bench implementation. The design and test process is fully iterative. At any point, you can return to the original Simulink model, make modifications, and regenerate code.

When the design and test phase of the project has been completed, you can easily export the generated HDL code to synthesis and layout tools for hardware realization. Simulink HDL Coder generates synthesis scripts for the Synplify family of synthesis tools.

Extending the Code Generation Process

Simulink HDL Coder provides a number of ways to extend the code generation process.

By attaching a *code generation control file* to your model, you can direct many details of the code generation process. At the simplest level, you can use a control file to set code generation options; such a control file could be used as a template for code generation in your organization.

Control files also let you specify how code is generated for selected sets of blocks within the model. Simulink HDL Coder provides alternate HDL *block implementations* for a variety of blocks. You can use statements in a control file to select from among implementations optimized for characteristics such as speed, chip area, or low latency.

In some cases, block-specific optimizations may introduce latencies (delays) or numeric computations (for example, saturation or rounding operations) in the generated code that are not in the original Simulink model. To help you evaluate such cases, Simulink HDL Coder creates a *generated model* — a Simulink model that corresponds exactly to the generated HDL code. This generated model lets you run simulations that produce results that are bit-true to the HDL code, and whose timing is cycle-accurate with respect to the HDL code.

You can interface Simulink HDL Coder generated HDL to existing or legacy HDL code. One way to do this is to use a subsystem in your Simulink model as a placeholder for an HDL entity, and generate a *black box* interface (comprising I/O port definitions only) to that entity. Another way is to generate a cosimulation interface by placing a Link for ModelSim HDL Cosimulation block in your model.

Summary of Key Features

Key features and components of Simulink HDL Coder include

- Generation of synthesizble VHDL or Verilog code from Simulink models and Stateflow charts
- Code generation configured and initiated via graphical user interface, MATLAB command line interface, or M-file programs

- Test bench generation (VHDL or Verilog) for validating generated code
- Generation of models that are bit-true and cycle-accurate with respect to generated HDL code
- Numerous options for controlling the contents and style of the generated HDL code and test bench
- Block support:
  - Simulink built-in
  - Signal Processing Blockset
  - Link for ModelSim HDL Cosimulation block
  - Stateflow chart
  - User-selectable optimized block implementations provided for commonly used blocks
- Code generation control files support:
  - Selection of alternate block implementations for specific blocks or sets of blocks in the model
  - Setting of code generation options
  - Selection of the model or subsystem from which code is to be generated.
  - Definition of default or template HDL code generation settings for your organization
- Generation of subsystem-based identification comments and mapping files for easy tracing of HDL entities back to corresponding elements of the original model
- Generation of interfaces to existing HDL code via:
  - Black box subsystem implementation
  - Cosimulation with ModelSim HDL simulator (requires link for ModelSim)
- Compatibility checker utility that examines your model for HDL code generation compatibility, and generates HTML report with hyperlinks to problematic blocks
- Generation of scripts for EDA tools:

  - ModelSim
  - Synplify
- Model features supported for code generation in Version 1.0:
  - Real data types only (fixed-point and double)
    (Complex data types are not supported.)
  - Fixed-step, discrete, single-rate models
  - Scalar and vector ports (row or column vectors only)

Expected Users and Prerequisites

Simulink HDL Coder users are system and hardware architects and designers who develop, optimize, and verify ASICs or FPGAs. These designers are experienced with VHDL or Verilog but can benefit from automated HDL code generation.

Users are expected to have prerequisite knowledge in the following areas:

- Hardware design and system integration
- VHDL or Verilog
- MATLAB
- Simulink
- Simulink Fixed Point
- Signal Processing Blockset
- HDL simulators, such as ModelSim
- Synthesis tools, such as Synplify

Software Requirements and Installation

Before installing Simulink HDL Coder, make sure that you have the required MathWorks software listed in “Software Requirements” on page 1-7. See also “VHDL and Verilog Language Support” on page 1-8 to check compatibility with HDL compilers and other tools.

Software Requirements

Simulink HDL Coder requires the products listed below. (Version numbers correspond to MATLAB Release 2006b.)

- MATLAB 7.3
- Simulink 6.5
- Simulink Fixed Point 5.3
- Fixed-Point Toolbox 1.5

The following related products are recommended for use with Simulink HDL Coder:

- Stateflow 6.5
- Filter Design HDL Coder 1.5
- Link for ModelSim 2.1
- Signal Processing Toolbox 6.6
- Signal Processing Blockset 6.4

Software Requirements for Simulink HDL Coder Demos

To operate some demos shipped with this release, the following related products are required:

- Filter Design Toolbox 4.0
- Filter Design HDL Coder 1.5
- Link for ModelSim 2.1
- Communications Toolbox 3.4 (required to use Viterbi Decoder demo)

- Communications Blockset 3.4 (required to use Viterbi Decoder demo)

VHDL and Verilog Language Support

Simulink HDL Coder is compatible with HDL compilers, simulators and other tools that support

- VHDL versions 93 and 02
- Verilog-2001 (IEEE 1364-2001) or later

Installing the Software

For information on installing the required software listed above, and optional software, see the MATLAB installation documentation for your platform.

After completing your installation, work through the examples in Chapter 2, “Introduction to HDL Code Generation”, to acquaint yourself with the operation of the product.

Available Help and Demos

Online Help

The following online help is available:

- Online help is available in the MATLAB Help browser. Click the Simulink HDL Coder product link in the browser's Contents pane.
- Documentation in PDF format is available through the Simulink HDL Coder roadmap page in the MATLAB Help browser. Click the **Simulink HDL Coder > Printable Documentation (PDF)** link in the browser's Contents pane.
- M-help for the command line interface functions `makehdl`, `makehdltb`, `checkhdl`, `hdllib`, and `hdlsetup` is available through the MATLAB `doc` and `help` commands. For example:

```
help makehdl
```

Demos

Simulink HDL Coder provides a number of models demonstrating aspects of HDL code generation. To access the demo models:

1. Type the following command at the MATLAB prompt:

   ```
   demos
   ```

2. The **Help** window opens.
3. In the **Demos** pane on the left, select **Simulink > Simulink HDL Coder**.
4. The right pane displays hyperlinks to the available demos. Click the link to the desired demo and follow the demo instructions.

2

Introduction to HDL Code Generation

Overview of Exercises

Simulink HDL Coder supports HDL code generation in your choice of environments:

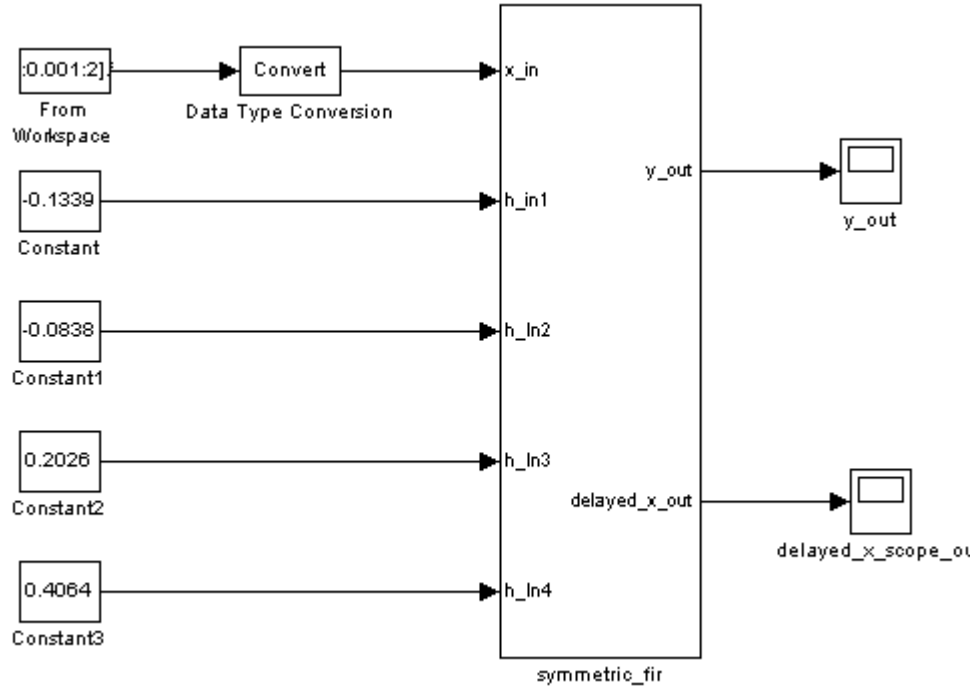
- The MATLAB Command Window supports code generation using the `makehdl`, `makehdltb`, and other functions.
- The Simulink graphical environment (the Simulink Configuration Parameters dialog and/or Model Explorer) provides an integrated view of the model simulation parameters and HDL code generation parameters and functions.

The hands-on exercises in this chapter introduce you to the mechanics of generating and simulating HDL code with Simulink HDL Coder, using the same model to generate code in both environments. In a series of steps, you will

- Configure a simple Simulink model for code generation.
- Generate VHDL code from a subsystem of the model.
- Generate a VHDL test bench and scripts for the ModelSim HDL simulator to drive a simulation of the model.
- Compile and execute the model and test bench code in ModelSim.
- Generate and simulate Verilog code from the same model.
- Check a model for compatibility with Simulink HDL Coder.

The sfir_fixed Demo Model

Simulink HDL Coder provides the `sfir_fixed` demo model as a source model for HDL code generation. The model simulates a symmetric finite impulse response (FIR) filter algorithm, implemented with fixed-point arithmetic. The figure below shows the top level of the model.



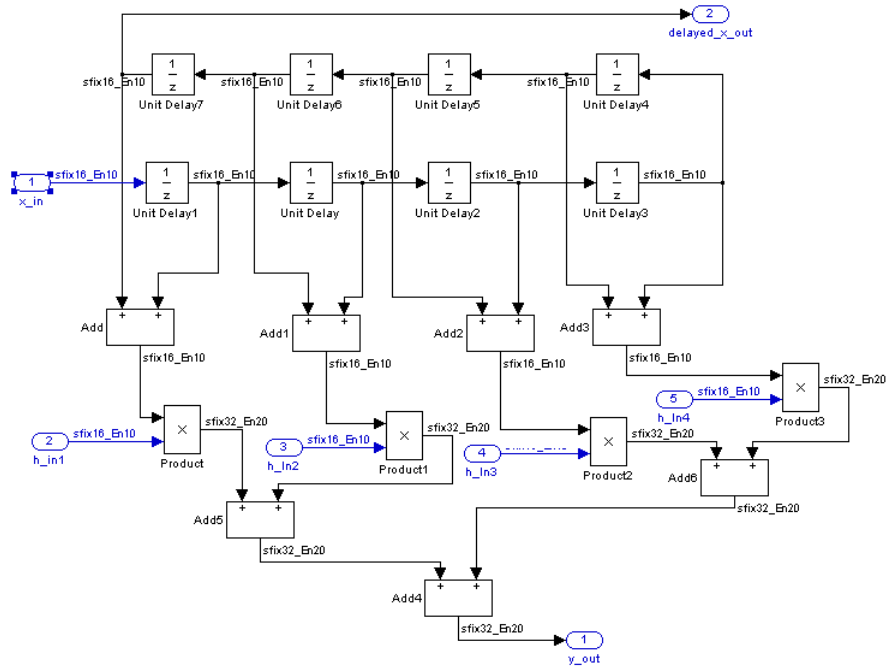
This model employs a division of labor that is useful in HDL design:

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity will be generated, tested, and eventually synthesized from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients.

Note that the Scope blocks are used in Simulink simulation only. They are virtual blocks, and do not generate any HDL code.

The figure below shows the `symmetric_fir` subsystem.



Simulink propagates appropriate fixed-point data types throughout the subsystem. Inputs inherit the data types of the signals presented to them. Where required, internal rules of the blocks determine the correct output data type, given the input data types and the operation performed (for example, the Product blocks output 32-bit signals).

The filter outputs a 32-bit fixed-point result at the `y_out` port, and also replicates its input (after passing it through several delay stages) at the `delayed_x_out` port.

In the exercises that follow, you generate VHDL code that implements the `symmetric_fir` subsystem as an entity. You then generate a test bench from the top-level model. The test bench drives the generated entity, for the required number of clock steps, with stimulus data generated from the Signal From Workspace block.

Generating HDL Code Using MATLAB Commands

This exercise provides a step-by-step introduction to the Simulink HDL Coder code and test bench generation commands, their arguments, and the files created by the code generator. The exercise assumes that you have familiarized yourself with the demo model (see "The sfir_fixed Demo Model" on page 2-3).

The exercise walks you through command line based code generation in the following sections:

Creating Directories and Local Model File

Make a local copy of the demo model and store it in a working directory, as follows.

1. Start MATLAB.
2. Create a directory named `sl_hdlcoder_work`, for example:

   ```
   mkdir D:\work\sl_hdlcoder_work
   ```

   The `sl_hdlcoder_work` will store a local copy of the demo model and to store directories and code generated by Simulink HDL Coder. The location of the directory does not matter, except that it should not be within the MATLAB directory tree.
3. Make the `sl_hdlcoder_work` directory your working directory, for example:

   ```
   cd D:\work\sl_hdlcoder_work
   ```

4 To open the demo model, type the following command at the MATLAB prompt:

```
demos
```

5 The **Help** window opens. In the **Demos** pane on the left, click the + for **Simulink**. Then click the + for **Simulink HDL Coder**. Then double-click the list entry for the Symmetric FIR Filter Demo.

The `sfir_fixed` model opens.

6 Select **Save As** from the Simulink **File** menu and save a local copy of `sfir_fixed.mdl`. to your working directory.

7 Leave the `sfir_fixed` model open and proceed to the next section.

Initializing Model Parameters with hdlsetup

Before generating code, you must set some parameters of the model. Rather than doing this manually, use the Simulink HDL Coder M-file utility, `hdlsetup.m`. The `hdlsetup` command uses the Simulink `set_param` function to set up models for HDL code generation quickly and consistently.

To set the model parameters:

1 At the MATLAB command prompt, type

```
hdlsetup('sfir_fixed')
```

2 Select **Save** from the Simulink **File** menu, to save the model with its new settings.

Before continuing with code generation, consider the settings that `hdlsetup` applies to the model.

`hdlsetup` configures Simulink **Solver** options that are recommended or required by Simulink HDL Coder. These are

- **Type**: `Fixed-step`. Simulink HDL Coder does not currently support variable-step solvers.

- **Solver**: `discrete (no continuous states)`. Other fixed-step solvers could be selected, but this option is usually the correct one for simulating discrete systems.
- **Tasking mode**: `SingleTasking`. Simulink HDL Coder does not currently support models that execute in multitasking mode.

`hdlsetup` also configures the model start and stop times and fixed-step size as follows:

- **Start Time**: `0.0` s
- **Stop Time**: `10` s
- **Fixed step size (fundamental periodic sample time)**: `auto`

Setting **Fixed step size** to `auto` causes Simulink to choose the step size, based on the sample times specified in the model. In the demo model, only the Signal From Workspace block specifies an explicit sample time (1 s); all other blocks inherit this sample time.

The model start and stop times determine the total simulation time. This in turn determines the size of data arrays that are generated to provide stimulus and output data for generated test benches. For the demo model, computation of 10 seconds of test data does not take a significant amount of time. Computation of sample values for more complex models can be time consuming. In such cases, you may want to decrease the total simulation time.

The remaining parameters set by `hdlsetup` affect Simulink error severity levels, data logging, and model display options. If you want to view the complete set of model parameters affected by `hdlsetup`, open `hdlsetup.m` in the MATLAB editor.

The model parameter settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications. For example, `hdlsetup` sets a default **Simulation stop time** of 10 s. A total simulation time of 1000 s would be more realistic for a test of the `sfir_fixed` demo model. If you would like to change the simulation time, enter the desired value into the **Simulation stop time** field of the Simulink window.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

Generating a VHDL Entity from a Subsystem

In this section, you will use the `makehdl` function to generate code for a VHDL entity from the `symmetric_fir` subsystem of the demo model. `makehdl` also generates script files for third-party HDL simulation and synthesis tools.

`makehdl` lets you specify numerous properties that control various features of the generated code. In this example, you will use defaults for all `makehdl` properties.

Before generating code, make sure that you have completed the steps described in “Creating Directories and Local Model File” on page 2-6 and “Initializing Model Parameters with hdlsetup” on page 2-7.

To generate code:

1. Select **Current Directory** from the **Desktop** menu in the MATLAB window. This displays the MATLAB Current Directory browser, which lets you easily access your working directory and the files that will be generated within it.

2. At the MATLAB prompt, type the command

   ```
   makehdl('sfir_fixed/symmetric_fir')
   ```

   This command directs Simulink HDL Coder to generate code from the `symmetric_fir` subsystem within the `sfir_fixed` model, using default values for all properties.

3. Simulink HDL Coder generates code and displays progress messages. The process should complete successfully with the message

   ```
   ### HDL Code Generation Complete.
   ```

   Observe that the names of generated files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB editor.

Note that `makehdl` compiles the model before generating code. Depending on model display options (for example, sample time colors, port data types, etc.), the appearance of the model may change after code generation.

4 By default, `makehdl` generates VHDL code. Code files and scripts are written to a *target directory*. The default target directory is a subdirectory of your working directory, named `hdlsrc`.

A folder icon for the `hdlsrc` directory is now visible in the Current Directory browser. To view generated code and script files, double-click the `hdlsrc` folder icon.

5 The files that `makehdl` has generated in the `hdlsrc` directory are

- `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir` filter.
- `symmetric_fir_compile.do`: ModelSim compilation script (`vcom` command) to compile the generated VHDL code.
- `symmetric_fir_synplify.tcl`: Synplify synthesis script
- `symmetric_fir_map.txt`: Mapping file. This report file maps generated entities (or modules) to the Simulink subsystems that generated them (see "Code Tracing Using the Mapping File" on page 6-5).

6 To view the generated VHDL code in the MATLAB editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser.

At this point it is suggested that you study the `ENTITY` and `ARCHITECTURE` definitions while referring to "HDL Code Generation Defaults" on page 12-13 in the `makehdl` reference documentation. The reference documentation describes the default naming conventions and correspondences between the elements of a Simulink model (subsystems, ports, signals, etc.) and elements of generated HDL code.

7 Before proceeding to the next section, close any files you have opened in the MATLAB editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `sl_hdlcoder_work` directory.

8 Leave the `sfir_fixed` model open and proceed to the next section.

Generating VHDL Test Bench Code

In this section, you use the Simulink HDL Coder test bench generation function, `makehdltb`, to generate a VHDL test bench. The test bench is designed to drive and verify the operation of the `symmetric_fir` entity that was generated in the previous section. A generated test bench includes

- Stimulus data generated by signal sources connected to the entity under test.
- Output data generated by the entity under test. During a test bench run, this data is compared to the outputs of the VHDL model, for verification purposes.
- Clock, reset, and clock enable inputs to drive the entity under test.
- A component instantiation of the entity under test.
- Code to drive the entity under test and compare its outputs to the expected data.

In addition, `makehdltb` generates ModelSim scripts to compile and execute the test bench.

This exercise assumes that your working directory is the same as that used in the previous section. This directory now contains an `hdlsrc` folder containing the previously generated code.

To generate a test bench:

1. At the MATLAB prompt, type the command

   ```
   makehdltb('sfir_fixed/symmetric_fir')
   ```

   This command generates a test bench that is designed to interface to and validate code generated from `symmetric_fir` (or from a subsystem with a functionally identical interface). By default, VHDL test bench code, as well as scripts, are generated in the `hdlsrc` target directory.

2. Simulink HDL Coder generates code and displays progress messages. The process should complete successfully with the message

   ```
   ### HDL TestBench Generation Complete.
   ```

3 To view generated test bench and script files, double-click the `hdlsrc` folder icon in the Current Directory browser. Alternatively, you can click the hyperlinked names of generated files in the code test bench generation progress messages.

   The files generated by `makehdltb` are

   - `symmetric_fir_tb.vhd`: VHDL test bench code and generated test and output data.
   - `symmetric_fir_tb_compile.do`: ModelSim compilation script (`vcom` commands). This script compiles and loads both the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
   - `symmetric_fir_tb_sim.do`: ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

4 If you want to view the generated test bench code in the MATLAB editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser. You may want to study the code while referring to the `makehdltb` reference documentation, which describes the default actions of the test bench generator.

5 Before proceeding to the next section, close any files you have opened in the MATLAB editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `sl_hdlcoder_work` directory.

Verifying Generated Code

You can now take the previously generated code and test bench to an HDL simulator for simulated execution and verification of results. See “Simulating and Verifying Generated HDL Code” on page 2-29 for an example of how to use generated test bench and script files with the Mentor Graphics HDL simulator, ModelSim SE/PE.

Generating a Verilog Module and Test Bench

The procedures for generating Verilog code differ only slightly from those for generating VHDL code. This section provides an overview of the command syntax and the generated files.

Generating a Verilog Module

By default, `makehdl` generates VHDL code. To override the default and generate Verilog code, you must pass in a property/value pair to `makehdl`, setting the `TargetLanguage` property to `'verilog'`, as in this example.

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage','verilog')
```

The above command generates Verilog source code, as well as ModelSim and Synplify scripts, in the default target directory, `hdlsrc`.

The files generated by this example command are

- `symmetric_fir.v`: Verilog code. This file contains a Verilog module implementing the `symmetric_fir` subsystem.
- `symmetric_fir_compile.do`: ModelSim compilation script (`vlog` command) to compile the generated Verilog code.
- `symmetric_fir_synplify.tcl`: Synplify synthesis script.
- `symmetric_fir_map.txt.`: Mapping file. This report file maps generated entities (or modules) to the Simulink subsystems that generated them (see "Code Tracing Using the Mapping File" on page 6-5).

Generating and Executing a Verilog Test Bench

The `makehdltb` syntax for overriding the target language is exactly the same as that for `makehdl`. The following example generates Verilog test bench code to drive the Verilog module, `symmetric_fir`, in the default target directory.

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','verilog')
```

The files generated by this example command are

- `symmetric_fir_tb.v`: Verilog test bench code and generated test and output data.
- `symmetric_fir_tb_compile.do`: ModelSim compilation script (`vlog` commands). This script compiles and loads both the entity to be tested (`symmetric_fir.v`) and the test bench code (`symmetric_fir_tb.v`).

- `symmetric_fir_tb_sim.do`: ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

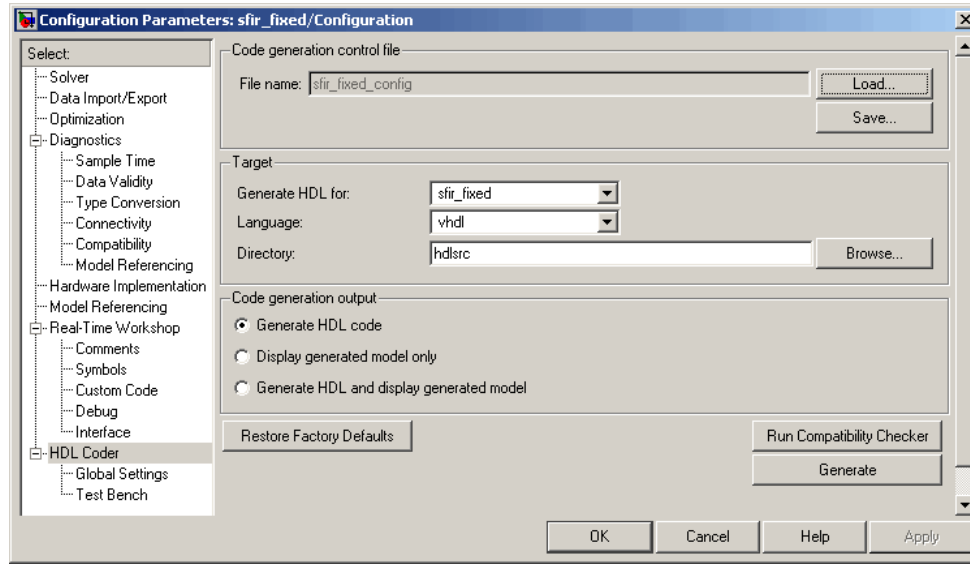
The following listing shows the commands and responses from a ModelSim test bench session using the generated scripts.

```
ModelSim>vlib work
ModelSim> do symmetric_fir_tb_compile.do
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
# -- Compiling module symmetric_fir
#
# Top level modules:
#  symmetric_fir
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
# -- Compiling module symmetric_fir_tb
#
# Top level modules:
#  symmetric_fir_tb
ModelSim>do symmetric_fir_tb_sim.do
# vsim work.symmetric_fir_tb
# Loading work.symmetric_fir_tb
# Loading work.symmetric_fir
# **** Test Complete. ****
# Break at d:/work/sl_hdlcoder_work/vlog_code/symmetric_fir_tb.v line 142
# Simulation Breakpoint: Break at d:/work/sl_hdlcoder_work/vlog_code/symmetric_fir_tb.v line 142
# MACRO ./symmetric_fir_tb_sim.do PAUSED at line 14
```

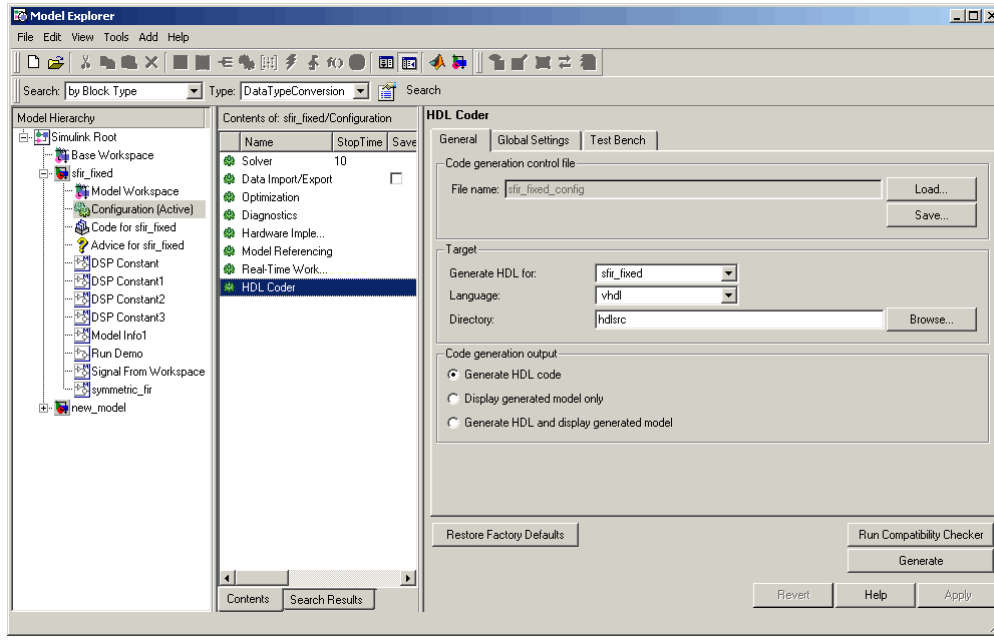
Generating HDL Code in the Simulink GUI

Simulink provides visual access to options and parameters that affect HDL code generation, within the framework of a Simulink configuration set. You can view and edit these options in the Simulink Configuration Parameters dialog, or in the Simulink Model Explorer.

The figure below shows the top-level **HDL Coder** options pane as displayed in the Configuration Parameters dialog.



The figure below shows the top-level **HDL Coder** options pane as displayed in the Model Explorer dialog.



If you are not familiar with Simulink configuration sets and how to view and edit them in the Configuration Parameters dialog, see the following sections of the Simulink documentation:

- “Configuration Sets”
- “Configuration Parameters Dialog Box”

If you are not familiar with the Model Explorer, see “Exploring, Searching, and Browsing Models” in the Simulink documentation.

In the hands-on code generation exercises that follow, you will use the Configuration Parameters dialog view of Simulink HDL Coder options and controls. The exercises use the `sfir_fixed` demo model (see “The sfir_fixed Demo Model” on page 2-3) in basic code generation steps, including

- “Creating Directories and Local Model File” on page 2-17
- “Initializing Model Parameters With hdlsetup” on page 2-18

Creating Directories and Local Model File

Start by setting up a working directory and making a copy of the `sfir_fixed` demo model:

1. Start MATLAB.

2. Create a directory named `sl_hdlcoder_work`, for example:

   ```
   mkdir D:\work\sl_hdlcoder_work
   ```

   You will use `sl_hdlcoder_work` to store a local copy of the demo model and to store directories and code generated by Simulink HDL Coder. The location of the directory does not matter, except that it should not be within the MATLAB directory tree.

3. Make the `sl_hdlcoder_work` directory your working directory, for example:

   ```
   cd D:\work\sl_hdlcoder_work
   ```

4. To open the demo model, type the following command at the MATLAB prompt:

   ```
   demos
   ```

   The **Help** window opens.

5. In the **Demos** pane on the left, click the + for **Simulink**. Then click the + for **Simulink HDL Coder**. Then double-click the list entry for the Symmetric FIR Filter Demo.

The `sfir_fixed` model opens.

6 Select **Save As** from the Simulink **File** menu and save a local copy of `sfir_fixed.mdl` to your working directory.

7 Leave the `sfir_fixed` model open and proceed to the next section.

Initializing Model Parameters With hdlsetup

Before generating code, you must set some parameters of the model. Rather than doing this manually, use the Simulink HDL Coder M-file utility, `hdlsetup.m`. The `hdlsetup` command uses the Simulink `set_param` function to set up models for HDL code generation quickly and consistently.

To set the model parameters:

1 At the MATLAB command prompt, type

```
hdlsetup('sfir_fixed')
```

2 Select **Save** from the Simulink **File** menu, to save the model with its new settings.

Before continuing with code generation, consider the settings that `hdlsetup` applies to the model.

`hdlsetup` configures Simulink **Solver** options that are recommended or required by Simulink HDL Coder. These are

- **Type**: `Fixed-step`. Simulink HDL Coder does not currently support variable-step solvers.
- **Solver**: `discrete (no continuous states)`. Other fixed-step solvers could be selected, but this option is usually the correct one for simulating discrete systems.
- **Tasking mode**: `SingleTasking`. Simulink HDL Coder does not currently support models that execute in multitasking mode.

`hdlsetup` also configures the model start and stop times and fixed-step size as follows:

- **Start Time**: `0.0` s
- **Stop Time**: `10` s
- **Fixed step size (fundamental periodic sample time)**: `auto`

Setting **Fixed step size** to `auto` causes Simulink to choose the step size, based on the sample times specified in the model. In the demo model, only the Signal From Workspace block specifies an explicit sample time (1 s); all other blocks inherit this sample time.

The model start and stop times determine the total simulation time. This in turn determines the size of data arrays that are generated to provide stimulus and output data for generated test benches. For the demo model, computation of 10 seconds of test data does not take a significant amount of time. Computation of sample values for more complex models can be time consuming. In such cases, you may want to decrease the total simulation time.

The remaining parameters set by `hdlsetup` affect Simulink error severity levels, data logging, and model display options. If you want to view the complete set of model parameters affected by `hdlsetup`, open `hdlsetup.m` in the MATLAB editor.

The model parameter settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications. For example, `hdlsetup` sets a default **Simulation stop time** of 10 s. A total simulation time of 1000 s would be more realistic for a test of the `sfir_fixed` demo model. If you would like to change the simulation time, enter the desired value into the **Simulation stop time** field of the Simulink window.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

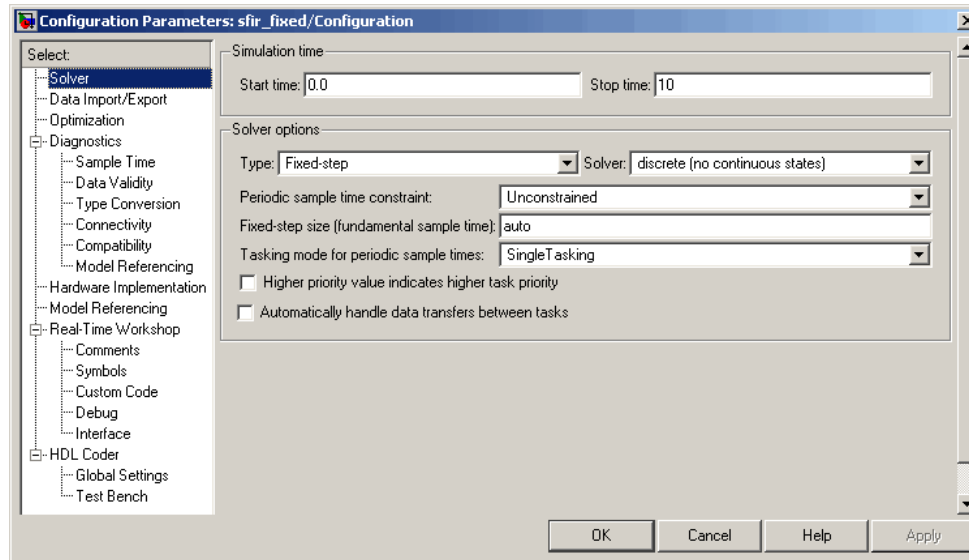
Viewing Simulink HDL Coder Options in the Configuration Parameters Dialog

The Simulink HDL Coder option settings are a category of the model’s active configuration set. You can view and edit these options in the Configuration Parameters dialog, or in the Simulink Model Explorer. This discussion uses the Configuration Parameters dialog.

To access the Simulink HDL Coder settings:

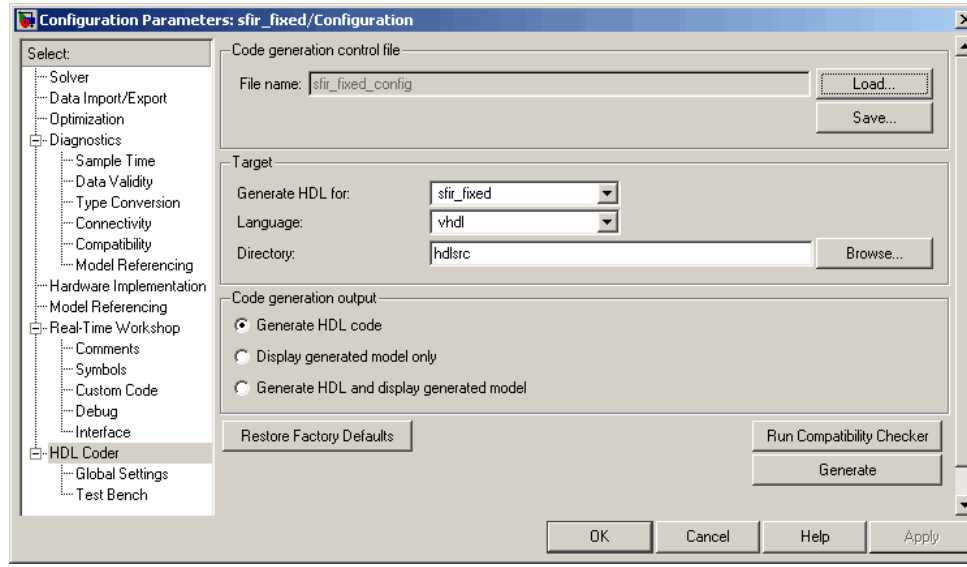
1 Select **Configuration Parameters** from the **Simulation** menu in the `sfir_fixed` model window.

The Configuration Parameters dialog opens with the **Solver** options pane displayed, as shown in the figure below.



2 Observe that the **Select** tree in the left panel of the dialog includes an **HDL Coder** category, as shown.

3 Click the **HDL Coder** category in the **Select** tree. The **HDL Coder** pane is displayed, as shown below.

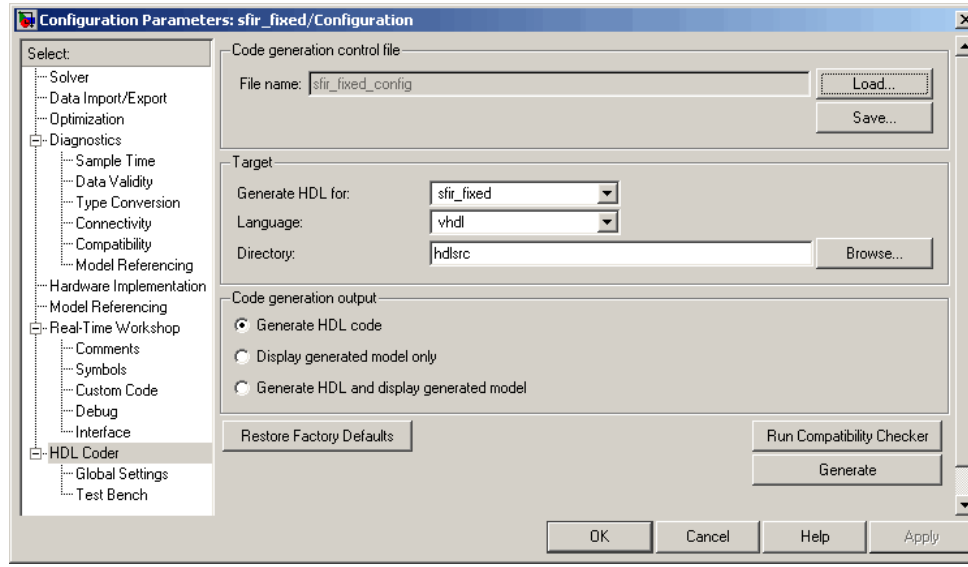


The **HDL Coder** pane contains top-level options and buttons that control the HDL code generation process. Several other categories of options are available under the **HDL Coder** entry in the **Select** tree. This exercise uses a small subset of these options, leaving the others at their default settings.

"Summary of Controls and Properties" on page 3-6 summarizes all the options available in the **HDL Coder** category.

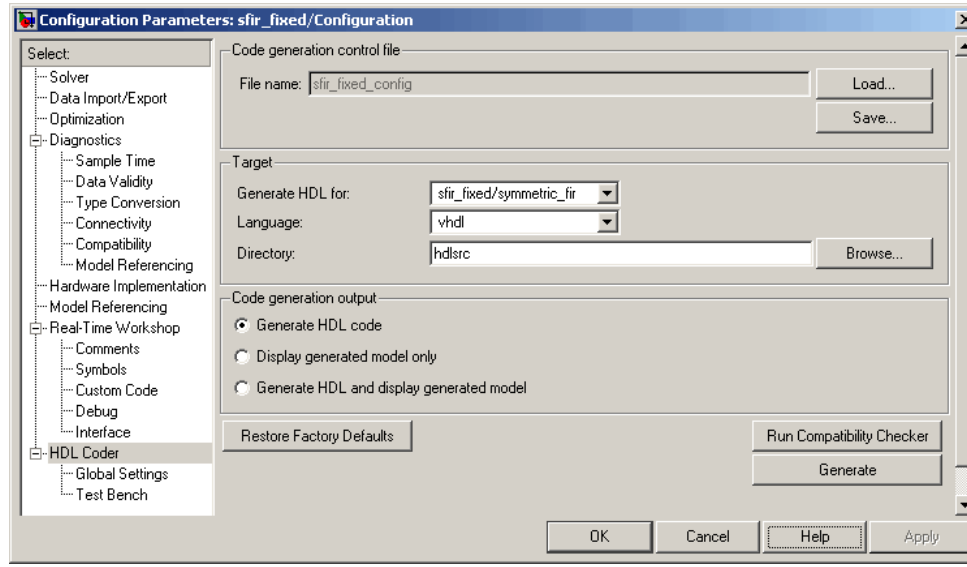
Selecting and Checking a Subsystem for HDL Compatibility

Simulink HDL Coder generates code from either the current model or from a subsystem at the root level of the current model. You use the **Generate HDL for** menu to select the model or subsystem from which code is to be generated. Each entry in the menu shows the full Simulink path to the model or one of its subcomponents. By default, the current model (`sfir_fixed`) is selected, as shown in the figure below.



In this exercise, you select and check the `symmetric_fir` subsystem for compatibility. First, select the subsystem for code generation, as follows:

1. Select `sfixed_fir/symmetric_fir` from the **Generate HDL for** menu.
2. Click **Apply**. The dialog should now appear as shown below.



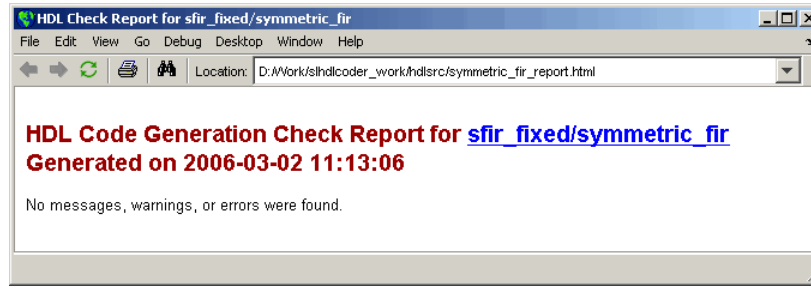
To check HDL compatibility for the subsystem:

1. Click the **Run Compatibility Checker** button.

2. The HDL compatibility checker examines the system selected in the **Generate HDL for** menu for any compatibility problems. In this case, the selected subsystem is fully HDL-compatible, and the compatibility checker displays the following message in the MATLAB Command Window.

   ```
   ### Starting HDL Check.
   ### HDL Check Complete with 0 errors, warnings and messages.
   ```

3. The compatibility checker also displays an HTML report in a Web browser, as shown below.

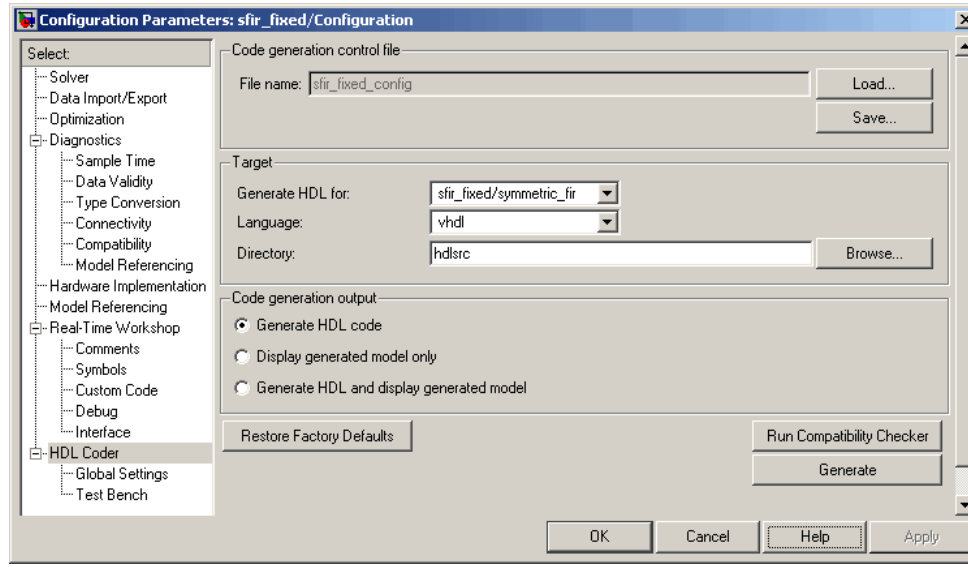


Generating VHDL Code

The top-level **HDL Coder** options are now set as follows:

- The **Generate HDL for** field specifies the `sfixed_fir/symmetric_fir` subsystem for code generation.
- The **Language** field specifies (by default) generation of VHDL code.
- The **Directory** field specifies a *target directory* that stores generated code files and scripts. The default target directory is a subdirectory of your working directory, named `hdlsrc`.

The figure below shows these settings.



Before generating code, select **Current Directory** from the **Desktop** menu in the MATLAB window. This displays the MATLAB Current Directory browser, which lets you easily access your working directory and the files that will be generated within it.

To generate code:

1. Click the **Generate** button.

2. Simulink HDL Coder generates code and displays progress messages. The process should complete successfully with the message

   ```
   ### HDL Code Generation Complete.
   ```

   Observe that the names of generated files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB editor.

   Note that Simulink HDL Coder compiles the model before generating code. Depending on model display options (for example, sample time colors, port data types, etc.), the appearance of the model may change after code generation.

3 A folder icon for the `hdlsrc` directory is now visible in the Current Directory browser. To view generated code and script files, double-click the `hdlsrc` folder icon.

4 The files that were generated in the `hdlsrc` directory are

- `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir` filter.
- `symmetric_fir_compile.do`: ModelSim compilation script (`vcom` command) to compile the generated VHDL code.
- `symmetric_fir_synplify.tcl`: Synplify synthesis script.
- `symmetric_fir_map.txt`: Mapping file. This report file maps generated entities (or modules) to the Simulink subsystems that generated them (see "Code Tracing Using the Mapping File" on page 6-5).

5 To view the generated VHDL code in the MATLAB editor, double-click the `symmetric_fir.vhd` file icon in the Current Directory browser.

At this point it is suggested that you study the `ENTITY` and `ARCHITECTURE` definitions while referring to "HDL Code Generation Defaults" on page 12-13 in the `makehdl` reference documentation. The reference documentation describes the default naming conventions and correspondences between the elements of a Simulink model (subsystems, ports, signals, etc.) and elements of generated HDL code.

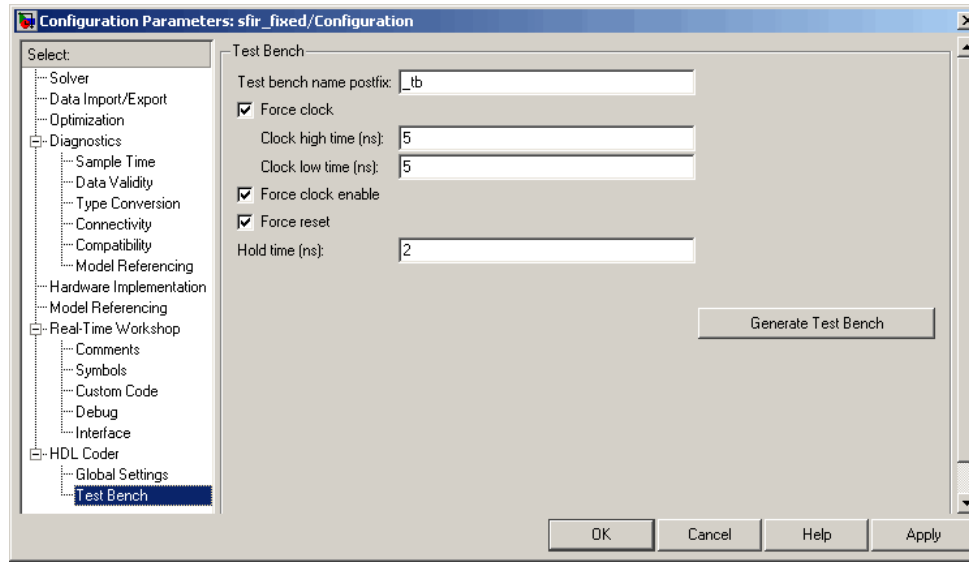
6 Before proceeding to the next section, close any files you have opened in the MATLAB editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `sl_hdlcoder_work` directory.

Generating VHDL Test Bench Code

At this point, the **Generate HDL for**, **Language**, and **Directory** fields are set as they were in the previous section. Accordingly, you can now generate VHDL test bench code to drive the VHDL code generated previously for the `sfixed_fir/symmetric_fir` subsystem. The code will be written to the same target directory as before.

To generate a test bench:

1 Click the **Test Bench** entry in the **HDL Coder** list in the **Select** tree. The **Test Bench** pane is displayed, as shown below.



2 Click the **Generate Test bench** button.

3 Simulink HDL Coder generates code and displays progress messages in the MATLAB window. The process should complete successfully with the message

```
### HDL TestBench Generation Complete.
```

4 The files that were generated in the `hdlsrc` directory are

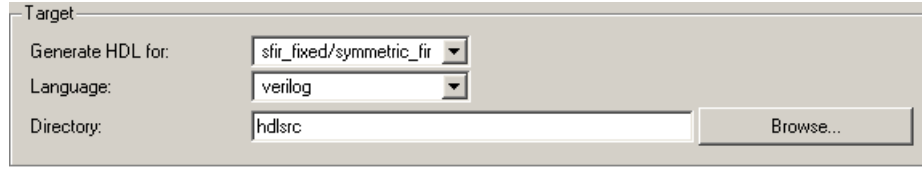
- `symmetric_fir_tb.vhd`: VHDL test bench code and generated test and output data.
- `symmetric_fir_tb_compile.do`: ModelSim compilation script (`vcom` commands). This script compiles and loads both the entity to be tested (`symmetric_fir.vhd`) and the test bench code (`symmetric_fir_tb.vhd`).
- `symmetric_fir_tb_sim.do`: ModelSim script to initialize the simulator, set up **wave** window signal displays, and run a simulation.

Verifying Generated Code

You can now take the generated code and test bench to an HDL simulator for simulated execution and verification of results. See “Simulating and Verifying Generated HDL Code” on page 2-29 for an example of how to use generated test bench and script files with the Mentor Graphics HDL simulator, ModelSim SE/PE.

Generating Verilog Model and Test Bench Code

The procedure for generating Verilog code is the same as for generating VHDL code (see “Generating a VHDL Entity from a Subsystem” on page 2-9 and “Generating VHDL Test Bench Code” on page 2-11), except that you should select `verilog` from the **Language** field of the **HDL Coder** options, as shown below.



Simulating and Verifying Generated HDL Code

Note This section requires the use of the Mentor Graphics HDL simulator, ModelSim SE/PE.

This section assumes that you have generated code from the `sfir_fixed` demo model as described in either of the following exercises:

- "Generating HDL Code Using MATLAB Commands " on page 2-6
- "Generating HDL Code in the Simulink GUI" on page 2-15

In this section you compile and run a simulation of the previous generated model and test bench code in ModelSim. The scripts generated by Simulink HDL Coder let you do this with just a few simple commands. The procedure is the same, whether you generated code in the command line environment or in the Simulink environment.

To run the simulation:

1. Start ModelSim.
2. Set the ModelSim working directory to the directory in which you previously generated code.

   ```
   ModelSim>cd D:/work/sl_hdlcoder_work/hdlsrc
   ```

3. Use the generated compilation script to compile and load the generated model and text bench code. The following listing shows the command and responses from ModelSim.

   ```
   ModelSim>do symmetric_fir_tb_compile.do
   # Model Technology ModelSim SE vcom 6.0 Compiler 2004.08 Aug 19 2004
   # -- Loading package standard
   # -- Loading package std_logic_1164
   # -- Loading package numeric_std
   # -- Compiling entity symmetric_fir
   # -- Compiling architecture rtl of symmetric_fir
   # Model Technology ModelSim SE vcom 6.0 Compiler 2004.08 Aug 19 2004
   ```

```
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling package symmetric_fir_tb_pkg
# -- Compiling package body symmetric_fir_tb_pkg
# -- Loading package symmetric_fir_tb_pkg
# -- Loading package symmetric_fir_tb_pkg
# -- Compiling entity symmetric_fir_tb
# -- Compiling architecture rtl of symmetric_fir_tb
# -- Loading entity symmetric_fir
```

4 Use the generated simulation script to execute the simulation. The following listing shows the command and responses from ModelSim. The warning messages are benign.

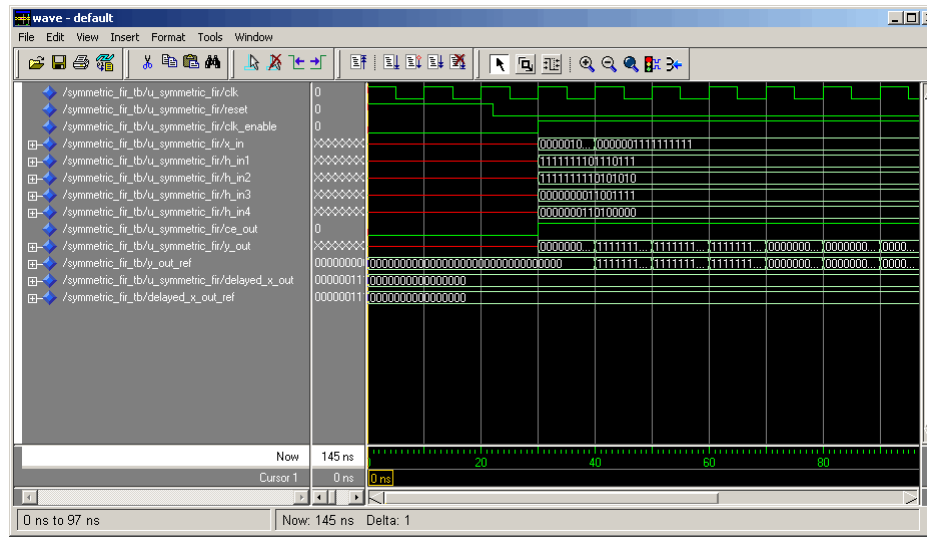
```
ModelSim>do  symmetric_fir_tb_sim.do
# vsim work.symmetric_fir_tb
# Loading D:\Applications\ModelTech_6_0\win32/../std.standard
# Loading D:\Applications\ModelTech_6_0\win32/../ieee.std_logic_1164(body)
# Loading D:\Applications\ModelTech_6_0\win32/../ieee.numeric_std(body)
# Loading work.symmetric_fir_tb_pkg(body)
# Loading work.symmetric_fir_tb(rtl)
# Loading work.symmetric_fir(rtl)
# ** Warning: NUMERIC_STD."<": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /symmetric_fir_tb
.
.
.
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ns  Iteration: 1  Instance: /symmetric_fir_tb
# ** Note: **************TEST COMPLETED **************
#    Time: 140 ns  Iteration: 1  Instance: /symmetric_fir_tb
```

The test bench termination message indicates that the simulation has run to completion successfully, without any comparison errors.

```
# ** Note: **************TEST COMPLETED **************
```

5 The simulation script displays all inputs and outputs in the model (including the reference signals `y_out_ref` and `delayed_x_out_ref`) in

the ModelSim **wave** window. The figure below shows the signals displayed in the **wave** window.



6 Exit ModelSim when you are through viewing signals.

7 Close any files you have opened in the MATLAB editor. Then, click the Go Up One Level button in the Current Directory browser, to set the current directory back to your `sl_hdlcoder_work` directory.

3

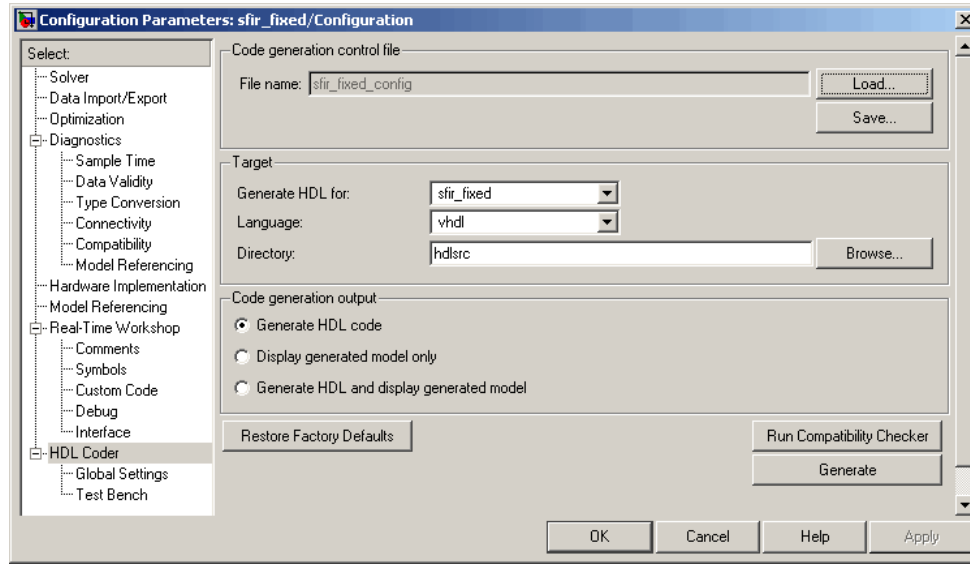
Code Generation Options in the Simulink HDL Coder GUI

Viewing and Setting HDL Coder Options

The Simulink Configuration Parameters dialog and the Simulink Model Explorer let you view and set the HDL code generation options, parameters, and controls within a Simulink configuration set. The following topics summarize these options:

HDL Coder Options in the Configuration Parameters Dialog

The figure below shows the top-level **HDL Coder** options pane as displayed in the Configuration Parameters dialog. To open this dialog, select **Simulation > Configuration Parameters** in the Simulink window. Then select **HDL Coder** from the list on the left.



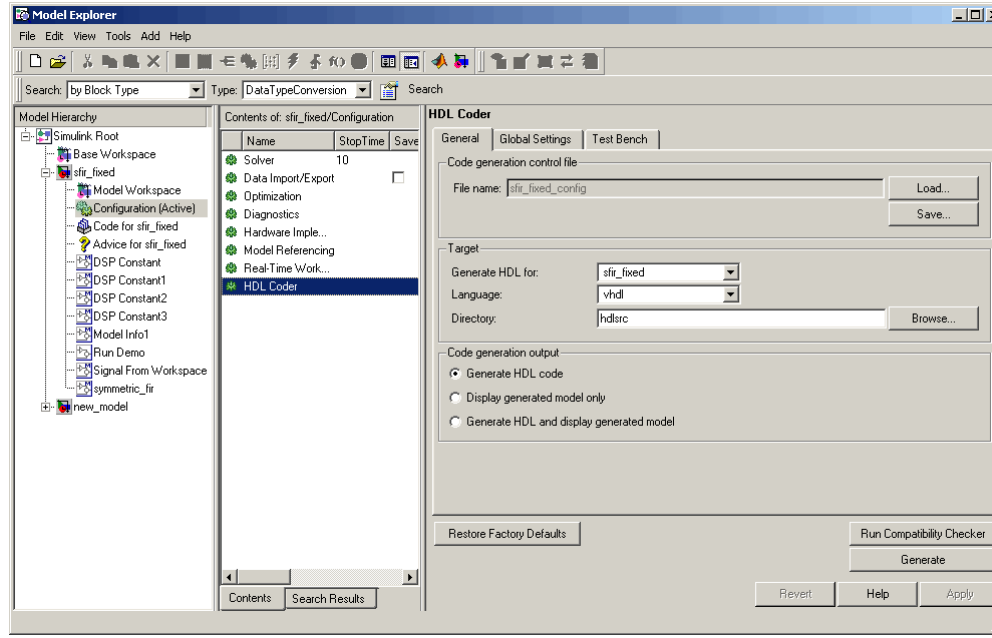
If you are not familiar with Simulink configuration sets and how to view and edit them in the Configuration Parameters dialog, see the “Configuration Sets” and “Configuration Parameters Dialog Box” sections of the Simulink documentation.

Note When the **HDL Coder** options pane of the Configuration Parameters dialog is selected, clicking the Help button displays general help for the Configuration Parameters dialog.

HDL Coder Options in the Model Explorer

The figure below shows the top-level **HDL Coder** options pane as displayed in the **Dialog** pane of the Model Explorer.

To view this dialog, select **View > Model Explorer** in the Simulink window. Then select your model’s active configuration set in the **Model Hierarchy** tree on the left. Then, select **HDL Coder** from the list in the **Contents** pane.

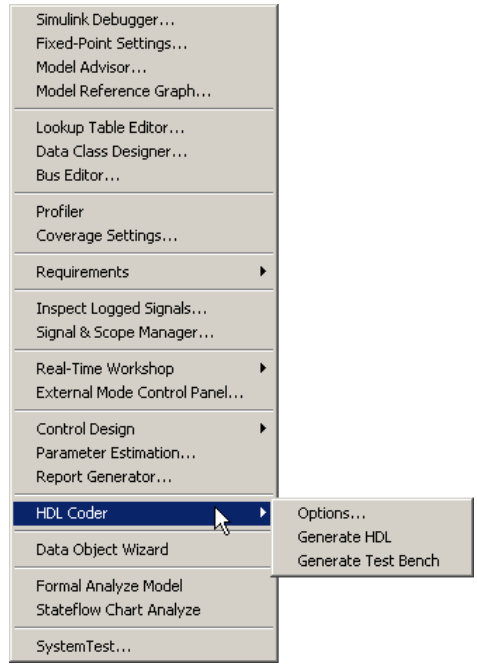


When the **HDL Coder** options pane is selected in the Model Explorer as shown above, clicking the Help button displays the documentation specific to the current tab.

If you are not familiar with the Model Explorer, see “Exploring, Searching, and Browsing Models” in the Simulink documentation.

HDL Coder Menu

The **HDL Coder** submenu of the Simulink **Tools** menu (see figure below) provides shortcuts to the HDL code generation options. You can also use this menu to initiate code generation.



The **HDL Coder** submenu options are

- **Options**: Open the **HDL Coder** options pane in the Configuration Parameters dialog.

- **Generate HDL**: Initiate HDL code generation; equivalent to the **Generate** button in the Configuration Parameters dialog or Model Explorer.
- **Generate Test Bench**: Initiate test bench code generation; equivalent to the **Generate Test Bench** button in the Configuration Parameters dialog or Model Explorer.

Summary of Controls and Properties

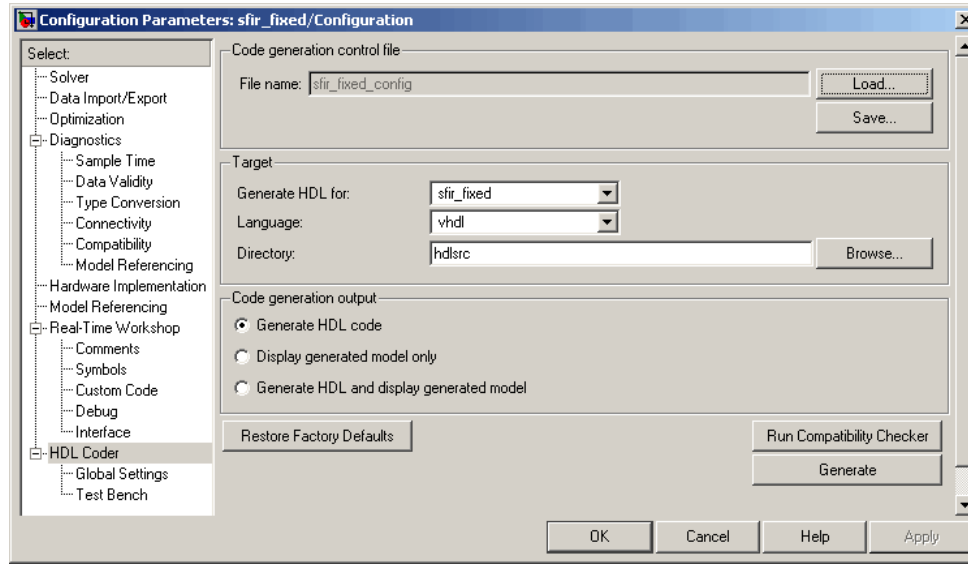
Each code generation option displayed on the GUI corresponds to a `makehdl` or `makehdltb` property. The tables in each section below contain hyperlinks to the appropriate property or function reference pages.

Illustrations show the default settings for all options.

The following sections summarize controls and properties in each pane of the Simulink HDL Coder GUI, as displayed in the Configuration Parameters dialog:

HDL Coder Pane

The top-level **HDL Coder** pane contains buttons that initiate code generation and compatibility checking, and sets parameters that affect overall operation of code generation.

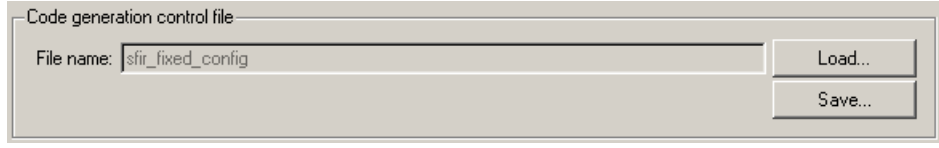


Main Pane

Control	Description
Generate	Initiates code generation for the system selected in the **Generate HDL for** menu. See also `makehdl`.
Run Compatibility Checker	Invokes the compatibility checker to examine the system selected in the **Generate HDL for** menu for any compatibility problems. See also `checkhdl`.
Restore Factory Defaults	Sets all Simulink HDL Coder properties to their default values. Unlinks the current code generation file (if any) from the model, and clears the **File name** field. **Restore Factory Defaults** resets all HDL code generation settings. This action cannot be cancelled or undone. To recover previous settings, you must close the model without saving it, and then reopen it.

Code Generation Control File Pane

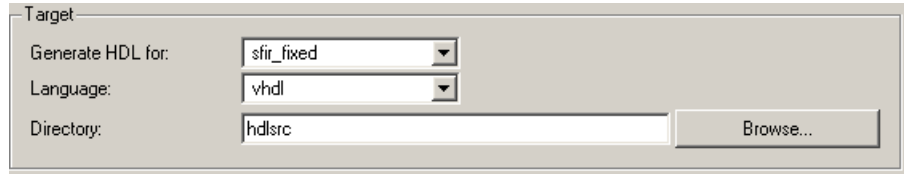
This pane contains options and controls that let you attach a code generation control file to your model. See Chapter 4, “Code Generation Control Files” for a detailed discussion of the structure and use of control files.



Control	Description
File Name	Displays the path and file name of the currently selected control file (if any). This is a display-only field. To select a control file, use the **Load** button. To clear the **File Name** field and unlink the current control file, use the **Restore Factory Defaults** button.
Load	When you click the **Load** button, a standard file selection dialog opens. You can then navigate to and select a control file and load it into memory.
Save	When you click the **Save** button, a standard file save dialog opens. You can then save current Simulink HDL Coder settings to a specified control file. Note that a full path to the control file is saved. If you want to specify a relative path, use the `HDLControlFiles` property of the makehdl command. (See “Using Control Files in the Code Generation Process” on page 4-12).

Target Pane

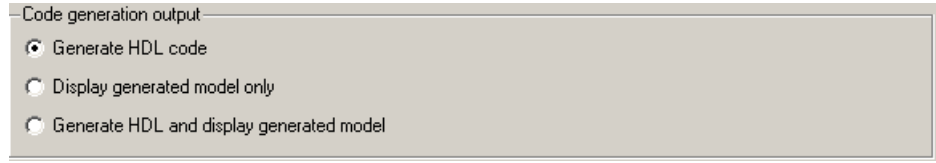
This pane contains top-level code generation options.



Option	Description
Generate HDL for	This pop-up menu selects the subsystem or model from which code is generated. The menu displays the Simulink path to the root model and to all root-level subsystems in the model. See also `makehdl`.
Language	This pop-up menu selects the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language. The default target language is VHDL. See also `TargetLanguage`.
Directory **Browse**	Specifies the directory into which code is generated. The selected directory is referred to as the target directory. The default target directory is a subdirectory of your working directory, named `hdlsrc`. You can enter a path to the target directory, or click the **Browse** button to navigate to and select a directory. See also `TargetDirectory`.

Code Generation Output Pane

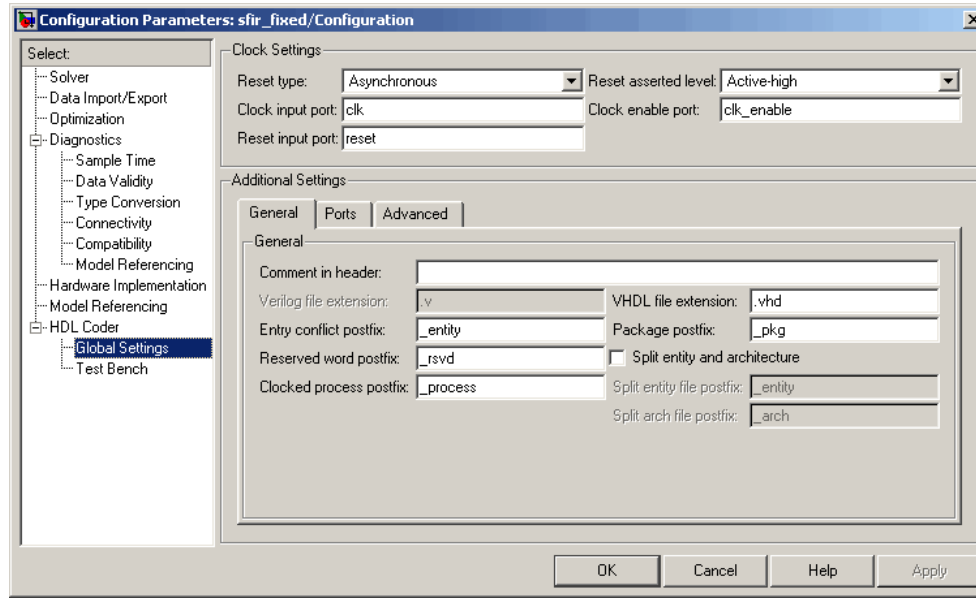
This pane contains options related to the creation and display of generated models. See also Chapter 5, “Generating Bit-True Cycle-Accurate Models”.



Option	Property
Generate HDL code	Generate HDL code without displaying the generated model. This is the default.
Display generated model only	Display the generated model without generating HDL code.
Generate HDL and display generated model	Display the generated model after HDL code generation completes.

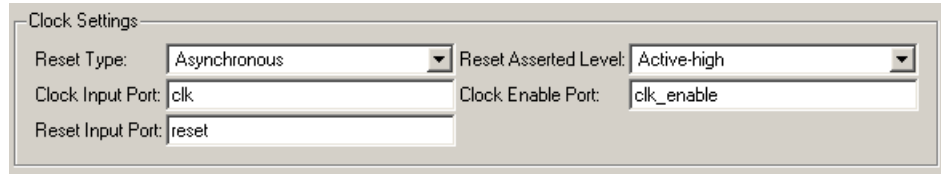
Global Settings Pane

The **Global Settings** pane lets you set options to specify detailed characteristics of the generated code, such as HDL element naming and whether certain optimizations are applied.



Clock Settings Pane

The **Clock Settings** pane contains options related to the operation of clock and reset signals in the generated HDL code.

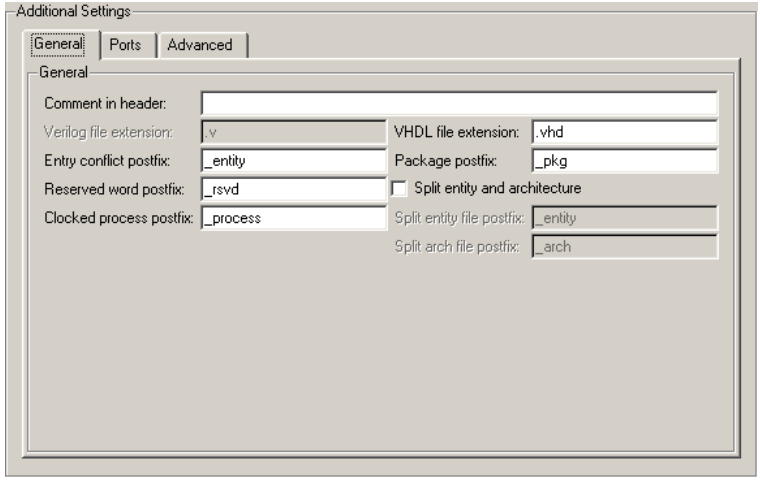


Option	Description
Reset Type	Specifies whether to use asynchronous or synchronous reset logic when generating HDL code for registers. Default: `Asynchronous`. See also `ResetType`.

Option	Description
Reset Asserted Level	Specifies whether the asserted (active) level of reset input signal is active-high (`1`) or active-low (`0`). Default: `Active-high`. See also `ResetAssertedLevel`.
Clock Input Port	Specifies the name for the clock input port in generated HDL code. Default: `clk`. See also `ClockInputPort`.
Clock Enable Port	Specifies the name for the clock enable input port in generated HDL code. Default: `clk_enable`. See also `ClockEnableInputPort`.
Reset Input Port	Specifies the name for the reset input port in generated HDL code. Default: `reset`. See also `ResetInputPort`.

Additional Settings : General Pane

This pane contains settings related to file naming for generated code, and comment generation.

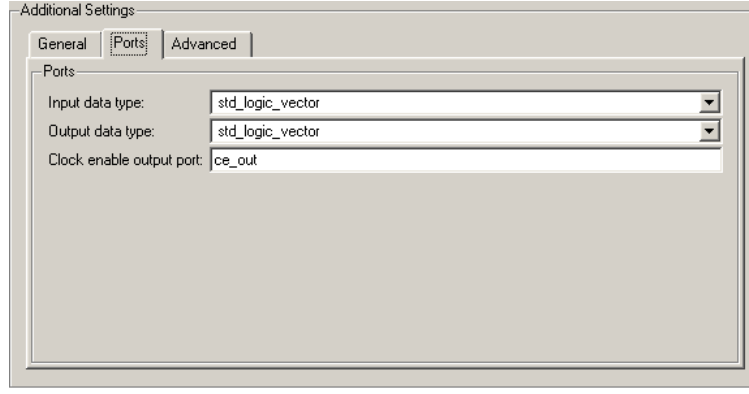


Option	Description
Comment in header	Text entered in this field generates a comment line in the header of generated model and test bench files. See also `UserComment`.
Verilog file extension	This field specifies the file name extension for generated Verilog files. The default extensions is `.v`. **Verilog file extension** is enabled when the target language is Verilog. See also `VerilogFileExtension`.
VHDL file extension	This field specifies the file name extension for generated VHDL files. The default extensions is `.vhd`. **VHDL file extension** is enabled when the target language is VHDL. See also `VHDLFileExtension`.
Entity conflict postfix	The string entered in this field is used to resolve duplicate VHDL entity or Verilog module names in generated code. The default is `_entity`. See also `EntityConflictPostfix`.
Package postfix	The string entered in this field is appended to the model or subsystem name to form the name of a VHDL package file.**Package postfix** is enabled when the target language is VHDL. The default is `_pkg`. See also `PackagePostfix`.
Reserved word postfix	The string entered in this field is appended to value names, postfix values, or labels in generated code that conflict with VHDL or Verilog reserved words. The default is `_rsvd`. See also `ReservedWordPostfix`.
Split entity and architecture	**Split entity and architecture** is enabled when the target language is VHDL. When this option is deselected (the default), VHDL entity and architecture code is written to a single VHDL file. When this option is selected VHDL entity and architecture definitions are written to separate files. See also `SplitEntityArch`.

Option	Description
Split entity file postfix	**Split entity file postfix** is enabled when **Split entity and architecture** is selected. The string entered in this field is appended to the model name to form the name of a generated VHDL entity file. The default is `_entity`. See also `SplitEntityFilePostfix`.
Split arch file postfix	**Split arch file postfix** is enabled when **Split entity and architecture** is selected. The string entered in this field is appended to the model or subsystem name to form the name of the file containing the model's VHDL architecture. The default is `_arch`. See also `SplitArchFilePostfix`.
Clocked process postfix	Specifies a string to append to HDL clock process names. Simulink HDL Coder uses process blocks for register operations. The label for each block drives from a register name and the postfix. The default is `_process`. See also `ClockProcessPostfix`.

Additional Settings : Ports Pane

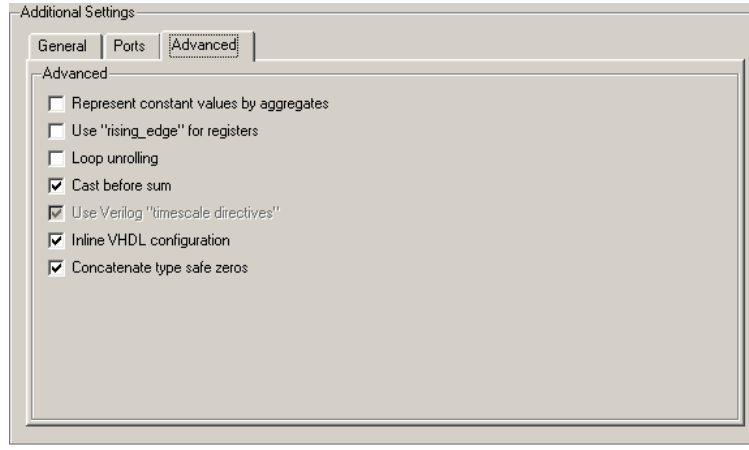
This pane contains options related to input, output, and clock enable output ports.



Option	Description
Input data type	Specifies the HDL data type for the model's input ports. For VHDL, the options are • `std_logic_vector`: Specifies VHDL type `STD_LOGIC_VECTOR`. • `signed/unsigned`: Specifies VHDL type `SIGNED` or `UNSIGNED`. **Input data type** is disabled when the target language is Verilog,. In generated Verilog code, the data type for all ports is `wire`. See also `InputType`.
Output data type	Specifies the HDL data type for the model's output ports. For VHDL, the options are • `std_logic_vector`: Specifies VHDL type `STD_LOGIC_VECTOR`. • `signed/unsigned`: Specifies VHDL type `SIGNED` or `UNSIGNED`. **Output data type** is disabled when the target language is Verilog. In generated Verilog code, the data type for all ports is `wire`. See also `OutputType`.
Clock enable output port	Specifies the name for the generated clock enable output. The default is `ce_out`.See also `ClockEnableOutputPort`.

Additional Settings : Advanced Pane

This pane contains advanced settings related to detailed characteristics of generated code. Most of these options are specific to either VHDL or Verilog.



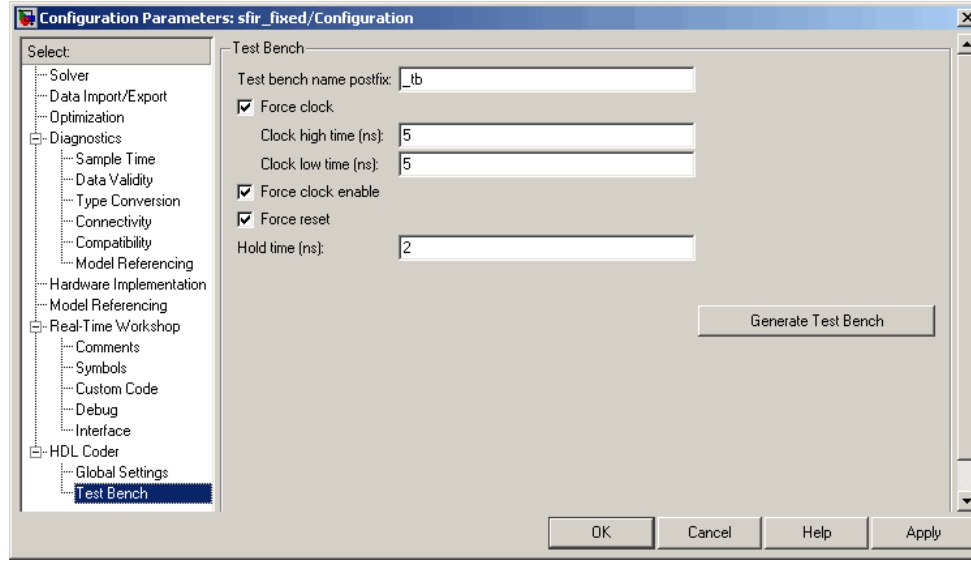
Option	Description
Represent constant values by aggregates	**Represent constant values by aggregates** is enabled when the target language is VHDL. When this option is deselected (the default), the coder represents constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. When this option is selected, all constants are represented as aggregates. See also `UseAggregatesForConst`.
Use "rising edge" for registers	**Use "rising edge" for registers** is enabled when the target language is VHDL. When this option is deselected (the default), generated code checks for clock events when operating on registers. When this option is selected, generated code uses the VHDL `rising_edge` function to check for rising edges when operating on registers. See also `UseRisingEdge`.

Option	Description
Loop unrolling	**Loop unrolling** is enabled when the target language is VHDL. When this option is deselected (the default), `FOR` and `GENERATE` loops are included in generated VHDL code. When this option is deselected, are unrolled and omitted from generated VHDL code. See also `LoopUnrolling`.
Cast before sum	When this option is selected (the default), operands in addition and subtraction operations are type cast to the result type before executing the operation. When this option is selected, operand types are preserved during addition and subtraction operations and the result is then converted to the desired result type. See also `CastBeforeSum`.
Use Verilog "timescale directives"	**Use Verilog "timescale directives"** is enabled when the target language is Verilog. When this option is selected (the default), the coder uses compiler `'timescale` directives in generated Verilog code. When this option is deselected, the coder suppresses compiler `'timescale` directives in generated Verilog code. This setting does not affect the generated test bench. See also `UseVerilogTimescale`.

Option	Description
Inline VHDL configuration	**Inline VHDL configuration** is enabled when the target language is VHDL. When this option is selected (the default), VHDL configurations are generated in any file that instantiates a component. When this option is deselected, generated configurations are suppressed and user-defined configurations are required.See also `InlineConfigurations`.
Concatenate type safe zeros	When this option is selected, (the default), the code uses the type-safe syntax, `'0' & '0'`, for concatenated zeros. Typically, this syntax is preferred. See also `SafeZeroConcat`.

Test Bench Pane

The **Test Bench** pane lets you set options that determine characteristics of generated test bench code.



Option or Control	Description
Generate Test Bench	Initiates test bench code generation for the model or system selected in the **Generate HDL for** menu. See also `makehdltb`.
Test bench name postfix	Specifies a suffix appended to the test bench name. The default is `_tb`. See also `TestBenchPostFix`.
Force clock	When this option is selected (the default), the test bench forces the clock input signals. See also `ForceClock`.
Clock high time	This option is enabled when **Force clock** is selected. Specify the period, in nanoseconds, during which a test bench drives clock input signals high. The default is 5 ns. See also `ClockHighTime`.
Clock low time	This option is enabled when **Force clock** is selected. Specify the period, in nanoseconds, during which a test bench drives clock input signals low. The default is 5 ns. See also `ClockLowTime`.
Force clock enable	When this option is selected (the default), the test bench forces the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value. See also `ForceClockEnable`.
Force reset	When this option is selected (the default), the test bench forces the reset input signals. See also `ForceReset`.
Hold time (ns)	This option specifies a hold time, in nanoseconds, for input signals and forced reset input signals. The default is 2 ns. During the hold interval, the model's data input signals and forced reset input signals are held past the clock rising edge. See also `HoldTime`.

4

Code Generation Control Files

Overview of Control Files

Code generation control files (referred to in this document as *control files*) let you extend the HDL code generation process and direct its details. A control file is an M-file that you attach to your model, using either the `makehdl` command or the Simulink Configuration Parameters dialog. You do not need to know any internal details of the code generation process to use a control file.

In the current release, control files support the following statement types:

- *Selection/action* statements provide a general framework for the application of different types of transformations to selected model components. Selection/action statements *select* a group of blocks within your model, and specify an *action* to be executed when code is generated for each block in the selected group.

  Selection criteria include block type and location within the model. For example, you might select all built-in Gain blocks at or below the level of a certain subsystem within your model.

  A typical action applied to such a group of blocks would be to direct the code generator to execute a specific *block implementation method* when generating HDL code for the selected blocks. For example, for Gain blocks, you might choose a method that generates code that is optimized for speed or chip area.

- *Property setting* statements let you
  - Select the model or subsystem from which code is to be generated.
  - Set the values of code generation properties to be passed to the code generator. The properties and syntax are the same as those used for the `makehdl` command.
  - Set up default or template HDL code generation settings for your organization.

Selectable Block Implementations

Selection/action statements provide a general framework that lets you define how Simulink HDL Coder acts upon selected model components. The current release supports one such action: execution of block implementation methods.

Block implementation methods are code generator components that emit HDL code for the blocks in a Simulink model. This document refers to block implementation methods as *block implementations* or simply *implementations*.

Simulink HDL Coder provides at least one block implementation for every supported block. This is called the *default implementation*. In addition, Simulink HDL Coder provides selectable alternate block implementations for certain block types. Each implementation is optimized for different characteristics, such as speed or chip area. For example, you can choose Gain block implementations that use canonic signed digit (CSD) techniques (reducing area), or use a default implementation that retains multipliers.

Implementation Mappings

Control files let you specify one or more *implementation mappings* that control how HDL code is to be generated for a specified group of blocks within the model. An implementation mapping is an association between a selected block or set of blocks within the model and a block implementation.

To select the set of blocks to be mapped to a block implementation, you specify

- A `modelscope`: a Simulink block path (which could incorporate an entire model or sublevel of the model, or a specific subsystem or block)
- A `blocktype`: a Simulink block type that corresponds to the selected block implementation

During code generation, each defined `modelscope` is searched for instances of the associated `blocktype`. For each such block instance encountered, the code generator uses the selected block implementation.

Control File Demo

The “Getting Started with Control Files” demo illustrates the use of simple control files to define implementation mappings and generate Verilog code. The demo is located in the **Demos** pane on the left of the MATLAB Help browser. To run the demo, select **Simulink > Simulink HDL Coder > Getting Started with Control Files** in the **Demos** pane. Then follow the demo instructions.

Structure of a Control File

The required elements for a code generation control file are as follows:

- A control file is an M-file that implements a single function, which is invoked during the code generation process.

  The function must instantiate a *code generation control object*, set its properties, and return the object to the code generator.

  Setting up a code generation control object requires the use of a small number of methods, as described in "Code Generation Control Objects and Methods" on page 4-6. You do not need to know internal details of the code generation control object or the class to which it belongs.

  The object is constructed using the `hdlnewcontrol` function. The argument to `hdlnewcontrol` is the name of the control file itself. Use the MATLAB `mfilename` function to pass in the file name, as shown in the following example.

  ```
  function c = dct8config
  c = hdlnewcontrol(mfilename);

  % Set target language for Verilog.
  c.set('TargetLanguage','Verilog');

  % Set top-level subsystem from which code is generated.
  c.generateHDLFor('dct8_fixed/OneD_DCT8');
  ```

- Following the constructor call, your code will invoke methods of the code generation control object. The example above calls the `set` and `generateHDLFor` methods. These and all other public methods of the object are discussed in "Code Generation Control Objects and Methods" on page 4-6.
- Your control file must be attached to your Simulink model before code generation, as described in "Using Control Files in the Code Generation Process" on page 4-12. The interface between the code generator and your attached control file is automatic.
- A control file is normally located in either the current working directory, or a directory that is in the MATLAB path.

If you want to locate a control file elsewhere, you should specify an explicit path to the control file when you attach it to your model.

However, your control files should not be located within the MATLAB directory tree because they could be overwritten by subsequent MATLAB installations.

Code Generation Control Objects and Methods

Code generation control objects are instances of the class `slhdlcoder.ConfigurationContainer`. This section describes the methods of that class that you can use in your control files. Other methods of this class are for MathWorks internal use only. The methods are described in the following sections:

hdlnewcontrol

The `hdlnewcontrol` function constructs a code generation control object. The syntax is

```
object = hdlnewcontrol(mfilename);
```

The argument to `hdlnewcontrol` is the name of the control file itself. Use the MATLAB `mfilename` function to pass in the file name string.

forEach

This method establishes an implementation mapping between an HDL block implementation and a selected block or set of blocks within the model. The syntax is

```
object.forEach({'modelscopes'}, ...
        'blocktype', {'block_parms'}, ...
        'implementation', {'implementation_parms'})
```

The `forEach` method selects a set of Simulink blocks (`modelscopes`) that is searched, during code generation, for instances of a specified type of block (`blocktype`). Code generation for each block instance encountered uses the HDL block implementation specified by the `implementation` parameter.

Note You can use the `hdlnewforeach` function to generate `forEach` method calls for insertion into your control files. See "Generating Selection/Action Statements with the hdlnewforeach Function" on page 4-16 for more information.

The following table summarizes the arguments to the `forEach` method.

Argument	Type	Description
`modelscopes`	String or cell array of strings	Strings defining one or more Simulink paths: `{'path1' 'path2'...'pathN'}` Each such path defines a `modelscope`: a set of Simulink blocks that participate in an implementation mapping. The selected set of blocks in a `modelscope` could include the entire model, all blocks at a specified level of the model, or a specific block or subsystem. A path terminating in a wildcard character (`'*'`) indicates inclusion of all blocks at or below the model level specified by the path. Supported syntax for `modelscope` paths is • `'model/*'`: all blocks in the model • `'model/subsyslevel/block'`: a specific block within a specific level of the model • `'model/subsyslevel/subsystem'`: a specific subsystem block within a specific level of the model • `'model/subsyslevel/*'`: any block within a specific level of the model See also "Resolution of modelscopes" on page 4-9.

Argument	Type	Description

`blocktype`	String	Simulink block specification that identifies the type of block that is to be mapped to the HDL block implementation. The syntax for a block specification is the same as that used in the Simulink `add-block` command. For built-in Simulink blocks, the `blocktype` is of the form
		`'built-in/blockname'`
		For other blocks, `blocktype` must include the library containing the block, for example:
		`'dsparch4/Digital Filter'`
		If the block is contained in a sublibrary, the full path from the top-level library must be included.

`block_parms`	Cell array of strings	Reserved for future use; not supported in the current release. Pass in an empty cell array (`{}`) as placeholder.
`implementation`	String	An HDL block implementation to be used in code generation for all blocks that meet the `modelscope` and `blocktype` search criteria. An implementation is specified in the form `package.class`, for example: `hdldefaults.GainMultHDLEmission` "Specifying Block Implementations and Parameters in the Control File" on page 4-16 lists available implementations.
`implementation_parms`	Cell array of strings	Reserved for future use; not supported in the current release. Pass in an empty cell array (`{}`) as placeholder.

Resolution of modelscopes

A possible conflict exists in the `forEach` specifications in the following example:

```
% 1. Use default (multipliers)  Gain block implementation
% for one specific Gain block within OneD_DCT8 subsystem
c.forEach('dct8_fixed/OneD_DCT8/Gain14',...
         'built-in/Gain', {},...
         'hdldefaults.GainMultHDLEmission');
% 2. Use factored CSD Gain block implementation
% for all Gain blocks at or below level of OneD_DCT8 subsystem.
 c.forEach('dct8_fixed/OneD_DCT8/*',...
         'built-in/Gain', {},...
         'hdldefaults.GainFCSDHDLEmission');
```

The first `forEach` call defines an implementation mapping for a specific block within the subsystem `OneD_DCT8`. The second `forEach` call defines a different implementation mapping for all blocks within or below the subsystem `OneD_DCT8`.

Simulink HDL Coder resolves such ambiguities by always giving higher priority to the more specific `modelscope`. In the above example, the `Gain14` block uses the `hdldefaults.GainMultHDLEmission` implementation, while all other blocks within or below the subsystem `OneD_DCT8` use the `hdldefaults.GainFCSDHDLEmission` implementation.

Five levels of `modelscope` priority from most specific (1) to least specific (5) are defined:

1 `A/B/C/block`

2 `A/B/C/*`

3 `A/B/*`

4 `*`

5 Unspecified. Use MathWorks default implementation.

forAll

This method is a shorthand form of `forEach`. Only one `modelscope` path is specified. The `modelscope` argument is specified as a string (not a cell array) and it is implicitly terminated with `'/*'`. The syntax is

```
object.forAll('modelscope', ...
        'blocktype', {'block_parms'}, ...
        'implementation', {'implementation_parms'})
```

All other arguments are the same as those described for "forEach" on page 4-6.

set

The `set` method sets one or more code generation properties. The syntax is

```
object.set('PropertyName', PropertyValue,...)
```

The argument list specifies one or more code generation options as property/value pairs. You can set any of the code generation properties documented in Chapter 11, "Properties — Alphabetical List", *except* the `HDLControlFiles` property.

Note If you specify the same property in both your control file and your `makehdl` command, the property will be set to the value specified in the control file.

Likewise, when generating code via the GUI, if you specify the same property in both your control file and the **HDL Coder** options panes, the property will be set to the value specified in the control file.

---

generateHDLFor

This method selects the model or subsystem from which code is to be generated. The syntax is

```
object.generateHDLFor('simulinkpath')
```

The argument is a string specifying the full Simulink path to the model or subsystem from which code is to be generated.

Use of this method is optional. You can specify the same parameter in the **Generate HDL for** menu in the **HDL Coder** pane of the Configuration Parameters dialog, or in a `makehdl` command.

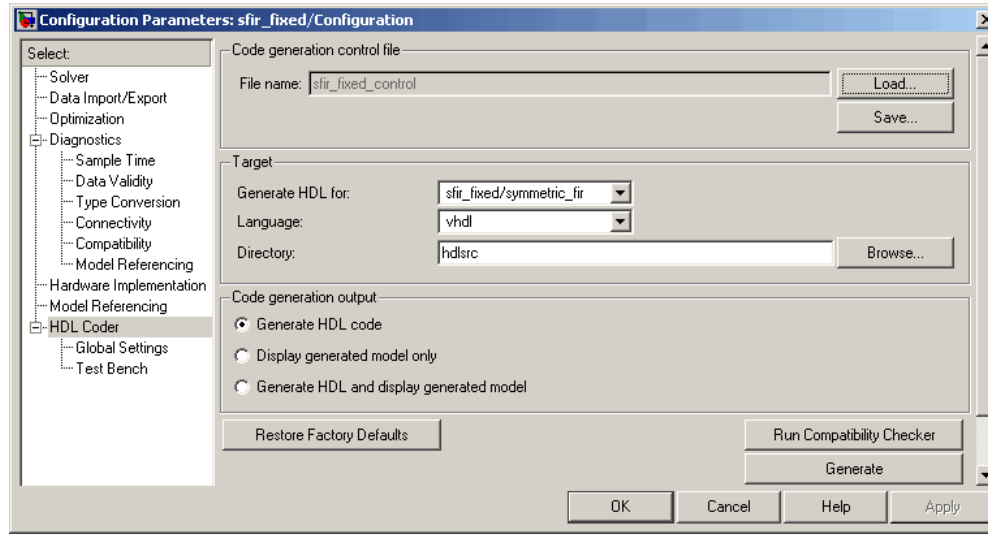
Using Control Files in the Code Generation Process

Creating a Control File

You can create a control file manually using the MATLAB editor or some other text editor. See “Structure of a Control File” on page 4-4 to make sure your files are set up correctly.

You can also use the GUI to save your current Simulink HDL Coder settings to a control file, as follows:

1. Open the Configuration Parameters dialog and select the **HDL Coder** options pane.
2. In the **Code generation control file** subpane, click the **Save** button.
3. A standard file dialog opens. Navigate to the directory where you want to save the control file. Then enter the file name and save the file.
4. The file name of the control file is now displayed in the **File name** field, as shown in the figure below.



5 The control file is now linked to your model and will be used when code is generated. Save the model if you want the control file linkage to persist in future MATLAB sessions with your model.

6 You can now edit the control file, for example, adding `ForEach` statements to define block implementation bindings, etc.

Associating an Existing Control File with Your Model

A control file must be associated with your model before you can use the control file in code generation.

If you are generating code via `makehdl` or `makehdltb` from the MATLAB command prompt, use the `HDLControlFiles` property to specify the location of the control file. In the following example, the control file is assumed to be located on the MATLAB path or in the current working directory, and to have the default file name extension `.m`.

```
makehdl('HDLControlFiles', {'dct8config'});
```

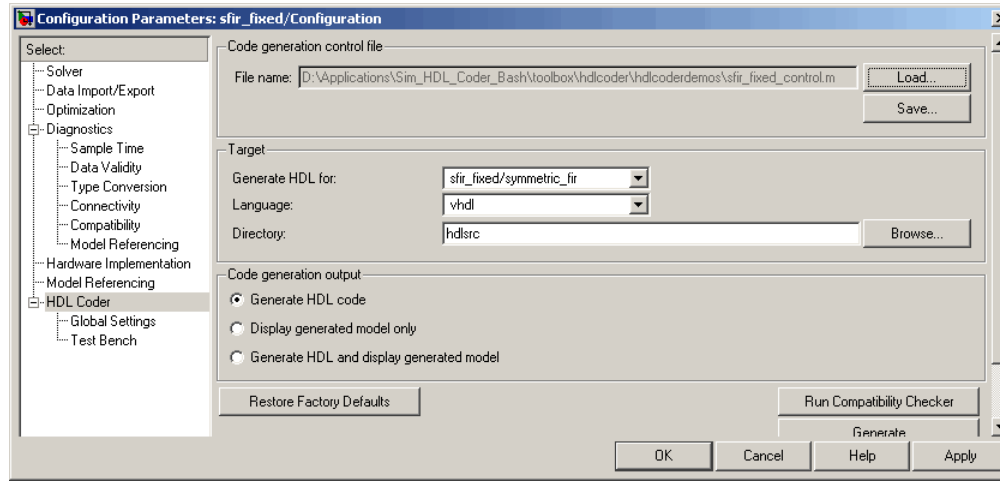
If you are using the GUI to generate code, specify the location of the control file as follows:

1 Open the Configuration Parameters dialog and select the **HDL Coder** options pane.

2 In the **Code generation control file** subpane, click the **Load** button.

3 A standard file dialog opens. Navigate to the desired control file, and select it.

4 The file name of the selected control file is displayed in the **File name** field, as shown in the figure below.



5 Click **Apply**.

6 The control file is now linked to your model and will be used when code is generated. Save the model if you want the control file linkage to persist in future MATLAB sessions with your model.

Detaching a Control File from Your Model

The quickest (and recommended) way to detach a control file from your model is to click the **Restore Factory Defaults** button. This button removes the control file linkage, clears the **File name** field, and resets all Simulink HDL Coder properties to their default settings.

Note **Restore Factory Defaults** resets all HDL code generation settings. This action cannot be cancelled or undone. To recover previous settings, you must close the model without saving it, and then reopen it.

Any of the following actions also detach a control file from a model:

- Attaching another control file, using either the **Load** button or a call to `makehdl`
- Closing the model after attaching a control file, without saving the model
- Clearing the `HDLControlFiles` property by passing in a null file name argument to `makehdl`, as in this example:

```
makehdl(gcb,'HDLControlFiles',{''});
```

Specifying Block Implementations and Parameters in the Control File

Simulink HDL Coder provides a default HDL block implementation for all supported blocks. In addition, Simulink HDL Coder provides selectable alternate HDL block implementations for several block types. Using selection/action statements (`forEach` or `forAll` method calls) in a control file, you can specify the block implementation to be applied to all blocks of a given type (within a specific `modelscope`) during code generation. (See "Code Generation Control Objects and Methods" on page 4-6.)

You select HDL block implementations by specifying an implementation package and class, in the form `package.class`. Pass in the `package.class` specification to the implementation parameter of a `forEach` or `forAll` call, as in the following example.

```
config.forEach('simplevectorsum/vsum/Sum',...
    'built-in/Sum',{},...
    'hdldefaults.SumTreeHDLEmission',{});
```

Given the `package.class` specification, Simulink HDL Coder will call the appropriate code generation method. You do not need to know any internal details of the implementation classes.

Generating Selection/Action Statements with the hdlnewforeach Function

Determining the block path, type, and implementation `package.class` specification for a large number of blocks in a model can be time-consuming. To help you create selection/action statements in your control files, Simulink HDL Coder provides the `hdlnewforeach` function. Given a selection of one or more blocks from your model, `hdlnewforeach` returns the following for each selected block, as string data in the MATLAB workspace:

- A `forEach` call coded with the correct `modelscope` , `blocktype`, and default implementation arguments for the block
- (Optional) A cell array of strings enumerating the available implementations for the block, in `package.class` form

Having generated this information, you can copy and paste the strings into your control file.

hdlnewforeach Example

This example uses `hdlnewforeach` to construct a `forEach` call that specifies a nondefault implementation for a Sum block within the `sfir_fixed` demo model. To do this:

1. In the MATLAB window, select **File > New > M-File**. The MATLAB editor opens an empty M-file.

2. Create a skeletal control file by entering the following code into the M-file window.

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

% Set top-level subsystem from which code is generated.
c.generateHDLFor('sfir_fixed/symmetric_fir');
% INSERT FOREACH CALL BELOW THIS LINE.
```

3. Save the file as `newforeachexamp.m`.

4. Open the `sfir_fixed` demo model.

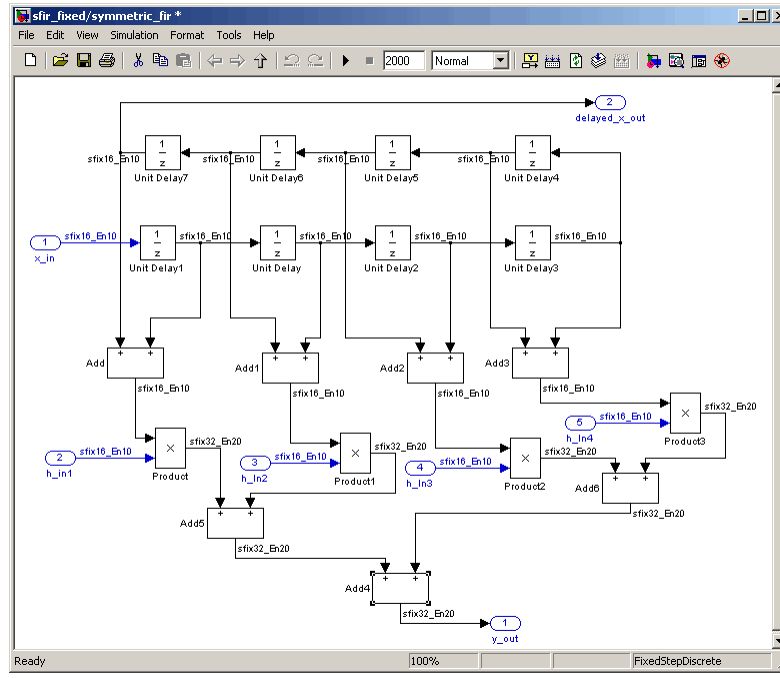
5. Before invoking `hdlnewforeach`, you must run `checkhdl` or `makehdl` to build in-memory information about the model. At the MATLAB command prompt, run `checkhdl` on the `symmetric_fir` subsystem, as shown below.

```
checkhdl('sfir_fixed/symmetric_fir')
### Starting HDL Check.
### HDL Check Complete with 0 errors, warnings and messages.
```

6. Close the `checkdl` report window, and activate the `sfir_fixed` model window.

7. Right-click the`symmetric_fir` subsystem and select **Look Under Mask** from the context menu.

8 In the `symmetric_fir` subsystem window, select the Add4 block, as shown below.



Now you are ready to generate a `forEach` call for the selected block. Do this as follows:

1 Type the following command at the MATLAB prompt.

```
[cmd,impl] = hdlnewforeach(gcb)
```

2 The command returns the results shown below. The first return value, `cmd`, contains the generated `forEach` call.

```
cmd =

c.forEach('sfir_fixed/symmetric_fir/Add4',...
 'built-in/Sum', {},...
 'hdldefaults.SumLinearHDLEmission', {});


impl =

    {3x1 cell}
```

Notice that the `forEach` call specifies the default implementation for the Sum block: `hdldefaults.SumLinearHDLEmission`.

3 The second return value, `impl`, is a cell array containing three strings representing the available implementations for the Sum block. List the contents of the `impl` array as shown below.

```
impl{1}

ans =

    'hdldefaults.SumTreeHDLEmission'
    'hdldefaults.SumLinearHDLEmission'
    'hdldefaults.SumCascadeHDLEmission'
```

See the table “Built-In/Sum of Elements on page 4-23” for information about these implementations.

4 Copy the three lines of `forEach` code from the MATLAB Command window and paste them into the end of your `newforeachexamp.m` file, as shown below.

```
% INSERT FOREACH CALL BELOW THIS LINE.
c.forEach('sfir_fixed/symmetric_fir/Add4',...
 'built-in/Sum', {},...
 'hdldefaults.SumCascadeHDLEmission', {});
```

5 Copy the nondefault implementation string, `'hdldefaults.SumCascadeHDLEmission'` (including quotes) from the MATLAB Command window and paste it into your control file, replacing the default implementation string, `'hdldefaults.SumTreeHDLEmission'`.

6 Save the file.

7 The complete control file is listed below.

```
function c = newforeachexamp
c = hdlnewcontrol(mfilename);

% Set target language for Verilog.
c.set('TargetLanguage','Verilog');

% Set top-level subsystem from which code is generated.
c.generateHDLFor('sfir_fixed/symmetric_fir');
% INSERT FOREACH CALLS HERE.
c.forEach('sfir_fixed/symmetric_fir/Add4',...
 'built-in/Sum', {},...
 'hdldefaults.SumCascadeHDLEmission', {});
```

Note For convenience, `hdlnewforeach` supports a more abbreviated syntax than that used in the above example. See the `hdlnewforeach` reference page.

Blocks with Multiple Implementations

The tables in this section summarize the block types that have multiple implementations. The “Implementations” column gives the `package.class` specification you should use in your control files. The “Description” column summarizes the trade-offs involved in choosing different implementations.

Simulink HDL Coder provides a default HDL block implementation for all supported blocks. If you want to use the default implementation, you do not usually need to specify it explicitly in a control file. However, the following example illustrates a situation in which the default implementation is specified as an exception for one particular block.

```
% 1. Use default (multipliers)  Gain block implementation
% for one specific Gain block within OneD_DCT8 subsystem.
c.forEach('dct8_fixed/OneD_DCT8/Gain14',...
         'built-in/Gain', {},...
         'hdldefaults.GainMultHDLEmission');
% 2. Use factored CSD Gain block implementation
% or all other Gain blocks at or below level of OneD_DCT8 subsystem.
 c.forEach('dct8_fixed/OneD_DCT8/*',...
         'built-in/Gain', {},...
         'hdldefaults.GainFCSDHDLEmission');
```

Built-In/Gain

Implementations	Description
`hdldefaults.GainMultHDLEmission`	*Default.* This implementation retains multiplier operations in HDL code generated by the Gain block.
`hdldefaults.GainCSDHDLEmission`	This implementation decreases the area used by the model while maintaining or increasing clock speed, using canonic signed digit (CSD) techniques. CSD replaces multiplier operations with shift and add operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
`hdldefaults.GainFCSDHDLEmission`	This implementation lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed. This implementation uses factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the operands.

Built-In/Lookup Table

Implementations	Description
`hdldefaults.LookupHDLEmission`	*Default.* Nonhierarchical lookup table.
`hdldefaults.LookupHDLInstantiation`	This implementation generates an additional level of HDL hierarchy (which does not exist in the Simulink model) for the lookup table.

Signal Processing Blockset/Minimum

Implementation	Description
`hdldefaults.MinMaxTreeHDLEmission`	*Default.* This implementation is large and slow but has minimal latency.
`hdldefaults.MinMaxCascadeHDLEmission`	This implementation is optimized for latency * area, with medium speed. See "A Note on Cascade Implementations" on page 4-25.

Signal Processing Blockset/Maximum

Implementation	Description
`hdldefaults.MinMaxTreeHDLEmission`	*Default.* This implementation is large and slow but has minimal latency.
`hdldefaults.MinMaxCascadeHDLEmission`	This implementation is optimized for latency * area, with medium speed. See "A Note on Cascade Implementations" on page 4-25.

Built-In/MinMax

Implementation	Description
`hdldefaults.MinMaxTreeHDLEmission`	*Default.* This implementation is large and slow but has minimal latency.
`hdldefaults.MinMaxCascadeHDLEmission`	This implementation is optimized for latency * area, with medium speed. See "A Note on Cascade Implementations" on page 4-25.

Built-In/Product of Elements

Implementation	Description
`hdldefaults.ProductLinearHDLEmission`	*Default.* Generates a chain of N operations (multipliers) for N inputs.
`hdldefaults.ProductTreeHDLEmission`	This implementation has minimal latency but is large and slow. It generates a tree-shaped structure of multipliers.
`hdldefaults.ProductCascadeHDLEmission`	This implementation optimizes latency * area and is faster than the `tree` implementation. It computes partial products and cascades multipliers. See "A Note on Cascade Implementations" on page 4-25.

Built-In/Sum of Elements

Implementation	Description
`hdldefaults.SumLinearHDLEmission`	*Default.* Generates a chain of N operations (adders) for N inputs.
`hdldefaults.SumTreeHDLEmission`	This implementation has minimal latency but is large and slow. Generates a tree-shaped structure of adders.
`hdldefaults.SumCascadeHDLEmission`	This implementation optimizes latency * area and is faster than the `tree` implementation. It computes partial sums and cascades adders. See "A Note on Cascade Implementations" on page 4-25.

Built-In/SubSystem

Implementation	Description
`hdldefaults.SubsystemBlackBoxHDLInstantiation`	This implementation generates a black box interface for subsystems. That is, the generated HDL code includes only the input/output port definitions for the subsystem. In this way, you can use a subsystem in your model to generate an interface to existing hand-written HDL code. The black box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals.
`hdldefaults.NoHDLEmission`	This implementation completely removes the subsystem from the generated code. This lets you use a subsystem in simulation but treat it as a “no-op” in the HDL code.

For more information on subsystem implementations, see Chapter 7, “Interfacing Subsystems and Models to HDL Code”.

Special-Purpose Implementations

Implementation	Description
`hdldefaults.PassThroughHDLEmission`	Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. (In effect, the block becomes a wire in the HDL code.) Several blocks are supported with a pass-through implementation.
`hdldefaults.NoHDLEmission`	This implementation completely removes the block from the generated code. This lets you use the block in simulation but treat it as a "no-op" in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code. You can also use this implementation as an alternative implementation for subsystems.

For more information related to special-purpose implementations, see Chapter 7, "Interfacing Subsystems and Models to HDL Code".

A Note on Cascade Implementations

Cascade implementations are available for the Sum of Elements, Product of Elements, and Minmax blocks. These implementations require multiple clock cycles to process their inputs; therefore, their inputs must be kept unchanged for their entire sample-time period. Simulink HDL Coder test benches accomplish this by using a register to drive the inputs.

A recommended design practice, when integrating HDL code generated by Simulink HDL Coder with other HDL code, is to provide registers at the inputs. While not strictly required, adding registers to the inputs improves timing and avoids problems with data stability for blocks that require multiple clock cycles to process their inputs.

Summary of Block Implementations

The following table summarizes all blocks that are supported for HDL code generation and their available implementations in the current release. The columns signify

- *Simulink Block*: Library path and block name as displayed in Simulink.
- *Blockscope*: Block path and name to be passed as a `blockscope` string argument to `forEach` or `forAll`.
- *Implementations*: Names of available implementations. When specifying an implementation argument to `forEach` or `forAll`, use the format `package.class`, for example, `hdldefaults.AssignmentHDLEmission` or `hdlstateflow.StateflowHDLInstantiation`.

  Almost all implementation classes currently belong to the package `hdldefaults`. In the table below, the package name is given explicitly only for classes that belong to some other package.

Simulink Block	Blockscope	Implementations
simulink/Model Verification/Assertion	built-in/Assertion	NoHDLEmission
simulink/Math Operations/Assignment	built-in/Assignment	AssignmentHDLEmission
simulink/Math Operations/Abs	built-in/Abs	AbsHDLEmission
simulink/Math Operations/Matrix Concatenate	built-in/Concatenate	MuxHDLEmission
simulink/Math Operations/Vector Concatenate	built-in/Concatenate	MuxHDLEmission
simulink/Commonly Used Blocks/Constant	built-in/Constant	ConstantHDLEmission

Simulink Block	Blockscope	Implementations
simulink/Commonly Used Blocks/Data Type Conversion	built-in/ DataTypeConversion	DataTypeConversionHDLEmission
simulink/Commonly Used Blocks/Demux	built-in/Demux	DemuxHDLEmission
simulink/Sinks/Display	built-in/Display	NoHDLEmission
dspsigattribs/Frame Conversion	built-in/FrameConversion	FrameConversionHDLEmission
simulink/Commonly Used Blocks/Gain	built-in/Gain	GainMultHDLEmission GainFCSDHDLEmission GainCSDHDLEmission
simulink/Commonly Used Blocks/Ground	built-in/Ground	ConstantHDLEmission
simulink/Commonly Used Blocks/In1	built-in/Inport	NoHDLEmission (Input ports are generated automatically.)
simulink/Commonly Used Blocks/Logical Operator	built-in/Logic	LogicHDLEmission
simulink/Lookup Tables/Lookup Table	built-in/Lookup	LookupHDLInstantiation LookupHDLEmission
simulink/Discrete/Memory	built-in/Memory	MemoryHDLEmission
simulink/Math Operations/MinMax	built-in/MinMax	MinMaxTreeHDLEmission MinMaxCascadeHDLEmission
simulink/Ports & Subsystems/Model	built-in/ModelReference	ModelReferenceHDLInstantiation
simulink/Signal Routing/Index Vector	built-in/MultiPortSwitch	MultiPortSwitchHDLEmission
simulink/Signal Routing/Multiport Switch	built-in/MultiPortSwitch	MultiPortSwitchHDLEmission

Simulink Block	Blockscope	Implementations
simulink/Commonly Used Blocks/Mux	built-in/Mux	MuxHDLEmission
simulink/Commonly Used Blocks/Out1	built-in/Outport	NoHDLEmission (Output ports are generated automatically.)
simulink/Commonly Used Blocks/Product	built-in/Product	ProductLinearHDLEmission ProductTreeHDLEmission ProductCascadeHDLEmission (ProductTreeHDLEmission and ProductCascadeHDLEmission are supported for Product blocks having two or more inputs.)
simulink/Math Operations/Product of Elements	built-in/Product	ProductTreeHDLEmission ProductLinearHDLEmission ProductCascadeHDLEmission
simulink/Signal Attributes/Rate Transition	built-in/RateTransition	RateTransitionHDLEmission
simulink/Commonly Used Blocks/Relational Operator	built-in/ RelationalOperator	RelationalOperatorHDLEmission
simulink/Commonly Used Blocks/Scope	built-in/Scope	NoHDLEmission
simulink/Sinks/Floating Scope	built-in/Scope	NoHDLEmission
dspsnks4/Time Scope	built-in/Scope	NoHDLEmission
simulink/Signal Routing/Selector	built-in/Selector	SelectorHDLEmission
simulink/Signal Attributes/Signal Conversion	built-in/SignalConversion	PassThroughHDLEmission

Simulink Block	Blockscope	Implementations
simulink/Signal Attributes/Signal Specification	built-in/ SignalSpecification	SignalSpecificationHDLEmission
simulink/Sinks/Stop Simulation	built-in/Stop	NoHDLEmission
simulink/Commonly Used Blocks/Sum	built-in/Sum	SumLinearHDLEmission SumTreeHDLEmission SumCascadeHDLEmission (SumTreeHDLEmission and SumCascadeHDLEmission are supported for Sum blocks having two or more inputs.)
simulink/Math Operations/Add	built-in/Sum	SumTreeHDLEmission SumLinearHDLEmission SumCascadeHDLEmission (SumTreeHDLEmission and SumCascadeHDLEmission are supported for Add blocks having two or more inputs.)
simulink/Math Operations/Subtract	built-in/Sum	SumTreeHDLEmission SumLinearHDLEmission SumCascadeHDLEmission (SumTreeHDLEmission and SumCascadeHDLEmission are supported for Subtract blocks having two or more inputs.)

Simulink Block	Blockscope	Implementations
simulink/Math Operations/Sum of Elements	built-in/Sum	SumTreeHDLEmission SumLinearHDLEmission SumCascadeHDLEmission (SumTreeHDLEmission and SumCascadeHDLEmission are supported for Sum of Elements blocks having two or more inputs.)
simulink/Commonly Used Blocks/Switch	built-in/Switch	SwitchHDLEmission
simulink/Commonly Used Blocks/Terminator	built-in/Terminator	NoHDLEmission
simulink/Sinks/To File	built-in/ToFile	NoHDLEmission
simulink/Sinks/To Workspace	built-in/ToWorkspace	NoHDLEmission
simulink/Commonly Used Blocks/Unit Delay	built-in/UnitDelay	UnitDelayHDLEmission
simulink/Discrete/ Zero-Order Hold	built-in/ZeroOrderHold	ZeroOrderHoldHDLEmission
simulink/Discrete/Integer Delay	simulink/Discrete/Integer Delay	IntegerDelayHDLEmission
simulink/Discrete/Tapped Delay	simulink/Discrete/Tapped Delay	TappedDelayHDLEmission
simulink/Logic and Bit Operations/Bit Clear	simulink/Logic and Bit Operations/Bit Clear	BitOpsHDLEmission
simulink/Logic and Bit Operations/Bit Set	simulink/Logic and Bit Operations/Bit Set	BitOpsHDLEmission
simulink/Logic and Bit Operations/Bitwise Operator	simulink/Logic and Bit Operations/Bitwise Operator	BitOpsHDLEmission

Simulink Block	Blockscope	Implementations
simulink/Logic and Bit Operations/Compare To Constant	simulink/Logic and Bit Operations/Compare To Constant	CompareToConstHDLEmission
simulink/Logic and Bit Operations/Compare To Zero	simulink/Logic and Bit Operations/Compare To Zero	CompareToZeroHDLEmission
simulink/Logic and Bit Operations/Shift Arithmetic	simulink/Logic and Bit Operations/Shift Arithmetic	BitOpsHDLEmission
simulink/Math Operations/Reshape	simulink/Math Operations/Reshape	PassThroughHDLEmission
simulink/Math Operations/Unary Minus	simulink/Math Operations/Unary Minus	UnaryMinusHDLEmission
simulink/Model Verification/Check Dynamic Gap	simulink/Model Verification/Check Dynamic Gap	NoHDLEmission
simulink/Model Verification/Check Dynamic Range	simulink/Model Verification/Check Dynamic Range	NoHDLEmission
simulink/Model Verification/Check Static Gap	simulink/Model Verification/Check Static Gap	NoHDLEmission
simulink/Model Verification/Check Static Range	simulink/Model Verification/Check Static Range	NoHDLEmission
simulink/Model Verification/Check Discrete Gradient	simulink/Model Verification/Check Discrete Gradient	NoHDLEmission
simulink/Model Verification/Check Dynamic Lower Bound	simulink/Model Verification/Check Dynamic Lower Bound	NoHDLEmission

Simulink Block	Blockscope	Implementations
simulink/Model Verification/Check Dynamic Upper Bound	simulink/Model Verification/Check Dynamic Upper Bound	NoHDLEmission
simulink/Model Verification/Check Input Resolution	simulink/Model Verification/Check Input Resolution	NoHDLEmission
simulink/Model Verification/Check Static Lower Bound	simulink/Model Verification/Check Static Lower Bound	NoHDLEmission
simulink/Model Verification/Check Static Upper Bound	simulink/Model Verification/Check Static Upper Bound	NoHDLEmission
simulink/Signal Attributes/Data Type Duplicate	simulink/Signal Attributes/Data Type Duplicate	NoHDLEmission
simulink/Signal Attributes/Data Type Propagation	simulink/Signal Attributes/Data Type Propagation	NoHDLEmission
simulink/Sinks/XY Graph	simulink/Sinks/XY Graph	NoHDLEmission
simulink/Sources/Counter Free-Running	simulink/Sources/Counter Free-Running	CounterFreeRunningHDLEmission
simulink/Sources/Counter Limited	simulink/Sources/Counter Limited	CounterLimitedHDLEmission
dsparch4/Digital Filter	dsparch4/Digital Filter	DigitalFilterHDLInstantiation
dspindex/Multiport Selector	dspindex/Multiport Selector	MultiportSelectorHDLEmission
dspindex/Variable Selector	dspindex/Variable Selector	VariableSelectorHDLEmission
dspsigattribs/Convert 1-D to 2-D	dspsigattribs/Convert 1-D to 2-D	PassThroughHDLEmission
dspsigops/Delay	dspsigops/Delay	DSPDelayHDLEmission

Simulink Block	Blockscope	Implementations
dspsnks4/Matrix Viewer	dspsnks4/Matrix Viewer	NoHDLEmission
dspsnks4/Signal To Workspace	dspsnks4/Signal To Workspace	NoHDLEmission
dspsnks4/Spectrum Scope	dspsnks4/Spectrum Scope	NoHDLEmission
dspsnks4/Vector Scope	dspsnks4/Vector Scope	NoHDLEmission
dspsnks4/Waterfall	dspsnks4/Waterfall	NoHDLEmission
dspsrcs4/DSP Constant	dspsrcs4/DSP Constant	ConstantHDLEmission
dspstat3/Maximum	dspstat3/Maximum	MinMaxTreeHDLEmission MinMaxCascadeHDLEmission
dspstat3/Minimum	dspstat3/Minimum	MinMaxTreeHDLEmission MinMaxCascadeHDLEmission
modelsimlib/HDL Cosimulation	modelsimlib/HDL Cosimulation	ModelSimHDLInstantiation
modelsimlib/To VCD File	modelsimlib/To VCD File	NoHDLEmission
sflib/Chart	sflib/Chart	hdlstateflow.StateflowHDLInstantiation

5

Generating Bit-True Cycle-Accurate Models

Overview of Generated Models

In some circumstances, significant differences in behavior can arise between a Simulink model and the HDL code generated from that model. Such differences fall into two categories:

- *Numerics*: differences in intermediate and/or final computations. For example, a selected block implementation may restructure arithmetic operations to optimize for speed (see "Example: Numeric Differences" on page 5-4). Where such numeric differences exist, the HDL code is no longer *bit-true* to the Simulink model.
- *Latency*: insertion of delays of one or more clock cycles at certain points in the HDL code. Some block implementations that optimize for area can introduce these delays. Where such latency exists, the timing of the HDL code is no longer *cycle-accurate* with respect to the Simulink model.

To help you evaluate such cases, Simulink HDL Coder creates a *generated model* that is bit-true and cycle-accurate with respect to the generated HDL code. The generated model lets you

- Run Simulink simulations that accurately reflect the behavior of the generated HDL code.
- Create test benches based on the generated model, rather than the original model.
- Visually detect (by color highlighting of affected subsystems) all differences between the original and generated models.

Simulink HDL Coder always creates a generated model as part of the code generation process, and always generates test benches based on the generated model, rather than the original model. In cases where no latency or numeric differences occur, you can disregard the generated model except when generating test benches.

Simulink HDL Coder also provides options that let you

- Suppress display of the generated model.
- Create and display the only generated model, with code generation suppressed.

- Specify the color highlighting of differences between the original and generated models.
- Specify a name or prefix for the generated model.

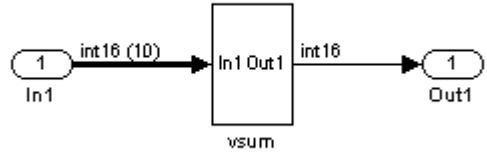
These options are described in “Defaults and Options for Generated Models” on page 5-8.

Example: Numeric Differences

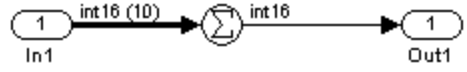
This example first examines a simple model that uses a code generation control file to select a speed-optimized Sum block implementation. It then examines a generated model and locates the numeric changes introduced by the optimization.

If you are not familiar with code generation control files and selection of block implementations, see Chapter 4, “Code Generation Control Files”.

The model, `simplevectorsum`, consists of a subsystem, `vsum`, driven by a vector input of width 10, with a scalar output. The figure below shows the root level of the model.



The device under test is the `vsum` subsystem, shown in the figure below. The subsystem contains a Sum block, configured for vector summation.

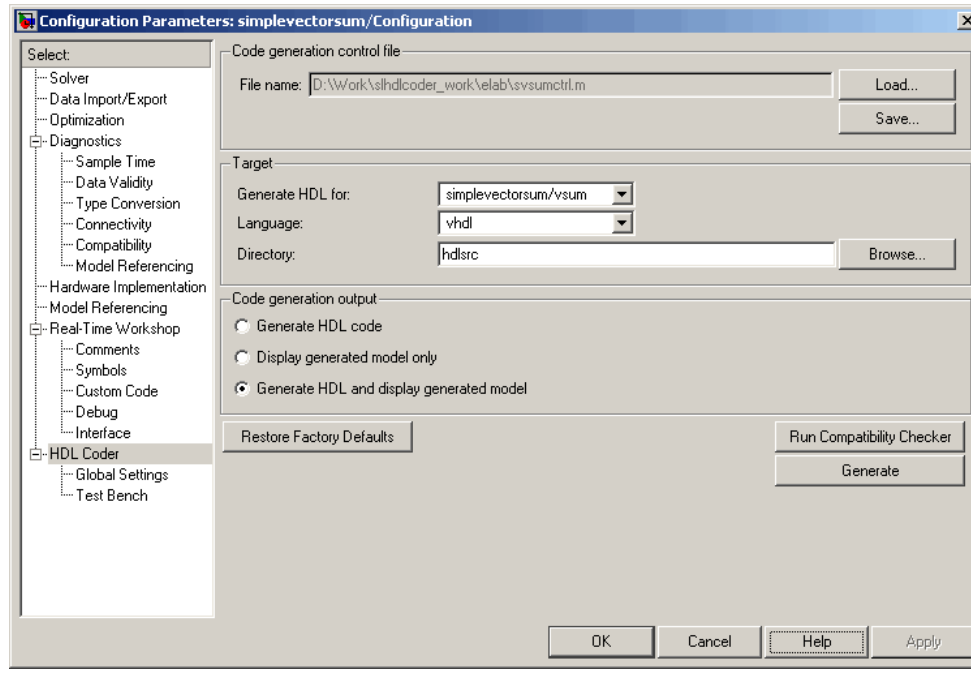


The model is configured to use a code generation control file, `svsumctrl.m`. The control file (listed below) maps the `SumTreeHDLEmission` implementation to the Sum block within the `vsum` subsystem. This implementation, optimized for minimal latency, generates a tree-shaped structure of adders for the Sum block.

```
function config = svsumctrl
% Code generation control file for simplevectorsum model.
```

```
config = hdlnewcontrol(mfilename);
% Specify tree-structured adders implementaton for Sum block.
config.forEach('simplevectorsum/vsum/Sum',...
    'built-in/Sum',{},...
    'hdldefaults.SumTreeHDLEmission',{});
```

The **File name** field of the Configuration Parameters dialog (shown below) specifies that this control file is to be used during code generation.

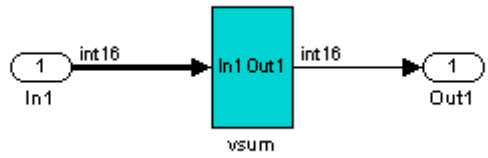


When code generation is initiated, Simulink HDL Coder displays messages similar to those below, indicating that the control file is applied, followed by creation of the generated model and generation of HDL code.

```
### Applying HDL Code Generation Control Statements
###   1 Control Statements to be applied
### Begin Elaborated Model Generation
### Generating new model: gm_simplevectorsum.mdl
### Elaborated Model Generation Complete.
```

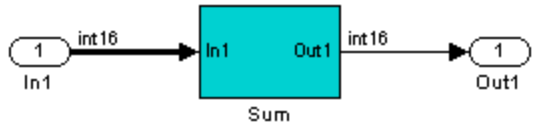
```
### Begin Vhdl Code Generation
### Working on simplevectorsum/vsum as hdlsrc/vsum.vhd
### HDL Code Generation Complete.
```

The generated model, `gm_ simplevectorsum`, is displayed after code generation. This model is shown in the figure below.

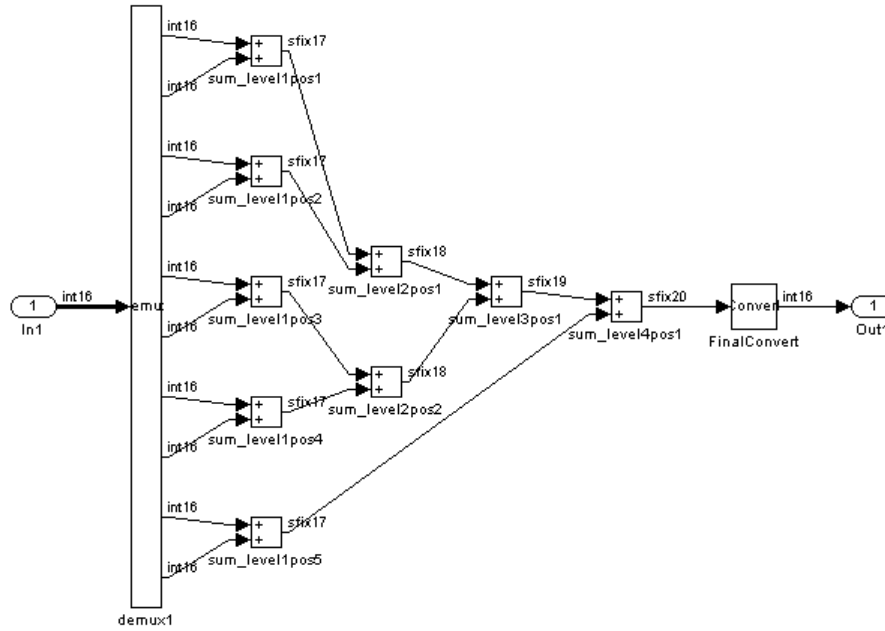


At the root level, this model appears identical to the original model, except that the `vsum` subsystem has been highlighted in cyan. This highlighting indicates that the subsystem differs in some respect from the `vsum` subsystem of the original model.

The figure below shows the `vsum` subsystem in the generated model. Observe that the Sum block is now implemented as a subsystem, which is also highlighted.



The figure below shows the internal structure of the Sum subsystem.



The vector sum is implemented as a tree of adders (Sum blocks). The vector input signal is demultiplexed and connected, as five pairs of operands, to the five leftmost adders. The widths of the adder outputs increase from left to right, as required to avoid overflow in computing intermediate results. A Data Conversion block, inserted before the final output, converts the 20-bit fixed-point result to the `int16` data type required by the model.

Defaults and Options for Generated Models

This section summarizes

- The defaults used by Simulink HDL Coder when generated models are built (see "Defaults for Model Generation" on page 5-8).
- GUI options and `makehdl` properties that provide control over the generation, naming, and appearance of generated models (see "GUI Options" on page 5-9 and "Generated Model Properties for makehdl" on page 5-10).

Defaults for Model Generation

Model Generation

Simulink HDL Coder always creates a generated model as part of the code generation process. The generated model is built in memory, before actual generation of HDL code. The HDL code and the generated model are bit-true and cycle-accurate with respect to one another.

Note The in-memory generated model is not written to a model file unless you explicitly save it.

Naming of Generated Models

The naming convention for generated models is

```
prefix_modelname
```

where the default prefix is `gm_`, and the default `modelname` is the name of the original model.

If code is generated more than once from the same original model, and previously generated model(s) exist in memory, an integer is suffixed to the name of each successively generated model. The suffix ensures that each generated model has a unique name. For example, if the original model is named `test`, generated models will be named `gm_test`, `gm_test0`, `gm_test1`, etc.

Note Take care, when regenerating code from your models, to select the original model for code generation, not a previously generated model. Generating code from a generated model may introduce unintended delays or numeric differences that could make the model operate incorrectly.

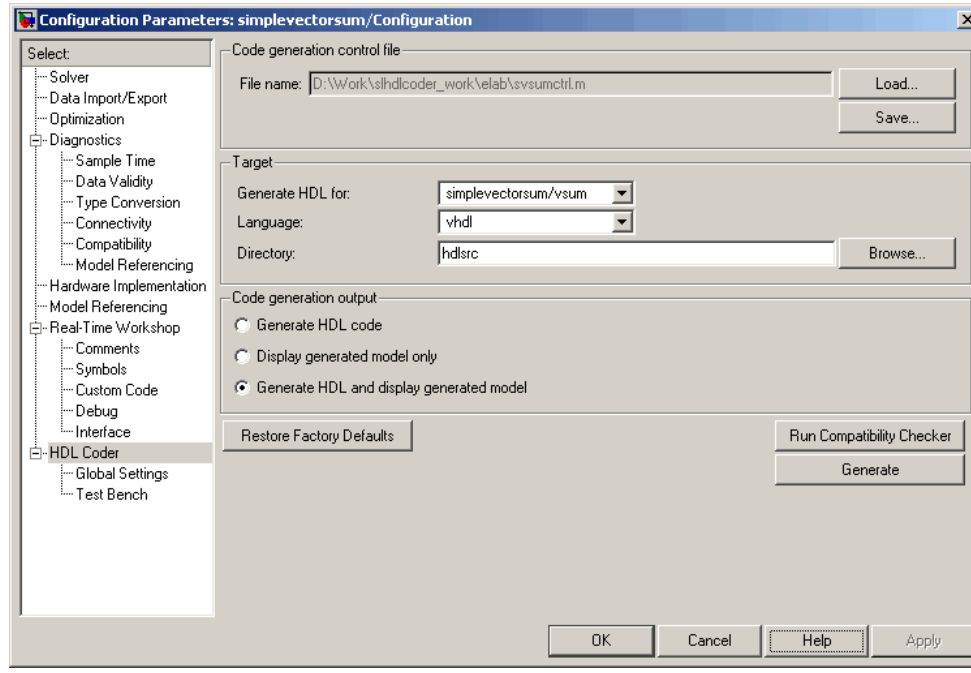
Block Highlighting

By default, blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy, are highlighted in the default color, cyan. You can quickly see whether any differences have been introduced, by examining the root level of the generated model.

If there are no differences between the original and generated models, no blocks will be highlighted.

GUI Options

The Simulink HDL Coder GUI provides high-level options controlling the generation and display of generated models. More detailed control is available through the `makehdl` command (see “Generated Model Properties for makehdl” on page 5-10). Generated model options are located in the top-level **HDL Options** pane of the Configuration Parameters dialog, as shown in the figure below.



The options are

- **Generate HDL code**: (Default) Generate code, but do not display the generated model.
- **Display generated model only**: Create and display the generated model, but do not proceed to code generation.
- **Generate HDL and display generated model**: Generate both code and model, and display the model when completed.

Generated Model Properties for makehdl

The following table summarizes `makehdl` properties that provide detailed controls for the generated model.

Property and Value(s)	Description
`'Generatedmodelnameprefix', ['string']`	The default name for the generated model is `gm_modelname`, where `gm_` is the default prefix and `modelname` is the original model name. To override the default prefix, assign a string value to this property.
`'Generatemodelname', ['string']`	By default, the original model name is used as the `modelname` substring of the generated model name. To specify a different model name, assign a string value to this property.
`'CodeGenerationOutput', 'string'`	Controls the production of generated code and display of the generated model. Values are • `GenerateHDLCode`: (Default) Generate code, but do not display the generated model. • `GenerateHDLCodeAndDisplayGeneratedModel`: Create and display generated model, but do not proceed to code generation. • `DisplayGeneratedModelOnly`: Generate both code and model, and display model when completed.
`'Highlightancestors', ['on' | 'off']`	By default, blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy, are highlighted in a color specified by the `Highlightcolor` property. If you do not want the ancestor blocks to be highlighted, set this property to `'off'`.
`'Highlightcolor', 'RGBName'`	Specify the color used to highlight blocks in a generated model that differ from the original model (default: cyan). Specify the color (`RGBName`) as one of the following color string values: • `cyan` (default) • `yellow` • `magenta` • `red`

Property and Value(s)	Description
	• `green` • `blue` • `white` • `black`

6

HDL Compatibility, Code Tracing, and Block Support Reports

HDL Compatibility Checker

The HDL compatibility checker lets you check whether a subsystem or model is compatible with HDL code generation. You can run the compatibility checker from the MATLAB command line or an M-file script, or from the Simulink GUI.

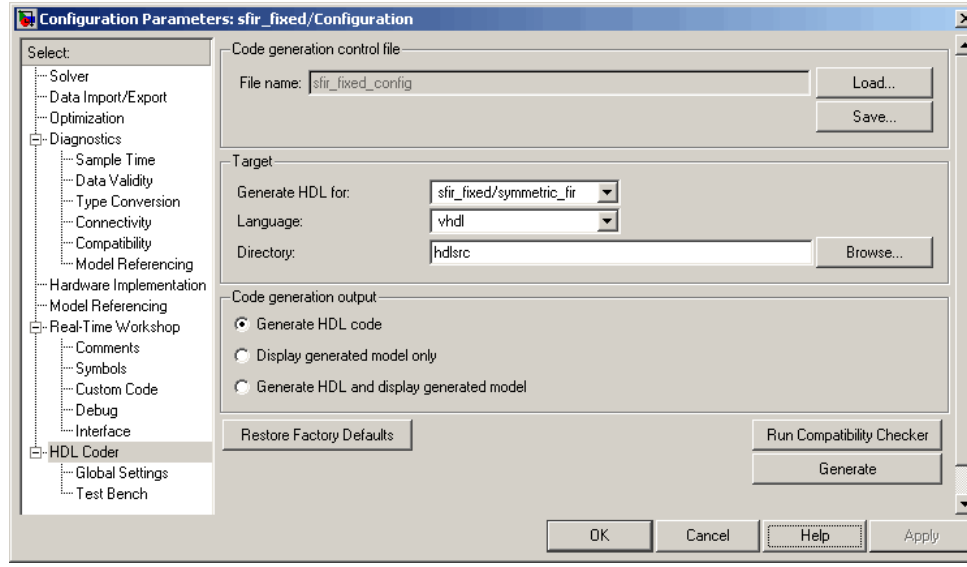
To run the compatibility checker from the command line or an M-file script, use the `checkhdl` function. The syntax of the function is

```
checkhdl('system')
```

where *`system`* is the device under test (DUT), typically a subsystem within the current Simulink model.

To run the compatibility checker from the Simulink GUI:

1 Open the Configuration Parameters dialog or the Model Explorer. Select the **HDL Coder** options category. The figure below shows the **HDL Coder** pane of the Configuration Parameters dialog.



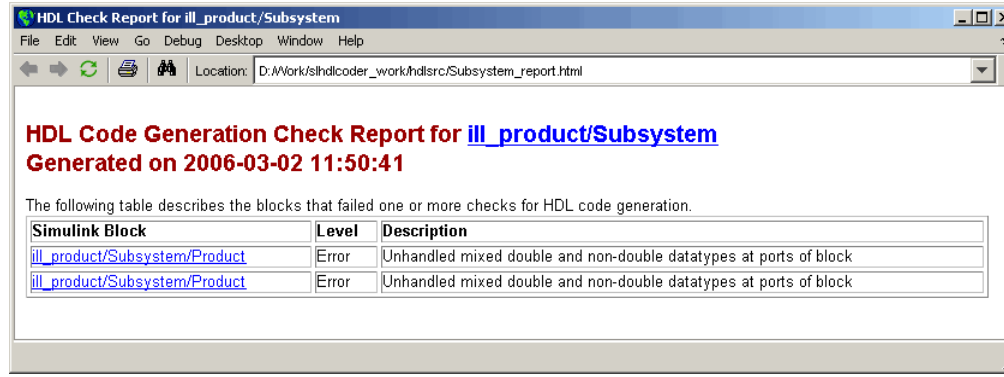
2 Select the subsystem you want to check from the **Generate HDL for** pop-up menu.

3 Click the **Run Compatibility Checker** button.

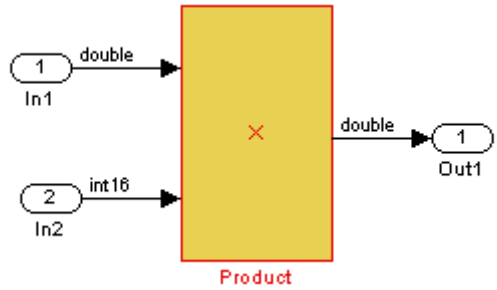
The HDL compatibility checker examines the specified system for any compatibility problems, such as use of unsupported blocks, illegal data type usage, etc. The HDL compatibility checker generates an HDL Code Generation Check Report, which is stored in the target directory. The report file naming convention is *system*`_report.html`, where *system* is the name of the subsystem or model that was passed in to the HDL compatibility checker.

The HDL Code Generation Check Report is displayed in a browser window. Each entry in the HDL Code Generation Check Report is hyperlinked to the block or subsystem that caused the problem. When you click the hyperlink, Simulink highlights and displays the block of interest (provided that the model referenced by the report is open).

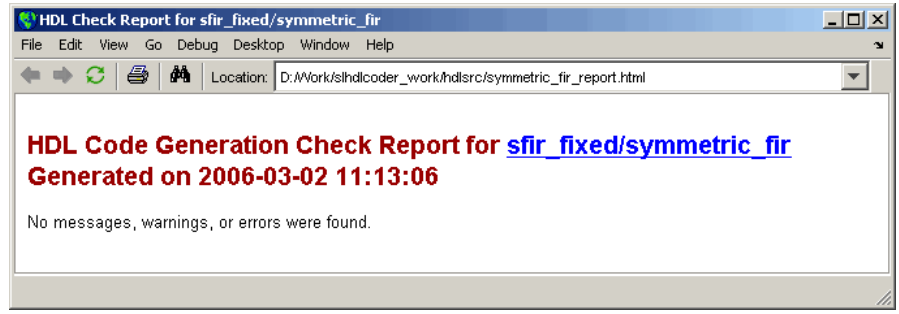
The figure below shows an HDL Code Generation Check Report that was generated for a subsystem with a Product block that was configured with a mixture of double and integer port data types. This configuration is legal in Simulink, but incompatible with Simulink HDL Coder.



When you click the hyperlink in the left column, Simulink opens the subsystem containing the offending block. The block of interest is highlighted, as shown in the following figure.



The figure below shows an HDL Code Generation Check Report that was generated for a subsystem that passed all compatibility checks. In this case, the report contains only a hyperlink to the subsystem that was checked.



Code Tracing Using the Mapping File

Note This section refers to generated VHDL entities or Verilog modules generically as “entities.”

A *mapping file* is a text report file generated by `makehdl`. Mapping files are generated as an aid in tracing generated HDL entities back to the corresponding Simulink systems.

A mapping file shows the relationship between systems in the Simulink model and the VHDL entities or Verilog modules that were generated from them. A mapping file entry has the form

```
Simulink_path --> HDL_name
```

where *`Simulink_path`* is the full Simulink path to a system in the Simulink model and *`HDL_name`* is the name of the VHDL entity or Verilog module that was generated from that system. The mapping file contains one entry per line.

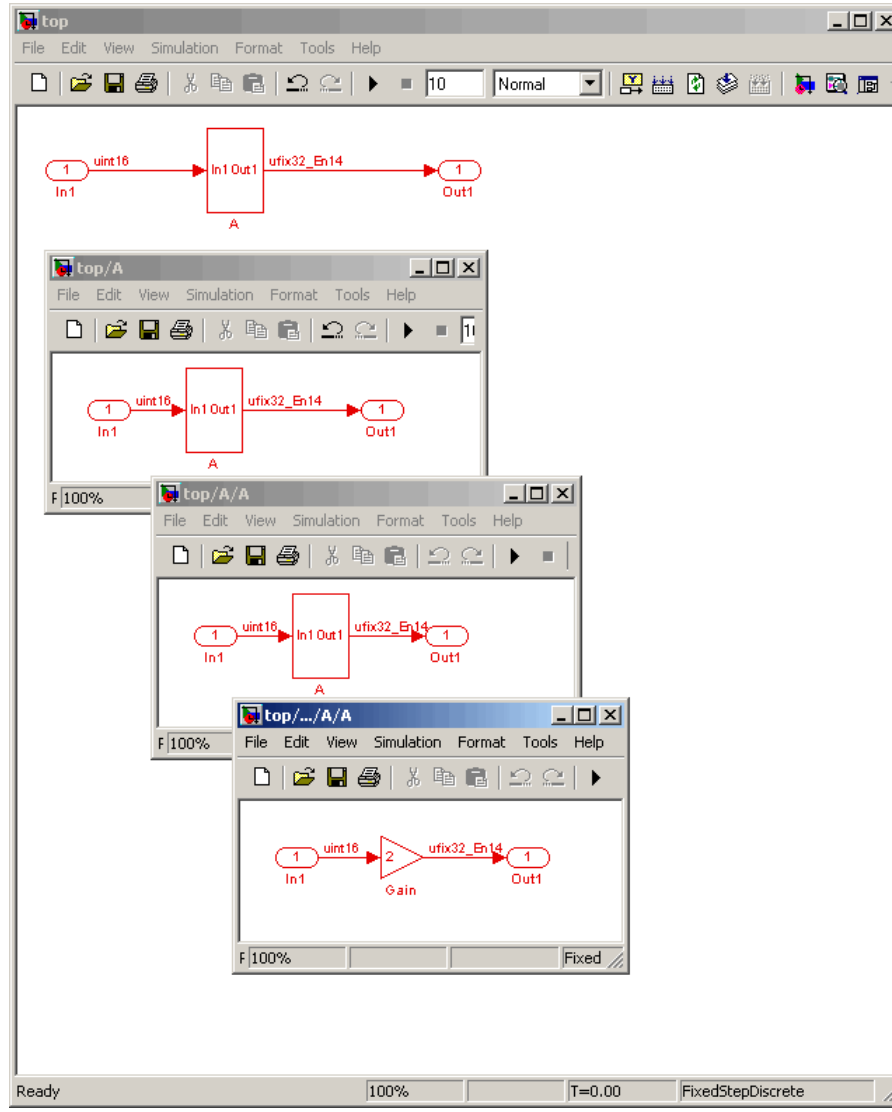
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` demo model generates the following mapping file:

```
sfir_fixed/symmetric_fir --> symmetric_fir
```

Mapping files are more useful when HDL code is generated from complex models where multiple subsystems generate many entities, and in cases where conflicts between identically named subsystems are resolved by Simulink HDL Coder.

If a subsystem name is unique within the model, Simulink HDL Coder simply uses the subsystem name as the generated entity name. Where identically named subsystems are encountered, Simulink HDL Coder attempts to resolve the conflict by appending a postfix string (by default, `'_entity'`) to the conflicting subsystem. If subsequently generated entity names conflict in turn with this name, incremental numerals (`1,2,3,...`*`n`*) are appended.

As an example, consider the model shown below. The top-level model contains subsystems named A nested to three levels.



When code is generated for the top-level subsystem `A`, `makehdl` works its way up from the deepest level of the model hierarchy, generating unique entity names for each subsystem.

```
makehdl('top/A')
### Working on top/A/A/A as A_entity1.vhd
### Working on top/A/A as A_entity2.vhd
### Working on top/A as A.vhd

### HDL Code Generation Complete.
```

The contents of the resultant mapping file are shown below.

```
top/A/A/A --> A_entity1
top/A/A --> A_entity2
top/A --> A
```

Given this information, you could trace any generated entity back to its corresponding subsystem by using the `open_system` command, for example:

```
open_system('top/A/A')
```

Note that each generated entity file also contains the Simulink path for its corresponding subsystem in the header comments at the top of the file, as in the excerpt shown below.

```
-- Module: A_entity2
-- Simulink Path: top/A
-- Created: 2005-04-20 10:23:46
-- Hierarchy Level: 0
```

Supported Blocks Library

Simulink HDL Coder provides an M-file utility, `hdllib.m`, that creates a library of all blocks that are currently supported for HDL code generation.

The block library, `hdlsupported.mdl`, affords quick access to all supported blocks. By constructing models using blocks from this library, you can ensure compatibility with Simulink HDL Coder.

The set of supported blocks will change in future releases of Simulink HDL Coder. To keep the `hdlsupported.mdl` current, The MathWorks recommends that you rebuild the library each time you install a new release. To create the library:

1. Type the following at the MATLAB prompt:

   ```
   hdllib
   ```

   `hdllib` starts generation of the `hdlsupported` library. Simulink loads many libraries during the creation of the `hdlsupported` library. When `hdllib` completes generation of the library, it does not unload these libraries.

2. After the library is generated, you must save it to a directory of your choice. You should retain the file name `hdlsupported.mdl`, because this document refers to the supported blocks library by that name.

7

Interfacing Subsystems and Models to HDL Code

Overview of HDL Interfaces

Simulink HDL Coder provides a number of different ways to generate interfaces to your hand-written or legacy HDL code. Depending on your application, you may want to generate such an interface from different levels of your model:

- Subsystem
- Model referenced by a higher-level model
- Cosimulation block

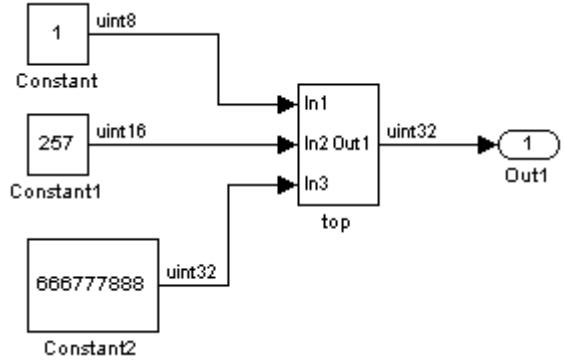
You can also generate a pass-through (wire) HDL implementation for a subsystem, or omit code generation entirely for a subsystem. Both of these techniques can be useful in cases where you need a subsystem in your simulation, but do not need the subsystem in your generated HDL code.

Generating a Black Box Interface for a Subsystem

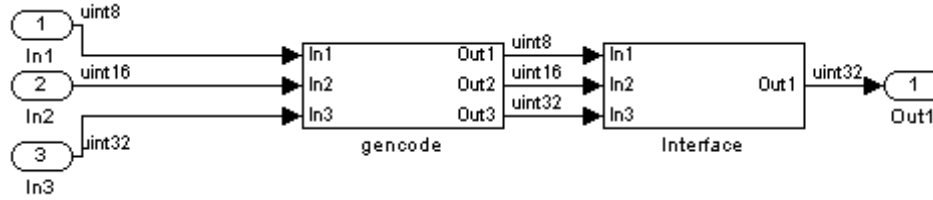
A *black box* interface for a subsystem is a generated VHDL component or Verilog module that includes only the HDL input/output port definitions for the subsystem. By generating such a component, you can use a subsystem in your model to generate an interface to existing hand-written HDL code.

To generate the interface, you use a control file to map one or more Subsystem blocks to the `hdldefaults.SubsystemBlackBoxHDLInstantiation` implementation. (See Chapter 4, "Code Generation Control Files" for a detailed description of the structure and use of control files.)

As an example, consider the model and subsystem shown in the figures below. The model, `subsystst`, contains a subsystem, `top`, which is the device under test.



The subsystem `top` contains two lower-level subsystems, `gencode` and `Interface`.



Suppose that you want to generate HDL code from `top`, with a black box interface from the `Interface` subsystem. The first step would be to create a control file that defines the Simulink path and block type for the `Interface` subsystem, and maps this subsystem to the `hdldefaults.SubsystemBlackBoxHDLInstantiation` implementation. An example control file is listed below.

```
% Code generation control file - blackbox_ctrl.m
function control = blackbox_ctrl
control = hdlnewcontrol(mfilename);
% Generate a black box interface for the subsystem labeled
% Interface within the top-level device

control.forEach( ...
  'subsystst/top/Interface', ...
  'built-in/SubSystem', {}, ...
  'hdldefaults.SubsystemBlackBoxHDLInstantiation');
```

The control file is attached to the model when code generation is invoked. In the `makehdl` command line below, VHDL code is generated by default.

```
makehdl('subsystst/top','HDLControlFiles',{'blackbox_ctrl.m'})
### Applying User Configuration File: blackbox_ctrl.m

### Begin Vhdl Code Generation
### Working on subsystst/top/gencode as hdlsrc/gencode.vhd
### Working on subsystst/top as hdlsrc/top.vhd
### HDL Code Generation Complete.
```

In the `makehdl` progress messages above, observe that the `gencode` subsystem generates a separate code file (`gencode.vhd`) for its VHDL entity definition. The `Interface` subsystem does not generate such a file. The interface code for this subsystem is in `top.vhd`, generated from `subsystst/top`. The component definition and instantiation generated for the `Interface` subsystem are excerpted in the code listing below.

```
COMPONENT Interface
    PORT( In1  :   IN    std_logic_vector(7 DOWNTO 0);  -- ufix8
          In2  :   IN    std_logic_vector(15 DOWNTO 0);  -- ufix16
          In3  :   IN    std_logic_vector(31 DOWNTO 0);  -- ufix32
```

```
          Out1 :   OUT   std_logic_vector(31 DOWNTO 0)  -- ufix32
          );
  END COMPONENT;
...
u_Interface : Interface
    PORT MAP
      (In1 => gencode_out1,  -- ufix8
       In2 => gencode_out2,  -- ufix16
       In3 => gencode_out3,  -- ufix32
       Out1 => Interface_out1  -- ufix32
       );
  ce_out <= enb;
```

The black box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals. (See also “Generating Interfaces for Referenced Models ” on page 7-6.)

Generating Interfaces for Referenced Models

The Simulink model referencing feature allows you to include models in other models as blocks. Included models are referenced through Model blocks (see "Referencing Models" in the Simulink documentation for detailed information).

For Model blocks, Simulink HDL Coder generates a VHDL component or a Verilog module instantiation. However, `makehdl` does not attempt to generate HDL code for the models referenced from Model blocks. You must generate HDL code for each referenced model individually. To generate code for a referenced model:

1. Select the referencing Model block.
2. Double-click the Model block to open its mask dialog.
3. Click the Open Model button to open the referenced model.
4. Invoke the `checkhdl` and `makehdl` functions to check and generate code from that model.

Note The `checkhdl` function does not check port data types within the referenced model.

The Model block is useful for multiply-instantiated blocks, or for blocks for which you already have hand-written HDL code. The generated HDL will contain all the code that is required to interface to the referenced HDL code. Code is generated with the following assumptions:

- Every HDL entity or module requires clock, clock enable, and reset ports. Therefore, these ports are defined for each generated entity or module.
- Use of Simulink data types is assumed. For VHDL code, port data types are assumed to be `STD_LOGIC` or `STD_LOGIC_VECTOR`.

Code Generation for the HDL Cosimulation Block

This feature requires the following products:

- Link for ModelSim 1.4
- Mentor Graphics ModelSim SE/PE HDL simulator

The Link for ModelSim HDL Cosimulation block cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model under simulation in ModelSim. For detailed information on the HDL Cosimulation block, see the Link for ModelSim documentation.

You can use the HDL Cosimulation block with Simulink HDL Coder to generate an interface to your hand-written or legacy HDL code. When an HDL Cosimulation block is included in a model, Simulink HDL Coder generates a VHDL or Verilog interface, depending on the selected target language.

When the target language is VHDL, the generated interface includes

- An entity definition. The entity defines ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. Clock enable and reset ports are also declared.
- An RTL architecture including a component declaration, a component configuration declaring signals corresponding to signals connected to the HDL Cosimulation ports, and a component instantiation.
- Port assignment statements as required by the model.

When the target language is Verilog, the generated interface includes

- A module defining ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. The module also defines clock enable and reset ports, and `wire` declarations corresponding to signals connected to the HDL Cosimulation ports.
- A module instance.
- Port assignment statements as required by the model.

The requirements for using the HDL Cosimulation block for code generation are the same as those for cosimulation. That is, a ModelSim session must be running and properly connected to the HDL Cosimulation block. If you want to check these conditions before initiating code generation, select **Update Diagram** from the Simulink **Edit** menu.

Pass-Through and No-Op Implementations

Simulink HDL Coder provides special-purpose implementations for subsystems that let you use a subsystem as a wire, or simply omit a subsystem entirely, in the generated HDL code. These implementations are summarized in the table below.

Implementation	Description
`hdldefaults.PassThroughHDLEmission`	Provides a pass-through implementation in which the subsystem's inputs are passed directly to its outputs. (In effect, the block becomes a wire in the HDL code.)
`hdldefaults.NoHDLEmission`	Completely removes the block from the generated code. Lets you use the block in simulation but treat it as a no-op in the HDL code.

Simulink HDL Coder uses these implementations for many built-in blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code.

8

Stateflow HDL Code Generation Support

Overview of Stateflow HDL Code Generation

Stateflow is a powerful graphical design and development tool for solving complex control and supervisory logic problems. Stateflow provides concise descriptions of complex system behavior using hierarchical finite state machine (FSM) theory, flow diagram notation, and state-transition diagrams.

You use a Stateflow diagram to model a finite state machine or a complex control algorithm intended for realization as an ASIC or FPGA. When the model meets design requirements, you use Simulink HDL Coder to generate HDL code that implements the design embodied in the model. Simulink HDL Coder generates HDL code (VHDL or Verilog) from Stateflow diagrams. You can simulate and synthesize generated HDL code using industry standard tools, and then map your system designs into FPGAs and ASICs.

The Stateflow HDL code generator is designed to

- Support the largest possible subset of Stateflow semantics that is consistent with HDL. This broad subset lets you generate HDL code from existing models without significant remodeling effort.
- Generate bit-true, cycle-accurate HDL code that is fully compatible with Stateflow simulation semantics.

In general, generation of VHDL or Verilog code from a model containing a Stateflow chart does not differ greatly from HDL code generation from any other model. However, there are a few special considerations related to Stateflow HDL code generation. This chapter describes them, in the following sections:

- "A Quick Guide to Requirements for Stateflow HDL Code Generation" on page 8-5 summarizes requirements and restrictions that apply to Stateflow charts intended for use in HDL code generation. Simulink HDL Coder supports a subset of Stateflow that is suitable for HDL code generation. The requirements and restrictions guarantee that a model can be successfully generated as HDL and that simulation results between Simulink and EDA tools will match.
- "Mapping Stateflow Chart Semantics to HDL" on page 8-9 discusses how Stateflow features map onto HDL constructs. The sections also describes aspects of Stateflow that do not lend themselves to hardware realization.

- "Structuring a Model for HDL Code Generation" on page 8-25 describes the interface between your Simulink model and your Stateflow chart that is required when generating HDL code.
- "Using Mealy and Moore Machine Types in HDL Code Generation" on page 8-16 describes the advantages of using Mealy or Moore charts as an alternative to Classic charts when generating HDL code.
- "Design Patterns Using Advanced Stateflow Features" on page 8-31 provides examples of the use of Stateflow extensions such as graphical functions, truth tables, and temporal logic in HDL code generation.

Demos and Related Documentation

Demos

Simluink HDL Coder provides several demos illustrating HDL code generation from subsystems that include Stateflow charts. These demos are:

- Greatest Common Divisor
- Pipelined Configurable FIR
- 2D FDTD Behavioral Model
- CPU Behavioral Model

To open the demo models, type the following command at the MATLAB prompt:

```
demos
```

This command opens the **Help** window. In the **Demos** pane on the left, select **Simulink > Simulink HDL Coder**. Then, double-click the icon for any of the following demos, and follow the instructions in the demo window.

Related Documentation

If you are familiar with Stateflow and Simulink but have not yet tried Simulink HDL Coder, see the hands-on exercises in Chapter 2, "Introduction to HDL Code Generation".

If you are not familiar with Stateflow, see "Getting Started with Stateflow".

For a comprehensive guide to Stateflow features, see the “Stateflow and Stateflow Coder User’s Guide”.

A Quick Guide to Requirements for Stateflow HDL Code Generation

This section summarizes the requirements and restrictions you should follow when configuring Stateflow charts that are intended to target HDL code generation. "Mapping Stateflow Chart Semantics to HDL" on page 8-9 provides a more detailed rationale for most of these requirements.

Stateflow to Simulink Interface

A Stateflow chart intended for HDL code generation must be part of a Simulink subsystem. See "Structuring a Model for HDL Code Generation" on page 8-25 for an example.

Data Type Usage

Supported Data Types

The current release supports a subset of MATLAB data types in Stateflow charts intended for use in HDL code generation. Supported data types are

- Signed and unsigned integer
- Double and single
- Fixed point without scaling (slope = 1, bias = 0, fraction length = 0)
- Boolean

Note Multidimensional arrays of the above types are supported, with the exception of data types assigned to ports. Port data types must be either scalar or vector.

Chart Initialization

In Stateflow charts intended for HDL code generation, enable the chart property **Execute (enter) Chart at Initialization**. When this property is enabled, default transitions are tested and all actions reachable from the default transition taken are executed. These actions correspond to the reset

process in HDL code. “Executing a Chart at Initialization” in the Stateflow documentation describes existing restrictions under this property.

The reset action must not entail the delay of combinatorial logic. Therefore, do not perform arithmetic in initialization actions.

Registered Output

Stateflow provides the **Initialize Outputs Every Time Chart Wakes Up** chart property specifically for HDL code generation. This property lets you control whether output is persistent (stored in registers) from one sample time to the next. Such use of registers is termed *registered output*.

When the **Initialize Outputs Every Time Chart Wakes Up** option is deselected (the default), registered output is used.

When the **Initialize Outputs Every Time Chart Wakes Up** option is selected, registered output is not used. A default initial value (defined in the **Initial value** field of the **Value Attributes** pane of the Data Properties dialog box) is given to each output when the chart wakes up. This assignment guarantees that there is no reference to outputs computed in previous time steps.

Restrictions on Imported Code

A Stateflow HDL chart must be entirely self-contained. The following restrictions apply:

- Do not call MATLAB functions other than `min` or `max`.
- Do not use MATLAB workspace data.
- Do not call C math functions
- If the **Enable C-like bit operations** property is disabled, do not use the exponentiation operator (`^`). The exponentiation operator is implemented with the C Math Library function `pow`.
- Do not include custom code. Any information entered in the Target Options dialog box is ignored.

Other Restrictions

Simulink HDL Coder imposes a number of additional restrictions on the use of classic Stateflow features. These limitations exist because HDL does not support some features of general-purpose sequential programming languages.

- Do not define machine-parented data, machine-parented events, or local events in a Stateflow chart from which HDL code is to be generated.

  Do not use the following implicit events:

  - `enter`
  - `exit`
  - `change`

  You can use the following implicit events:

  - `wakeup`
  - `tick`

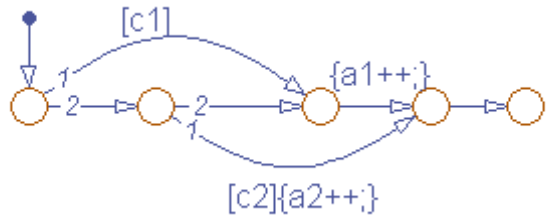
  Temporal logic can be used provided the base events are limited to the above implicit events.

- Do not use recursion through graphical functions. Simulink HDL Coder does not currently support recursion.

- Do not explicitly use loops other than `for` loops, such as in flow diagrams.

  Only constant-bounded loops are supported for HDL code generation. See the Stateflow FOR Loop demo (`sf_for.mdl`) to learn how to create a `for` loop using a graphical function.

- HDL does not support a `goto` statement. Therefore, do not use unstructured flow diagrams, such as the flow diagram shown in the following figure.



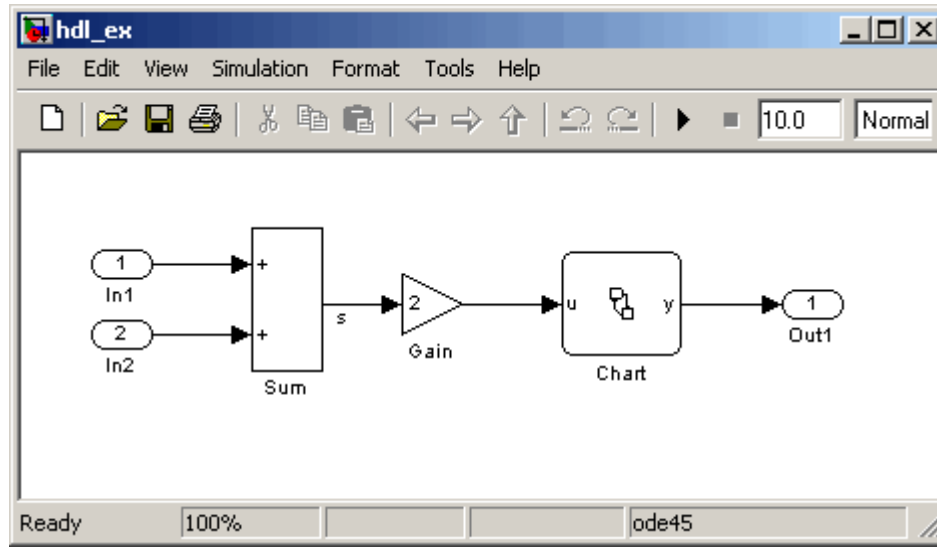
- Do not read from output data. Outputs can only be written (i.e., appear on the left side of assignments).
- Do not use Data Store Memory objects.
- Do not use pointer (`&`) or indirection (`*`) operators. See the discussion of "Pointer and Address Operations" in the Stateflow documentation.

Mapping Stateflow Chart Semantics to HDL

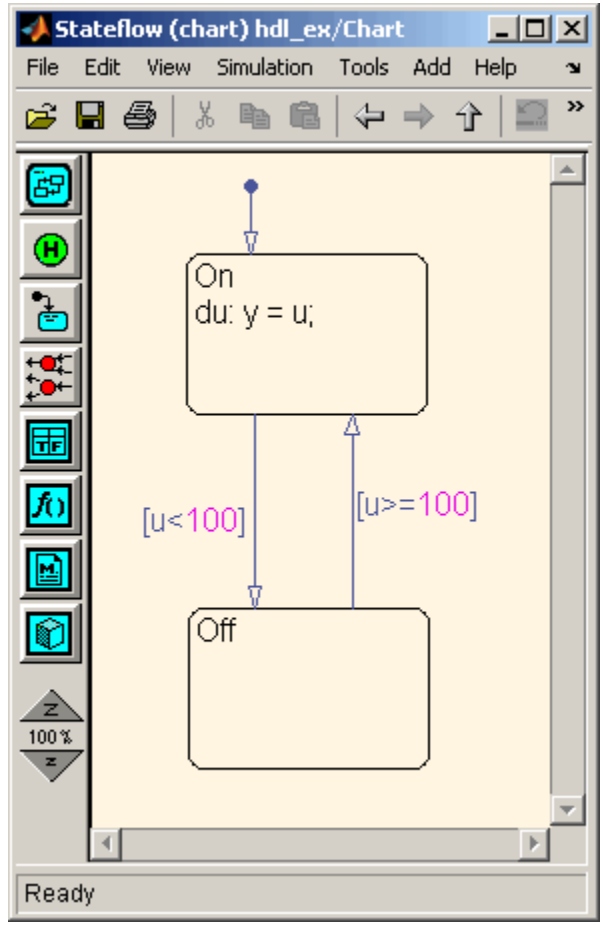
Software Realization of Stateflow Semantics

The top-down semantics of a Stateflow diagram describe how the diagram executes. Diagram semantics describe an explicit sequential execution order for elements of the diagram, such as states and transitions. These deterministic, sequential semantics map naturally to sequential programming languages, such as C. To support the rich semantics of a Stateflow diagram in the Simulink environment, it is necessary to combine the state variable updates and output computation in a single function that Simulink can call.

Consider the example mode shown in the following figure. The root level of the model contains three blocks (Sum, Gain and a Stateflow diagram) connected in series.



The Stateflow diagram from the model is shown in the following figure.



The following C code excerpt was generated by Real Time Workshop from this example model. The code illustrates how the Stateflow diagram combines the output computation and state-variable update.

```
s = In1 + In2; /* Output computation for the Sum block*/
u = s * 2.0; /* Output computation for the Gain block*/

/* Output computation and state variable update for Stateflow */
```

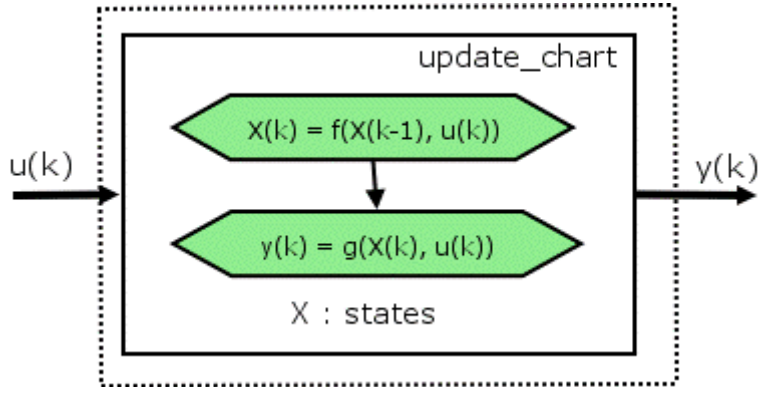
```
if(rtDWork.Chart.is_active_c1_hdl_ex == 0) {
   rtDWork.Chart.is_active_c1_hdl_ex = 1U;
   rtDWork.Chart.is_c1_hdl_ex = (uint8_T)IN_On;
 } else {
   switch(rtDWork.Chart.is_c1_hdl_ex) {
    case IN_Off:
     if(u >= 100.0) {
       rtDWork.Chart.is_c1_hdl_ex = (uint8_T)IN_On;
     }
     break;
    case IN_On:
     if(u < 100.0) {
       rtDWork.Chart.is_c1_hdl_ex = (uint8_T)IN_Off;
     } else {
       y = u;  /* Assignment to output variable */
     }
     break;
    default:
     rtDWork.Chart.is_c1_hdl_ex = (uint8_T)IN_On;
     break;

   }
 }
```

The preceding code assigns either the state or the output, but not both. Values of output variables, as well as state, persist from one time step to another. If an output value is not assigned during a chart execution, the output simply retains its value (as defined in a previous execution).

Hardware Realization of Stateflow Semantics

The following diagram shows a sequential implementation of Stateflow semantics for output/update computations, appropriate for targeting the C language.



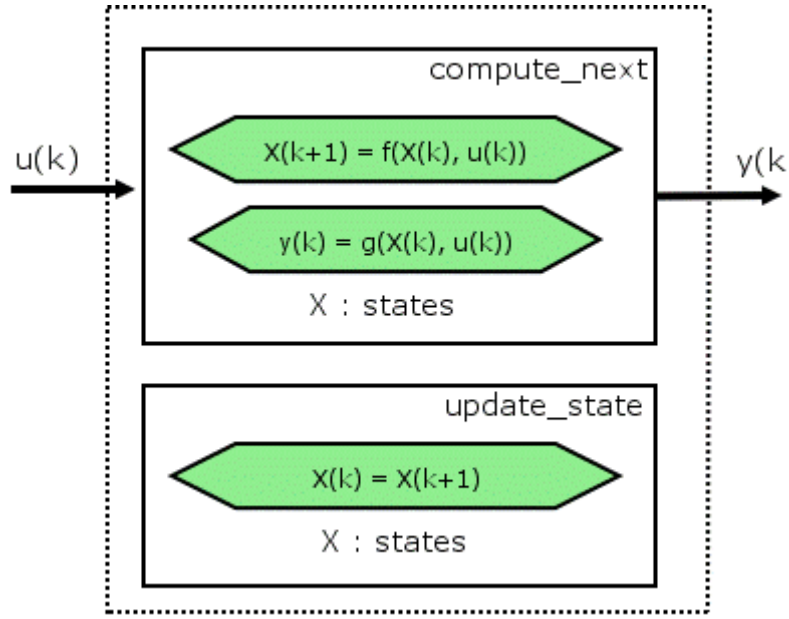
A mapping from Stateflow semantics to an HDL implementation demands a different approach. The following requirements must be met:

- **Requirement 1**: Hardware designs require separability of output and state update functions.
- **Requirement 2**: HDL is a concurrent language. To achieve the goal of bit-true simulation, execution ordering must be correct.

To meet Requirement 1, an FSM is coded in HDL as two concurrent blocks that execute under different conditions. One block evaluates the transition conditions, computes outputs and speculatively computes the next state variables. The other block updates the current state variables from the available next state and performs the actual state transitions. This second block is activated only on the trigger edge of the clock signal, or an asynchronous reset signal.

In practice, output computations usually occur more often than state updates. The presence of inputs drives the computation of outputs. State transitions occur at regular intervals (whenever the chart is activated).

The following diagram shows a concurrent implementation of Stateflow semantics for output and update computations, appropriate for targeting HDL.



The HDL code generator reuses the original single-function implementation of Stateflow semantics almost without modification. There is one important difference: instead of computing with state variables directly, all state computations are performed on local shadow variables. These variables are local to the HDL function `update_chart`. At the beginning of the `update_chart` functions, `current_state` is copied into the shadow variables. At the end of the `update_chart` function, the newly computed state is transferred to registers called collectively `next_state`. The values held in these registers are copied to `current_state` (also registered) when `update_state` is called.

By using local variables, this approach maps Stateflow sequential semantics to HDL sequential statements, avoiding the use of concurrent statements. For instance, Stateflow local variables in function scope map to VHDL variables in process scope. In VHDL, variable assignment is sequential. Therefore, statements in a Stateflow function that uses local variables can safely map to statements in a VHDL process that uses corresponding variables. The VHDL assignments execute in the same order as the assignments in the Stateflow function. The execution sequence is automatically correct.

Restrictions for HDL Realization

Some restrictions on Stateflow usage are required to achieve a valid mapping from Stateflow to HDL code. These are summarized briefly in "A Quick Guide to Requirements for Stateflow HDL Code Generation" on page 8-5. The following sections give a more detailed rationale for most of these restrictions.

Self-Contained Stateflow Charts

The Stateflow C target allows generated code to have some dependencies on code or data that is external to the diagram. Stateflow charts intended for HDL code generation, however, must be self-contained. Observe the following rules for creating self-contained charts:

- Do not use C math functions such as `sin` and `pow`. There is no HDL counterpart to the C math library.
- Do not use calls to functions coded in M or any language other than HDL. For example, do not call M functions for a simulation target, as in the following statement:

  ```
  ml.disp( hello )
  ```

- Do not use custom code. Stateflow does not provide a mechanism for embedding external HDL code into Stateflow generated HDL code. Custom C code (user-written C code intended for linkage with C code generated from a Stateflow chart) is ignored during HDL code generation.

  See also Chapter 7, "Interfacing Subsystems and Models to HDL Code".

- Do not use pointer (`&`) or indirection (`*`) operators. Pointer and indirection operators have no function in the Stateflow action language in the absence of custom code. Also, pointer and indirection operators do not map directly to synthesizable HDL.
- Do not share data (via machine-parented data or Data Store Memory blocks) between charts. Simulink HDL Coder does not map such global data to HDL, because HDL does not support global data.

Stateflow Charts Must Not Use Features Unsupported by HDL

When creating Stateflow charts intended for HDL code generation, follow these guidelines to avoid using Stateflow language features that cannot be mapped to HDL:

- Avoid recursion. While Stateflow permits recursion (through both event processing and user-written recursive graphical functions), HDL does not allow recursion.
- Do not use Stateflow machine-parented and local events. These event types do not have equivalents in HDL. Therefore, these event types are not supported for HDL code generation.
- Avoid unstructured code. Although Stateflow allows unstructured code to be written (through transition flow diagrams and graphical functions), this usage results in `goto` statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements.
- Select the **Execute (enter) Chart At Initialization** chart property. This option executes the update chart function immediately following chart initialization. The option is needed for HDL because outputs must be available at time 0 (hardware reset). You must select this option to ensure bit-true HDL code generation.

Using Mealy and Moore Machine Types in HDL Code Generation

Stateflow supports modeling of three types of state machines:

- Classic (default)
- Mealy
- Moore

This section discusses issues you should consider when generating HDL code for Mealy and Moore state machines. See "Building Mealy and Moore Charts in Stateflow" in the Stateflow documentation for detailed information on Mealy and Moore state machines.

Mealy and Moore state machines differ in the following ways:

- The outputs of a Mealy state machine are a function of the current state and inputs.
- The outputs of a Moore state machine are a function of the current state only.

Moore and Mealy state charts can be functionally equivalent; an equivalent Mealy chart can derive from a Moore chart, and vice versa. A Mealy state machine has a richer description and usually requires a smaller number of states.

The principal advantages of using Mealy or Moore charts as an alternative to Classic charts are:

- Stateflow verifies the Mealy and Moore charts you create to ensure that they conform to their formal definitions and semantic rules. Stateflow reports violations at compile time (not at design time).
- Moore charts provide a more efficient implementation of Stateflow than Classic charts, for both C and HDL targets.

The execution of a Mealy or Moore chart at time `t` is the evaluation of the function represented by that chart at time `t`. The initialization property for

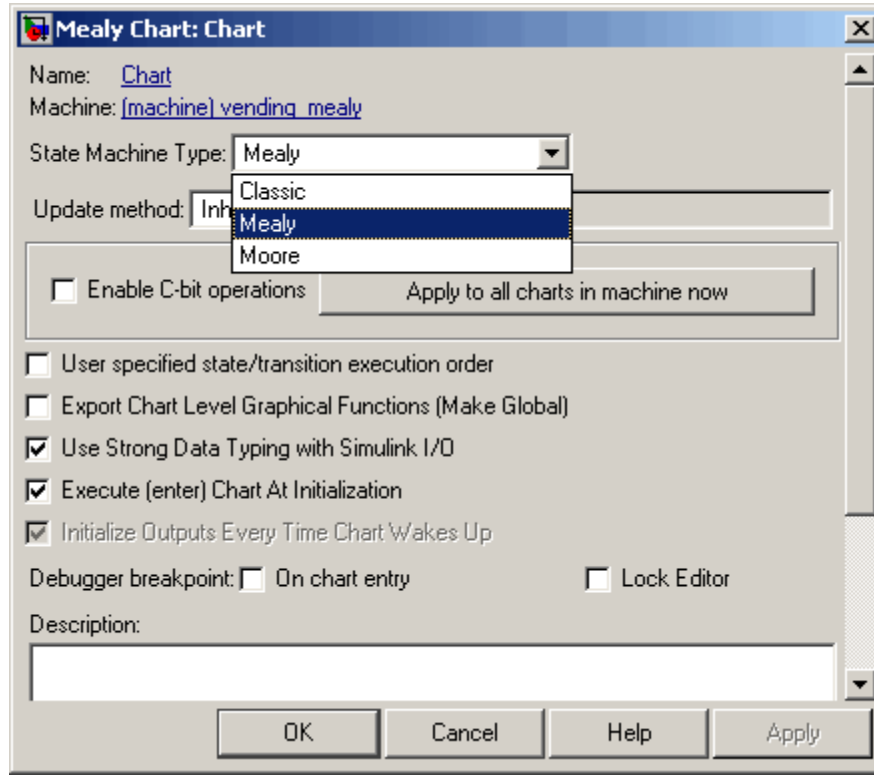
output ensures that every output is defined at every time step. Specifically, the output of a Mealy or Moore chart at one time step must not depend on the output of the chart at an earlier time step.

Consider the outputs of a Stateflow chart. Stateflow permits output latching. That is, the value of an output computed at time `t` persists until time `t+d`, when it is overwritten. The output latching feature in Stateflow corresponds to registered outputs. Therefore, Mealy and Moore charts intended for HDL code generation should not use registered outputs.

Generating HDL for a Mealy Finite State Machine

When generating HDL code for a chart that models a Mealy state machine, make sure that

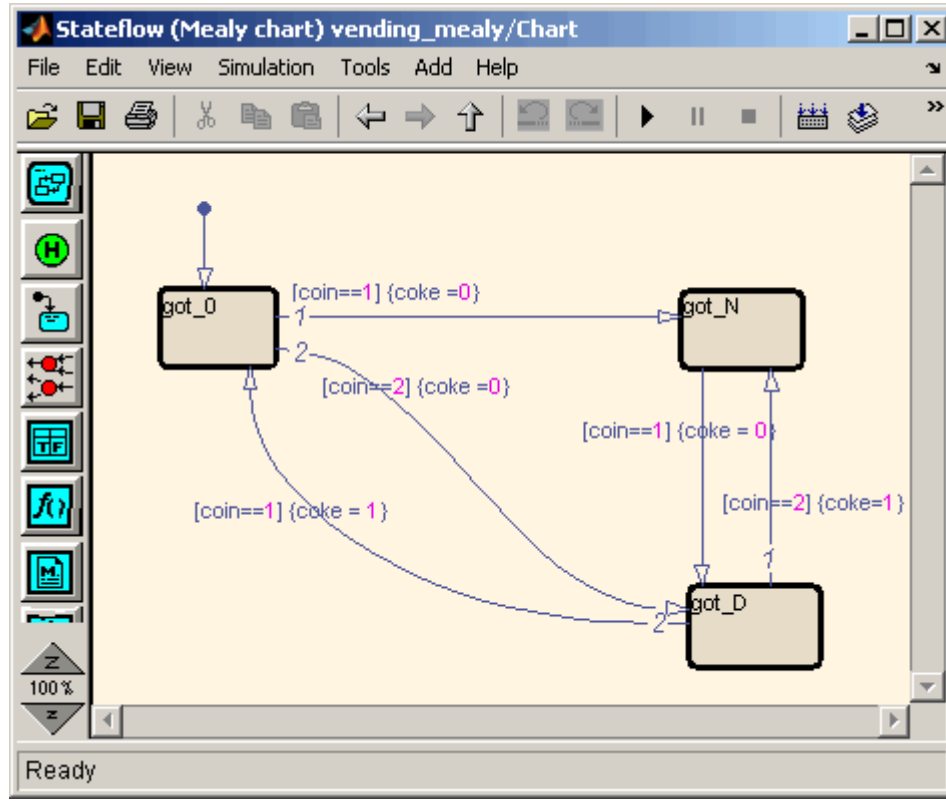
- The chart meets all general code generation requirements, as described in "A Quick Guide to Requirements for Stateflow HDL Code Generation" on page 8-5.
- The **Initialize Outputs Every Time Chart Wakes Up** option is selected. This option is selected automatically when the `Mealy` option is selected from the **State Machine Type** pop-up menu, as shown in the following figure.



- Actions are associated with transitions inner and outer transitions only.

Mealy actions are associated with transitions. In Mealy machines, output computation is expected to be driven by the change on inputs. In fact, the dependence of output on input is the fundamental distinguishing factor between the formal definitions of Mealy and Moore machines. The requirement that actions be given on transitions is to some degree stylistic, rather than necessary to enforce Mealy semantics. However, it is natural that output computation follows input conditions on input, because transition conditions are primarily input conditions in any machine type.

The following figure shows an example of a Stateflow chart that models a Mealy state machine.



The following code example lists the VHDL process code generated for the Mealy chart.

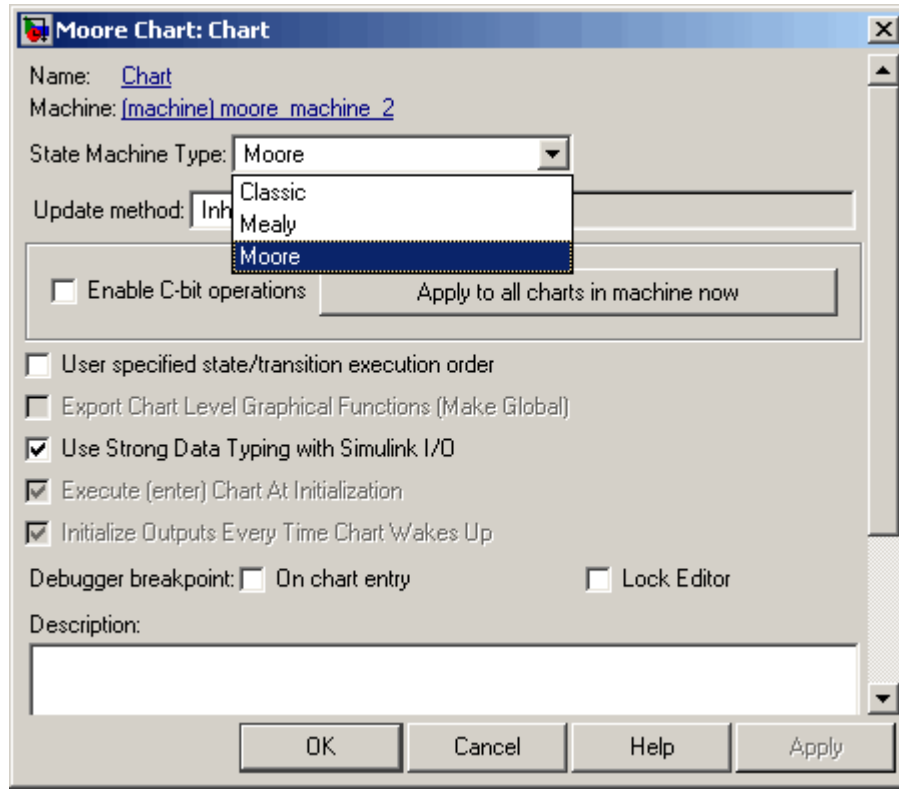
```
PROCESS (reset, clk)
        -- local variables
    BEGIN
        IF reset = '1' THEN
            is_c1_vending_mealy <= IN_got_0;
        ELSIF clk'EVENT AND clk= '1' THEN
            IF clkenable= '1' THEN
                is_c1_vending_mealy <= is_c1_vending_mealy_next;
            END IF;
        END IF;
    END PROCESS;
```

```
c1_vending_mealy : PROCESS (is_c1_vending_mealy, coin)
    -- local variables
    VARIABLE is_c1_vending_mealy_temp :     T_state_type_is_c1_vending_mealy;
BEGIN
    is_c1_vending_mealy_temp := is_c1_vending_mealy;
    coke <= '0';
    CASE is_c1_vending_mealy_temp IS
        WHEN IN_got_0 =>
            IF coin = 1.0 THEN
                coke <= '0';
                is_c1_vending_mealy_temp := IN_got_N;
            ELSE
                IF coin = 2.0 THEN
                    coke <= '0';
                    is_c1_vending_mealy_temp := IN_got_D;
                END IF;
            END IF;
        WHEN IN_got_D =>
            IF coin = 2.0 THEN
                coke <= '1';
                is_c1_vending_mealy_temp := IN_got_N;
            ELSE
                IF coin = 1.0 THEN
                    coke <= '1';
                    is_c1_vending_mealy_temp := IN_got_0;
                END IF;
            END IF;
        WHEN IN_got_N =>
            IF coin = 1.0 THEN
                coke <= '0';
                is_c1_vending_mealy_temp := IN_got_D;
            END IF;
  END CASE;
    is_c1_vending_mealy_next <= is_c1_vending_mealy_temp;
END PROCESS;
```

Generating HDL Code for a Moore Finite State Machine

When generating HDL code for a chart that models a Moore state machine, make sure that

- The chart meets all general code generation requirements, as described in "A Quick Guide to Requirements for Stateflow HDL Code Generation" on page 8-5.
- The **Initialize Outputs Every Time Chart Wakes Up** option is selected. This option is selected automatically when the `Moore` option is selected from the **State Machine Type** pop-up menu, as shown in the following figure.



- Actions occur in states only. These actions are unlabeled, and execute when exiting the states or remaining in the states.

Moore actions must be associated with states, because output computation must be dependent only on states, not input. Therefore, the current configuration of active states at time step `t` determines output. Thus, the single action in a Moore state serves as both `during` and `exit` action. If state `S` is active when a chart wakes up at time `t`, it contributes to the output whether it remains active into time `t+1` or not.

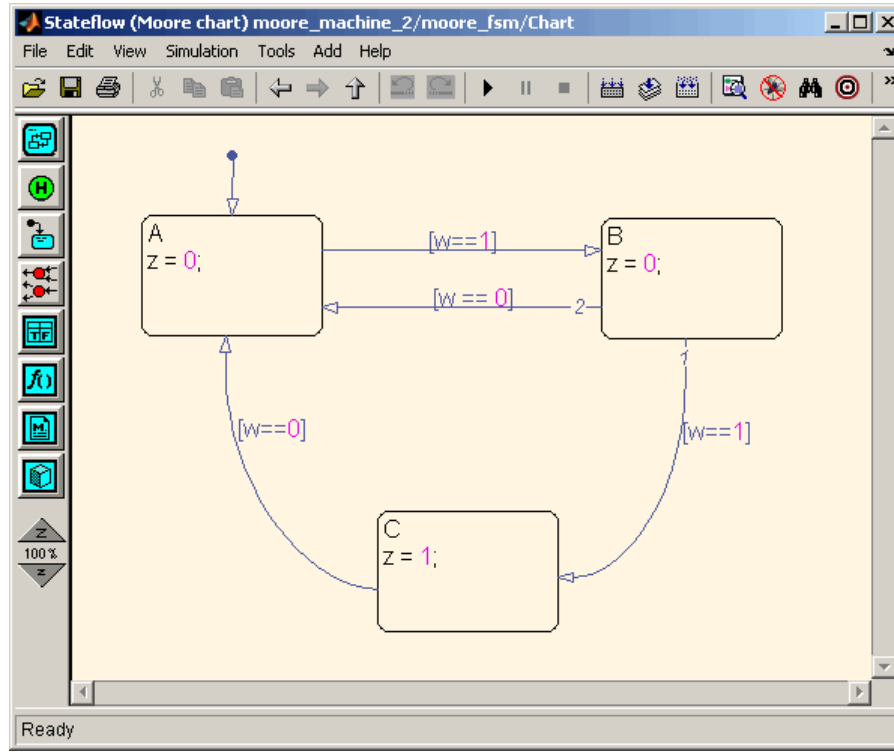
- No local data or graphical functions are used.

  Function calls and local data are not allowed in a Moore chart. This ensures that output does not depend on input in ways that would be difficult for the HDL code generator to verify. These restrictions strongly encourage coding practices that separate output and input.

- No references to input occur outside of transition conditions.
- Output computation occurs only in leaf states.

  This restriction guarantees that Stateflow's top-down semantics compute outputs as if actions were evaluated strictly before inner and outer flow diagrams.

The following figure shows a Stateflow chart of a Moore state machine.



The following code example illustrates generated Verilog code for the Moore chart.

```
reg [2:0]  is_c1_moore_machine_2_temp;
     parameter IN_NO_ACTIVE_CHILD = 0, IN_A = 1, IN_B = 2, IN_C = 3;
     always @(posedge(reset), posedge(clk))
        begin
            if (reset)
                is_c1_moore_machine_2 = IN_A;
            else
                if (clkenable)
                    is_c1_moore_machine_2 = is_c1_moore_machine_2_next;
         end
     always @(is_c1_moore_machine_2, w)
        begin
```

```
is_c1_moore_machine_2_temp = is_c1_moore_machine_2;
z = 0;
case ( is_c1_moore_machine_2_temp)
    IN_A :
        z = 0;
    IN_B :
        z = 0;
    IN_C :
        z = 1;
    default :
        is_c1_moore_machine_2_temp = IN_NO_ACTIVE_CHILD;
endcase
case ( is_c1_moore_machine_2_temp)
    IN_A :
        if (w == 1)
            is_c1_moore_machine_2_temp = IN_B;
    IN_B :
        if (w == 1)
            is_c1_moore_machine_2_temp = IN_C;
        else
            if (w == 0)
                is_c1_moore_machine_2_temp = IN_A;
    IN_C :
        if (w == 0)
            is_c1_moore_machine_2_temp = IN_A;
    default :
```

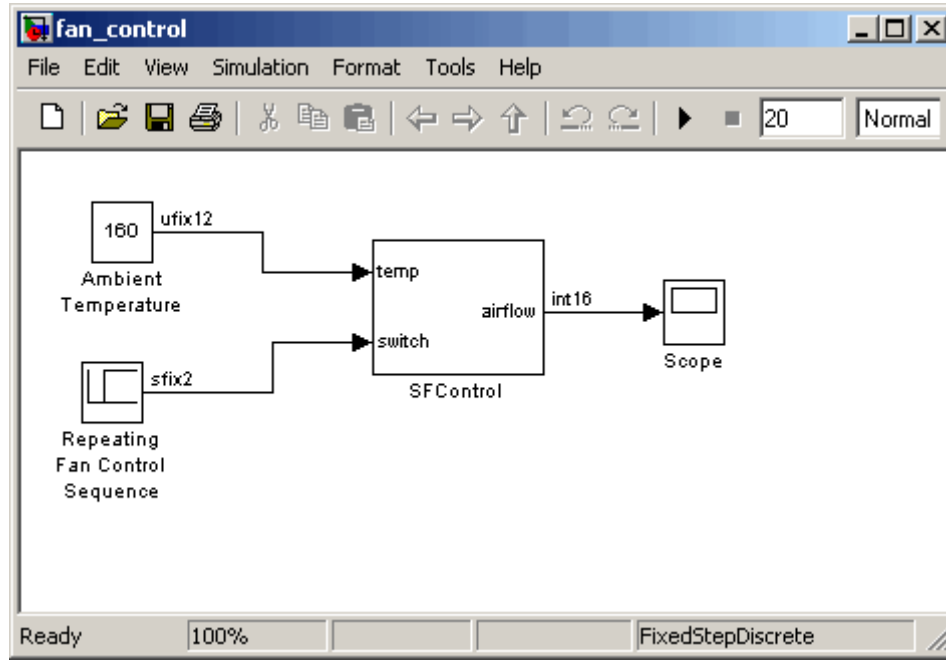
Structuring a Model for HDL Code Generation

In general, generation of VHDL or Verilog code from a model containing a Stateflow chart does not differ greatly from HDL code generation from any other model.

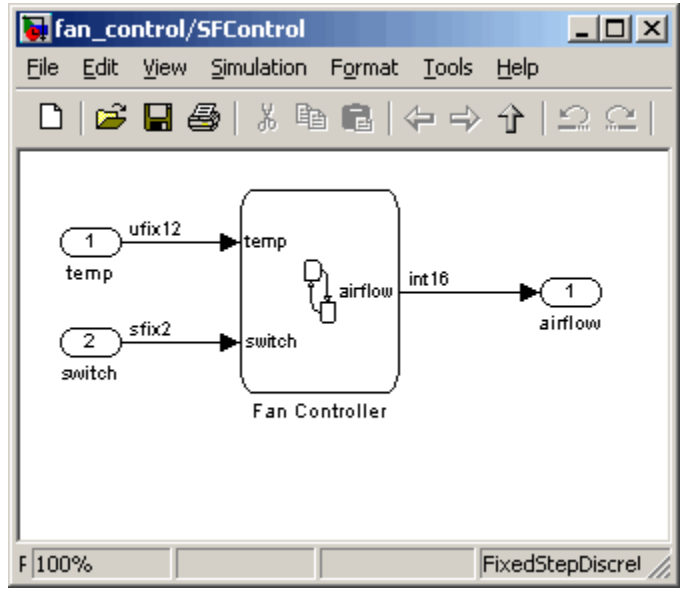
A Stateflow chart intended for HDL code generation *must* be part of a Simulink subsystem that represents the Device Under Test (DUT). The DUT corresponds to the top level VHDL entity or Verilog module for which code is generated, tested and eventually synthesized. The top level components in Simulink that drive the DUT correspond to the behavioral Simulink test bench.

You may need to restructure your models to meet this requirement. If the Stateflow chart for which you want to generate code is at the root level of your model, embed the chart in a subsystem and connect the appropriate signals to the subsystem inputs and outputs. In most cases, you can do this by simply clicking on the chart and then selecting **Edit > Create Subsystem** in the model window.

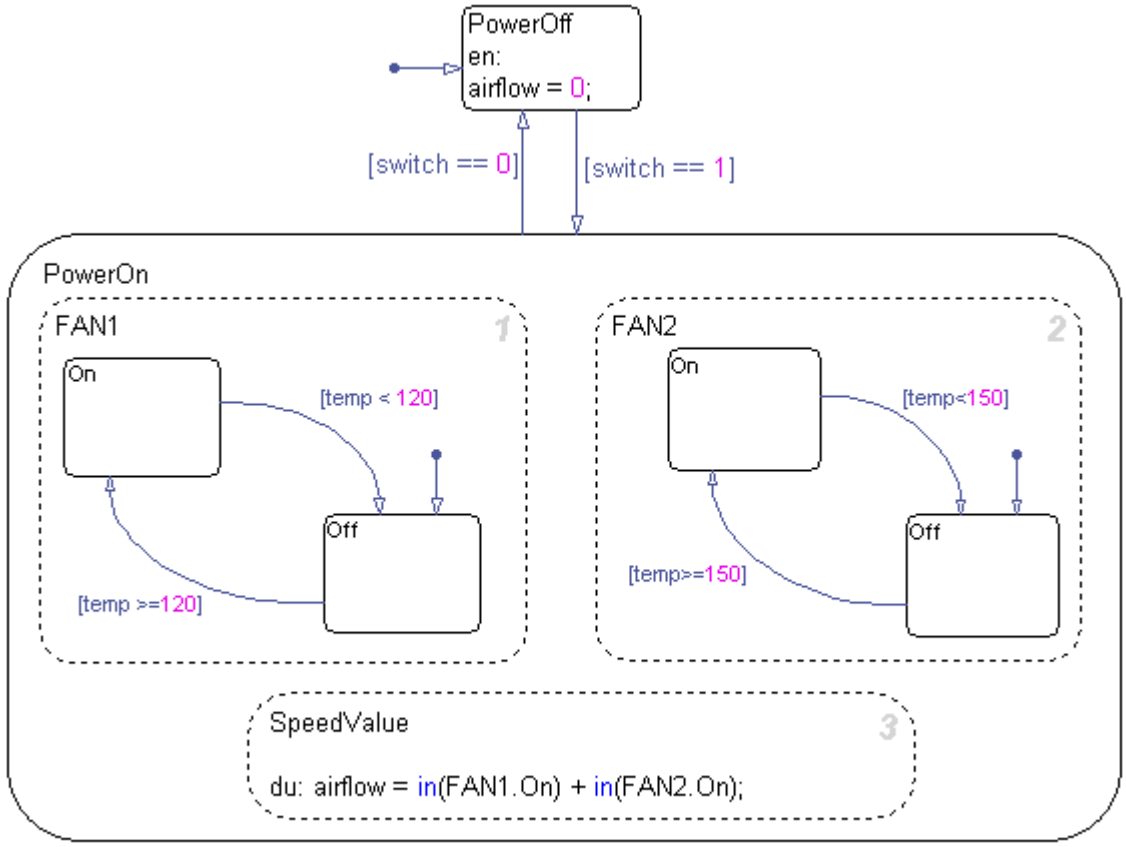
As an example of a properly structured model, consider the `fan_control` model shown in the following figure. In this model, the subsystem `SFControl` is the DUT. Two input signals drive the DUT.



The `SFControl` subsystem, shown in the following figure, contains a Stateflow chart, `Fan Controller`. The chart that has two inputs and an output.



The `Fan Controller` chart, shown in the following figure, models a simple system that monitors input temperature data (`temp`) and turns on the two fans (`FAN1` and `FAN2`) based on the range of the temperature. A manual override input (`switch`) is provided to turn the fans off forcibly. At each time step the `Fan Controller` outputs a value (`airflow`) representing the number of fans that are turned on.



The following `makehdl` command generates VHDL code (by default) for the subsystem containing the Stateflow chart.

```
makehdl(`fan_control/SF_Control')
```

As code generation for this subsystem proceeds, Simulink HDL Coder displays progress messages as shown in the following listing:

```
### Begin VHDL Code Generation
### Working on fan_control/SFControl as hdlsrc\SFControl.vhd
```

```
### Working on fan_control/SFControl/Fan Controller as hdlsrc\Fan_Controller.vhd
Stateflow parsing for model "fan_control"...Done
Stateflow code generation for model "fan_control"....Done
### HDL Code Generation Complete.
```

As the progress messages indicate, Simulink HDL Coder generates a separate code file for each level of hierarchy in the model. The following VHDL files are written to the target directory, `hdlsrc`:

- `Fan_Controller.vhd` contains the entity and architecture code (`Fan_Controller`) for the Stateflow chart.
- `SFControl.vhd` contains the code for the top level subsystem. This file also instantiates a `Fan_Controller` component.

Simulink HDL Coder also generates a number of other files (such as scripts for HDL simulation and synthesis tools) in the target directory. See the "HDL Code Generation Defaults" on page 12-13 for full details on generated files.

The following code excerpt shows the entity declaration generated for the `Fan_Controller` Stateflow chart in`Fan_Controller.vhd`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY Fan_Controller IS
    PORT (
        clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        temp : IN std_logic_vector(11 DOWNTO 0);
        b_switch : IN std_logic_vector(1 DOWNTO 0);
        airflow : OUT std_logic_vector(15 DOWNTO 0));
END Fan_Controller;
```

This model shows the use of fixed point data types without scaling (e.g. `ufix12`, `sfix2`) , as supported by Stateflow for HDL code generation. At the entity/instantiation boundary, all signals in the generated code are typed as `std_logic` or `std_logic_vector`, following general VHDL coding standard

conventions. In the architecture body, these signals are assigned to the corresponding typed signals for further manipulation and access.

Design Patterns Using Advanced Stateflow Features

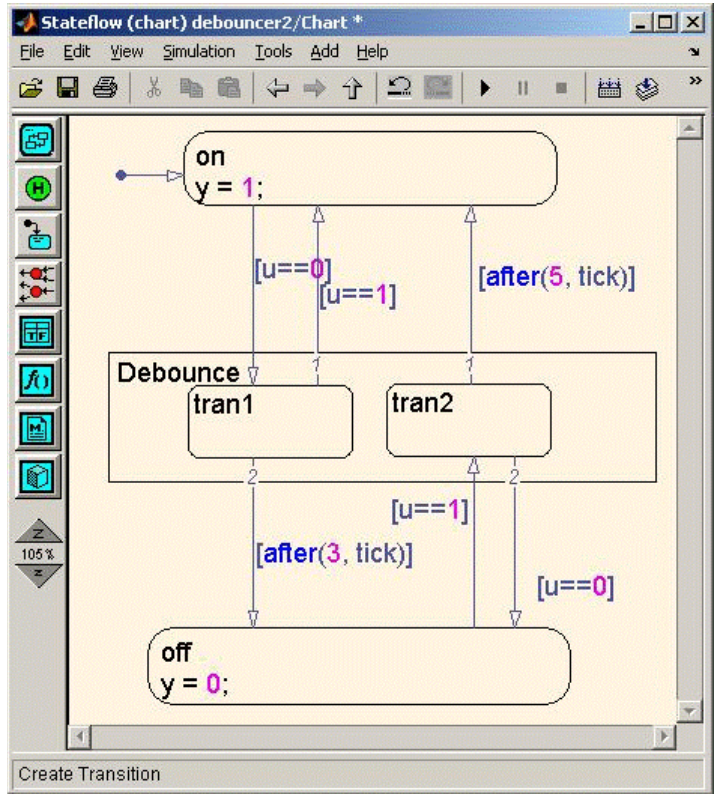
The following sections describe several design patterns that will help you to use advanced Stateflow features to generate efficient HDL code.

Temporal Logic

Stateflow temporal logic operators (such as `after`, `before`, or `every`) are Boolean operators that operate on recurrence counts of Stateflow events. Temporal logic operators can appear only in conditions on transitions that from states, and in state actions. Although temporal logic does not introduce any new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. You can use temporal logic operators in many cases where a counter is required. A common use case would be to use temporal logic to implement a time-out counter.

For detailed information about Stateflow temporal logic, see "Using Temporal Logic in Actions" in the Stateflow documentation.

The chart shown in the following figure uses temporal logic in a design for a debouncer. Instead of instantaneously switching between `on` and `off` states, the chart uses two intermediate states and temporal logic to ignore transients. The transition is committed based on a time-out.



The following code excerpt shows VHDL code generated from this chart.

```
IF temporalCounter_i1_temp < to_unsigned(3, 8) THEN
  temporalCounter_i1_temp:=
  tmw_to_unsigned(tmw_to_unsigned(temporalCounter_i1_temp,9)+to_unsigned(1, 9), 8);
        END IF;
        CASE is_c1_debouncer2_temp IS
            WHEN IN_tran1 =>
                IF u = '1' THEN
                    is_c1_debouncer2_temp := IN_on;
                ELSE
                    IF temporalCounter_i1_temp >= to_unsigned(3, 8) THEN
                        is_c1_debouncer2_temp := IN_off;
                        y_reg_temp := '0';
```

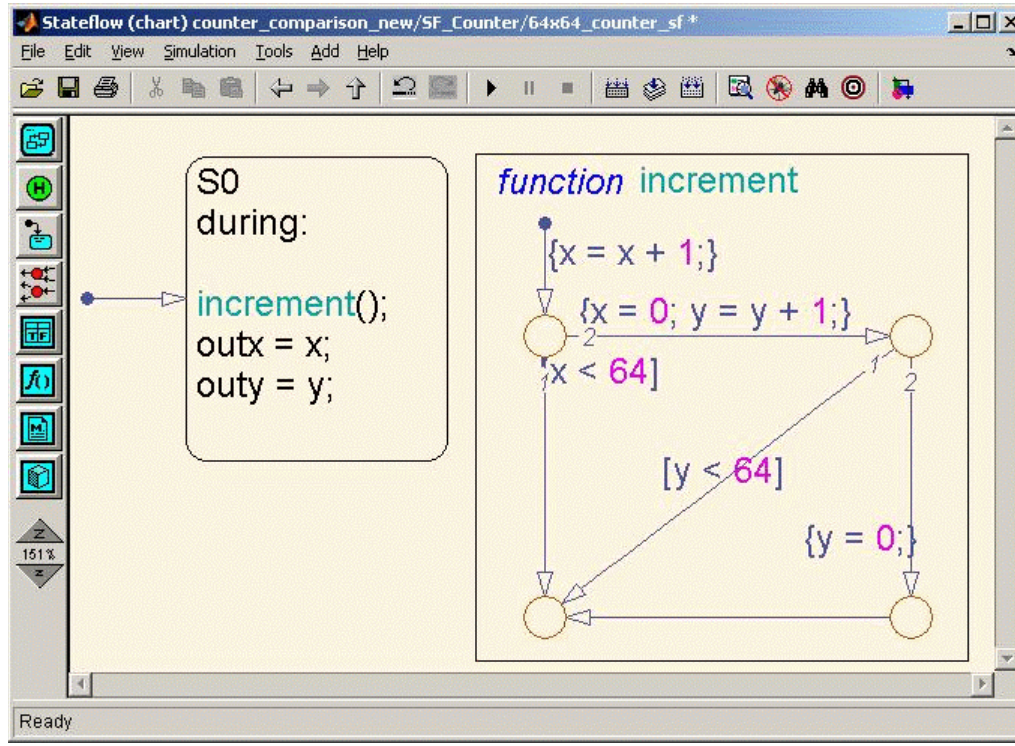
```
            END IF;
        END IF;
    WHEN IN_tran2 =>
        IF temporalCounter_i1_temp >= to_unsigned(3, 8) THEN
            is_c1_debouncer2_temp := IN_on;
        ELSE
            IF u = '0' THEN
                is_c1_debouncer2_temp := IN_off;
                y_reg_temp := '0';
            END IF;
        END IF;
    WHEN IN_off =>
        IF u = '1' THEN
            is_c1_debouncer2_temp := IN_tran2;
            temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
    WHEN IN_on =>
        IF u = '0' THEN
            y_reg_temp := '1';
            is_c1_debouncer2_temp := IN_tran1;
            temporalCounter_i1_temp := to_unsigned(0, 8);
        END IF;
    WHEN OTHERS =>
        is_c1_debouncer2_temp := IN_on;
END CASE;
```

Graphical Function

A Stateflow graphical function is a function defined graphically by a flow diagram. Graphical functions reside in a Stateflow chart along with the diagrams that invoke them. Like MATLAB and C functions, graphical functions can accept arguments and return results. Graphical functions can be invoked in transition and state actions.

The “Stateflow Notation” chapter of the Stateflow documentation includes a detailed description of graphical functions.

The following figure shows a graphical function that implements a 64–by–64 counter.



The following code excerpt shows VHDL code generated for this graphical function.

```
c1_counter_comparison_new : PROCESS (x, y, outx_reg, outy_reg)
    -- local variables
    VARIABLE x_temp : unsigned(7 DOWNTO 0);
    VARIABLE y_temp : unsigned(7 DOWNTO 0);
    VARIABLE outx_reg_temp : unsigned(7 DOWNTO 0);
    VARIABLE outy_reg_temp : unsigned(7 DOWNTO 0);
BEGIN
    x_temp := x;
    y_temp := y;
    x_temp := tmw_to_unsigned(tmw_to_unsigned(x_temp, 9) + to_unsigned(1, 9), 8);
    IF x_temp < to_unsigned(64, 8) THEN
        NULL;
    ELSE
```

```
            x_temp := to_unsigned(0, 8);
            y_temp := tmw_to_unsigned(tmw_to_unsigned(y_temp, 9) + to_unsigned(1, 9), 8);
            IF y_temp < to_unsigned(64, 8) THEN
                NULL;
            ELSE
                y_temp := to_unsigned(0, 8);
            END IF;
        END IF;
        outx_reg_temp := x_temp;
        outy_reg_temp := y_temp;
        x_next <= x_temp;
        y_next <= y_temp;
        outx_reg_next <= outx_reg_temp;
        outy_reg_next <= outy_reg_temp;
    END PROCESS;
```

Hierarchy and Parallelism

Stateflow diagrams support both hierarchy (states containing other states) and parallelism (multiple states that can be active simultaneously).

Parallelism, in Stateflow, is not synonymous with concurrency. In Stateflow semantics, parallel states can be active simultaneously, but they are executed sequentially according to their execution order. (Execution order is displayed on the upper right corner of a parallel state).

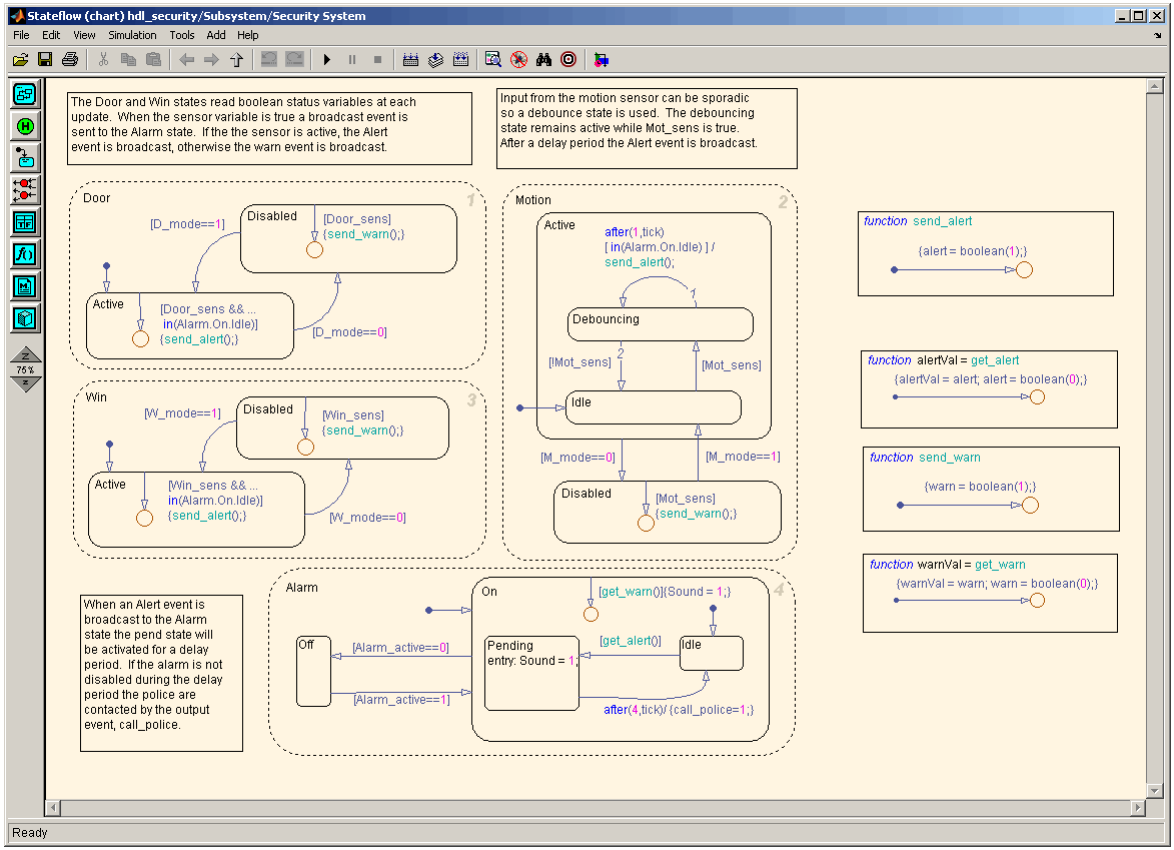
For detailed information on hierarchy and parallelism, see “Stateflow Hierarchy of Objects” and “Execution Order for Parallel States” in the Stateflow documentation.

For HDL code generation, an entire chart maps to a single output computation process. Within the output computation process:

- The execution of parallel states proceeds sequentially.
- Nested hierarchical states map to nested `CASE` statements in the generated HDL code.

The following diagram shows a chart that models a security system. The chart contains

- Simultaneously active parallel states (in order of execution: Door, Motion, Win, Alarm).
- Hierarchy, where the parallel states contain child states. For example, the Motion state contains Active and Inactive states, and the Active state contains further nested states (Debouncing and Idle).
- Graphical functions (such as `send_alert` and `send_warn`) that set and reset flags, simulating broadcast and reception of events. These functions are used, rather than Stateflow local events, because local events are not supported for HDL code generation.



The following VHDL code excerpt was generated for the parallel Door and Motion states from this chart. The higher-level `CASE` statements corresponding to Door and Motion are generated sequentially to match Stateflow simulation semantics. The hierarchy of nested states maps to nested `CASE` statements in VHDL.

```
CASE is_Door_temp IS
            WHEN IN_Active =>
                IF D_mode = '0' THEN
                    is_Door_temp := IN_Disabled;
                ELSE
                    IF tmw_to_boolean(Door_sens AND
                    tmw_to_stdlogic(is_On_temp = IN_Idle)) THEN
                        alert_temp := '1';
                    END IF;
                END IF;
            WHEN IN_Disabled =>
                IF D_mode = '1' THEN
                    is_Door_temp := IN_Active;
                ELSE
                    IF tmw_to_boolean(Door_sens) THEN
                        warn_temp := '1';
                    END IF;
                END IF;
            WHEN OTHERS =>
                --On the first sample call the door mode is set to active.
                is_Door_temp := IN_Active;
        END CASE;
        --This state models the modes of a motion detector sensor
        --and implements logic to respond when that sensor is producing a signal.
        CASE is_Motion_temp IS
            WHEN IN_Active =>
                IF M_mode = '0' THEN
                    is_Active_temp := IN_NO_ACTIVE_CHILD;
                    is_Motion_temp := IN_Disabled;
                ELSE
                    CASE is_Active_temp IS
                        WHEN IN_Debouncing =>
                            IF (temporalCounter_i2_temp >= to_unsigned(1, 8))
                            AND (is_On_temp = IN_Idle) THEN
```

```
            alert_temp := '1';
            is_Active_temp := IN_Debouncing;
            temporalCounter_i2_temp := to_unsigned(0, 8);
        ELSE
            IF tmw_to_boolean( NOT Mot_sens) THEN
                is_Active_temp := b_IN_Idle;
            END IF;
        END IF;
    WHEN b_IN_Idle =>
        IF tmw_to_boolean(Mot_sens) THEN
            is_Active_temp := IN_Debouncing;
            temporalCounter_i2_temp := to_unsigned(0, 8);
        END IF;
```

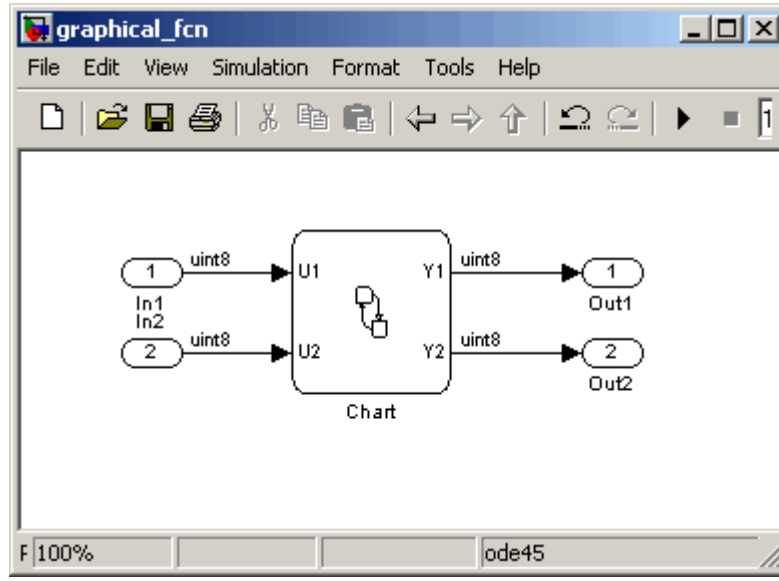
Stateless Charts

Stateflow charts consisting of pure flow diagrams (i.e., charts having no states) are useful in capturing `if-then-else` constructs used in procedural languages like C. The “Stateflow Notation” chapter in the Stateflow documentation discusses flow diagrams in detail.

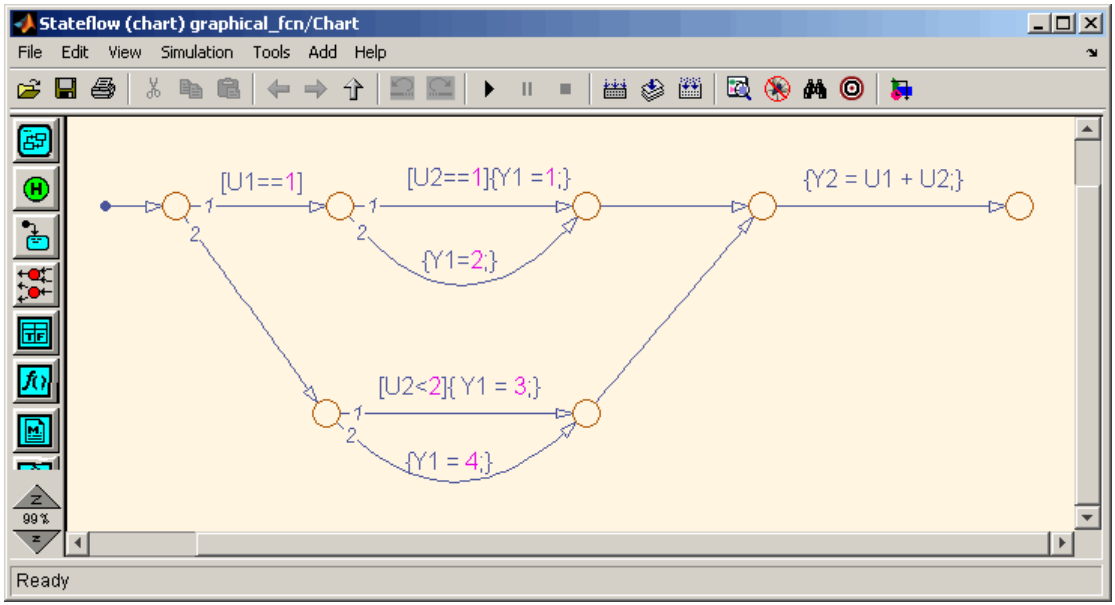
As an example, consider the following logic, expressed in C-like pseudocode.

```
if(U1==1) {
    if(U2==1) {
       Y = 1;
    }else{
      Y = 2;
    }
}else{
   if(U2<2) {
      Y = 3;
   }else{
      Y = 4;
    }
      }
```

The following figures illustrate how to model this control flow using a stateless Stateflow diagram. The root model contains a subsystem and inputs and outputs to the chart.


graphical_fcn
File Edit View Simulation Format Tools Help
1
uint8
In1
In2
2
uint8
U1
U2
Y1
Y2
uint8
uint8
1
Out1
2
Out2
Chart
100%
ode45

The following figure shows the Stateflow flow diagram that implements the `if-then-else` logic.



The following generated VHDL code excerpt shows the nested IF-ELSE statements obtained from the flow diagram.

```
c1_graphical_fcn : PROCESS (Y1_reg, Y2_reg, U1, U2)
        -- local variables
        VARIABLE Y1_reg_temp : unsigned(7 DOWNTO 0);
        VARIABLE Y2_reg_temp : unsigned(7 DOWNTO 0);
    BEGIN
        IF unsigned(U1) = to_unsigned(1, 8) THEN
            IF unsigned(U2) = to_unsigned(1, 8) THEN
                Y1_reg_temp := to_unsigned(1, 8);
            ELSE
                Y1_reg_temp := to_unsigned(2, 8);
            END IF;
        ELSE
            IF unsigned(U2) < to_unsigned(2, 8) THEN
                Y1_reg_temp := to_unsigned(3, 8);
```

```
            ELSE
                Y1_reg_temp := to_unsigned(4, 8);
            END IF;
        END IF;
        Y2_reg_temp := tmw_to_unsigned(tmw_to_unsigned(unsigned(U1), 9) + tmw_to_unsigned(unsigned(U2), 9
        Y1_reg_next <= Y1_reg_temp;
        Y2_reg_next <= Y2_reg_temp;
    END PROCESS;
```

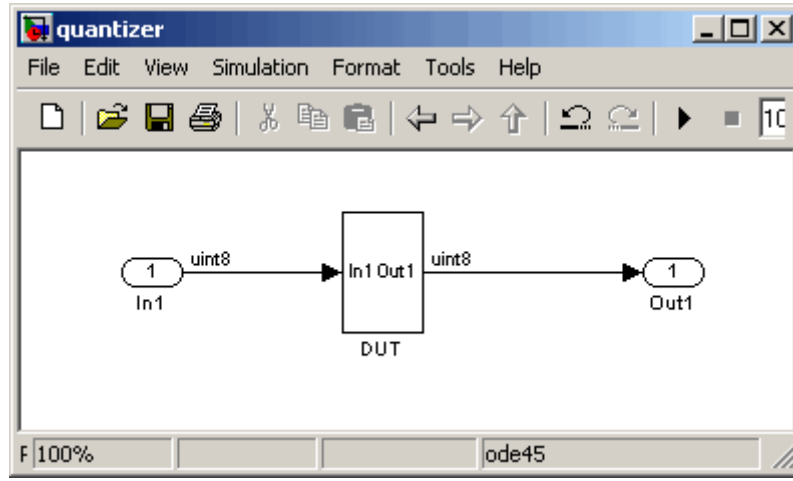
Truth Tables

Stateflow Truth Table functions (see "Truth Table Functions" in the Stateflow documentation) are well-suited for implementing compact combinatorial logic. A typical application for Truth Tables is to implement nonlinear quantization or threshold logic. Consider the following logic:

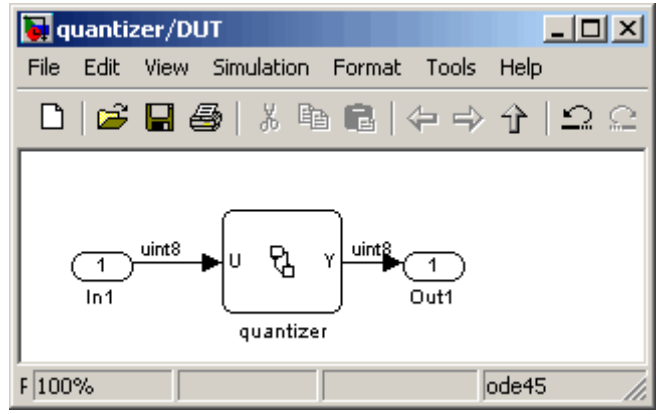
```
Y = 1 when 0  <= U <= 10
Y = 2 when 10 <  U <= 17
Y = 3 when 17 <  U <= 45
Y = 4 when 45 <  U <= 52
Y = 5 when 52 <  U
```

A stateless chart with a single call to a Truth Table function can represent this logic succinctly.

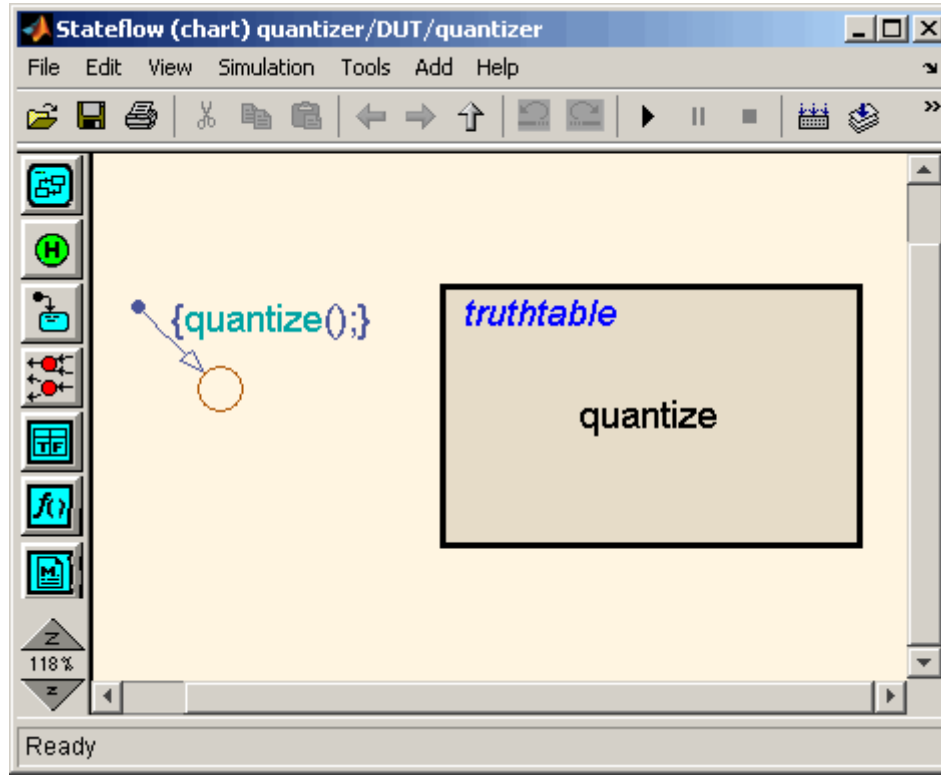
The following figure shows a model containing a subsystem, DUT.



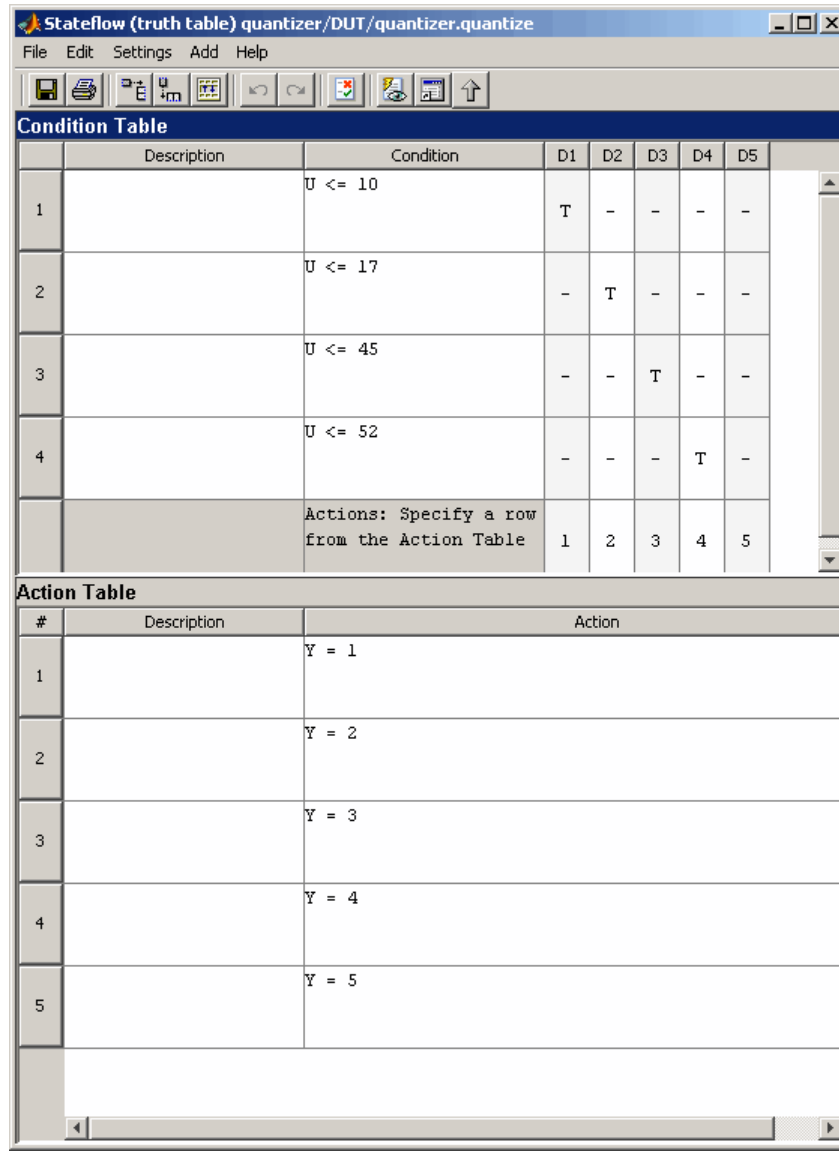
The subsystem contains a chart, `quantizer`, as shown in the following figure.



The next figure shows the `quantizer` chart, containing the Truth Table.



The following figure shows the threshold logic, as displayed in the Truth Table Editor.



The following code excerpt shows VHDL code generated for the `quantizer` chart.

```
quantizer : PROCESS (Y_reg, U)
    -- local variables
    VARIABLE aVarTruthTableCondition_1 : std_logic;
    VARIABLE aVarTruthTableCondition_2 : std_logic;
    VARIABLE aVarTruthTableCondition_3 : std_logic;
    VARIABLE aVarTruthTableCondition_4 : std_logic;
    VARIABLE Y_reg_temp : unsigned(7 DOWNTO 0);
BEGIN
    -- Condition #1
    aVarTruthTableCondition_1 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(10, 8));
    -- Condition #2
    aVarTruthTableCondition_2 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(17, 8));
    -- Condition #3
    aVarTruthTableCondition_3 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(45, 8));
    -- Condition #4
    aVarTruthTableCondition_4 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(52, 8));
    IF tmw_to_boolean(aVarTruthTableCondition_1) THEN
        -- D1
        -- Action 1
        Y_reg_temp := to_unsigned(1, 8);
    ELSE
        IF tmw_to_boolean(aVarTruthTableCondition_2) THEN
            -- D2
            -- Action 2
            Y_reg_temp := to_unsigned(2, 8);
        ELSE
            IF tmw_to_boolean(aVarTruthTableCondition_3) THEN
                -- D3
                -- Action 3
                Y_reg_temp := to_unsigned(3, 8);
            ELSE
                IF tmw_to_boolean(aVarTruthTableCondition_4) THEN
                    -- D4
                    -- Action 4
                    Y_reg_temp := to_unsigned(4, 8);
                ELSE
                    -- Default
```

```
                    -- Action 5
                    Y_reg_temp := to_unsigned(5, 8);
                END IF;
            END IF;
        END IF;
    END IF;
    Y_reg_next <= Y_reg_temp;
END PROCESS;
```

9

Generating Scripts for HDL Simulators and Synthesis Tools

Overview of Script Generation for EDA Tools

Simulink HDL Coder supports generation of script files for third-party electronic design automation (EDA) tools. These scripts let you compile and simulate generated HDL code or synthesize generated HDL code.

Using the defaults, you can automatically generate scripts for the following tools:

- Mentor Graphics ModelSim SE/PE HDL simulator
- The Synplify family of synthesis tools

You can customize both the names and the content of generated script files. To do this, you must use the `makehdl` or `makehdltb` function, and pass in the appropriate property name/property value arguments as described in "Customizing Script Names" on page 9-4 and "Customizing Script Code" on page 9-5.

Defaults for Script Generation

If the `EDAScriptGeneration` property is set `'on'` (the default), script generation takes place automatically, as part of the code and test bench generation process. To disable script generation, set `EDAScriptGeneration` to `'off'`.

All script files are generated in the target directory.

When you generate HDL code for a model or subsystem *`system`*, Simulink HDL Coder writes the following script files:

- *`system`*`_compile.do`: ModelSim compilation script. This script contains commands to compile the generated code, but not to simulate it.
- *`system`*`_synplify.tcl`: Synplify synthesis script

When you generate test bench code for a model or subsystem *`system`*, Simulink HDL Coder writes the following script files:

- *`system`*`_tb_compile.do`: ModelSim compilation script. This script contains commands to compile the generated code and test bench.
- *`system`*`_tb_sim.do`: ModelSim simulation script. This script contains commands to run a simulation of the generated code and test bench.

Customizing Script Names

When you generate HDL code, script names are generated by appending a postfix string to the model or subsystem name *system*.

When you generate test bench code , script names are generated by appending a postfix string to the test bench name *testbench_tb*.

The postfix string depends on the type of script (compilation, simulation, or synthesis) being generated. The default postfix strings are shown in the table below. For each type of script, you can define your own postfix using the associated property.

Script Type	Property	Default Value
Compilation	'HDLCompileFilePostfix'	'_compile.do'
Simulation	'HDLSimFilePostfix'	'_sim.do'
Synthesis	'HDLSynthFilePostfix'	'_synplify.tcl'

In the following example, VHDL code is generated for the subsystem `system`. A custom postfix string is specified for the compilation script. The name of the generated compilation script will be `system_test_compilation.do`.

```
makehdl('mymodel/system', 'HDLCompileFilePostfix', '_test_compilation.do')
```

Customizing Script Code

A generated EDA script consists of three sections, generated and executed in the following order:

1. An initialization (`Init`) phase. The `Init` phase performs any required setup actions, such as creating a design library or a project file. Some arguments to the `Init` phase are implicit, for example, the top-level entity or module name.

   Properties that apply to the `Init` phase are identified by the substring `Init` in the property name.

2. A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file or once per signal. On each call, a different file or signal name is passed in.

   Properties that apply to the `Cmd` phase are identified by the substring `Cmd` in the property name.

3. A termination phase (`Term`). This is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase takes no arguments.

   Properties that apply to the `Term` phase are identified by the substring `Term` in the property name.

`makehdl` and `makehdltb` generate scripts by passing format strings to the MATLAB `fprintf` function. Using the property name/property value pairs summarized in the table below, you can pass in customized format strings to `makehdl` or `makehdltb`.

You can use any legal `fprintf` formatting characters. For example, `'\n'` inserts a newline into the script file.

Some of these format strings can take arguments, such as the top-level entity or module name, or the names of the VHDL or Verilog files in the design. The `'HDLSimViewWaveCommand'` format string takes the top-level signal names as its argument.

Property Name and Default	Description
Name: `'HDLCompileInit'` Default:`'vlib work\n'`	Format string passed to `fprintf` to write the `Init` section of the compilation script.
Name: `'HDLCompileVHDLCmd'` Default: `'vcom %s %s\n'`	Format string passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files. The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).
Name: `'HDLCompileVerilogCmd'` Default: `'vlog %s %s\n'`	Format string passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files. The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).
Name:`'HDLCompileTerm'` Default:`''`	Format string passed to `fprintf` to write the termination portion of the compilation script.

Name: `'HDLSimInit'`
Default:
`['onbreak resume\n',...`
`'onerror resume\n']`
	Format string passed to `fprintf` to write the initialization section of the simulation script.

Name: `'HDLSimCmd'` Default: `'vsim work.%s\n'`	Format string passed to `fprintf` to write the simulation command. The implicit argument is the top-level module or entity name.

Property Name and Default	Description
Name: `'HDLSimViewWaveCmd'` Default: `'add wave sim:%s\n'`	Format string passed to `fprintf` to write the simulation script waveform viewing command. The implicit argument is the top-level module or entity name.
Name: `'HDLSimTerm'` Default: `'run -all\n'`	Format string passed to `fprintf` to write the `Term` portion of the simulation script
Name: `'HDLSynthInit'` Default: `'project -new %s.prj\n'`	Format string passed to `fprintf` to write the `Init` section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name.
Name: `'HDLSynthCmd'` Default: `'add_file %s\n'`	Format string passed to `fprintf` to write the `Cmd` section of the synthesis script. The argument is the file name of the entity or module.

Name: `'HDLSynthTerm'`
Default:
`['set_option -technology VIRTEX2\n',...`
`'set_option -part XC2V500\n',...`
`'set_option -synthesis_onoff_pragma 0\n',...`
`'set_option -frequency auto\n',...`
`'project -run synthesis\n']`
	Format string passed to `fprintf` to write the `Term` section of the synthesis script.

Example

The following example specifies a ModelSim command for the `Init` phase of a compilation script for VHDL code generated from the subsystem `system`.

```
makehdl(system, 'HDLCompileInit', 'vlib mydesignlib\n')
```

The resultant script, `system_compile.do`, is shown below.

```
vlib mydesignlib
vcom  system.vhd
```

Properties — By Category

Language Selection Properties

`TargetLanguage`	Specify HDL language to use for generated code

File Naming and Location Properties

`HDLMapPostfix`	Specify postfix string appended to file name for generated mapping file
`TargetDirectory`	Identify directory into which generated output files are written
`VerilogFileExtension`	Specify file type extension for generated Verilog files
`VHDLFileExtension`	Specify file type extension for generated VHDL files

Reset Properties

`ResetAssertedLevel`	Specify asserted (active) level of reset input signal
`ResetType`	Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers
`ResetValue`	Specify constant value to which test bench forces reset input signals

Header Comment and General Naming Properties

`ClockProcessPostfix`	Specify string to append to HDL clock process names
`EntityConflictPostfix`	Specify string to append to duplicate VHDL entity or Verilog module names
`InstancePrefix`	Specify string prefixed to generated component instance names
`PackagePostfix`	Specify string to append to specified model or subsystem name to form name of VHDL package file
`ReservedWordPostfix`	Specify string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words
`SplitArchFilePostfix`	Specify string to append to specified name to form name of file containing model's VHDL architecture
`SplitEntityArch`	Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files
`SplitEntityFilePostfix`	Specify string to append to specified model name to form name of generated VHDL entity file
`VectorPrefix`	Specify string prefixed to vector names in generated code

Script Generation Properties

`EDAScriptGeneration`	Enable or disable generation of script files for third-party tools
`HDLCompileFilePostfix`	Specify postfix string appended to file name for generated ModelSim compilation scripts
`HDLCompileInit`	Specify string written to initialization section of compilation script
`HDLCompileTerm`	Specify string written to termination section of compilation script
`HDLCompileVerilogCmd`	Specify command string written to compilation script for Verilog files
`HDLCompileVHDLCmd`	Specify command string written to compilation script for VHDL files
`HDLSimCmd`	Specify simulation command written to simulation script
`HDLSimFilePostfix`	Specify postfix string appended to file name for generated ModelSim test bench simulation scripts
`HDLSimInit`	Specify string written to initialization section of simulation script
`HDLSimTerm`	Specify string written to termination section of simulation script
`HDLSynthCmd`	Specify command written to synthesis script
`HDLSynthFilePostfix`	Specify postfix string appended to file name for generated Synplify synthesis scripts

`HDLSynthInit`	Specify string written to initialization section of synthesis script
`HDLSynthTerm`	Specify string written to termination section of synthesis script

Port Properties

`ClockEnableInputPort`	Name HDL port for model's clock enable input signals
`ClockEnableOutputPort`	Specify name of clock enable output port
`EnablePrefix`	Specify base name string for internal clock enables in generated code
`InputType`	Specify HDL data type for model's input ports
`OutputType`	Specify HDL data type for model's output ports
`ResetInputPort`	Name HDL port for model's reset input

Advanced Coding Properties

`BlockGenerateLabel`	Specify string to append to block labels used for HDL `GENERATE` statements
`CastBeforeSum`	Enable or disable type casting of input values for addition and subtraction operations before execution of operation

`CheckHDL`	Check model or subsystem for HDL code generation compatibility
`HDLControlfiles`	Attach code generation control file to Simulink model
`InlineConfigurations`	Specify whether generated VHDL code includes inline configurations
`InstanceGenerateLabel`	Specify string to append to instance section labels in VHDL `GENERATE` statements
`LoopUnrolling`	Specify whether VHDL `FOR` and `GENERATE` loops are unrolled and omitted from generated VHDL code
`OutputGenerateLabel`	Specify string that labels output assignment block for VHDL `GENERATE` statements
`SafeZeroConcat`	Specify syntax for concatenated zeros in generated VHDL code
`UseAggregatesForConst`	Specify whether all constants are represented by aggregates, including constants that are less than 32 bits
`UserComment`	Specify comment line in header of generated model and test bench files
`UseRisingEdge`	Specify VHDL coding style used to check for rising edges when operating on registers
`UseVerilogTimescale`	Use compiler `` `timescale `` directives in generated Verilog code
`Verbosity`	Specify level of detail for messages displayed during code generation

Test Bench Properties

`ClockHighTime`	Specify period, in nanoseconds, during which test bench drives clock input signals high (1)
`ClockInputPort`	Name HDL port for model's clock input signals
`ClockLowTime`	Specify period, in nanoseconds, during which test bench drives clock input signals low (0)
`ForceClock`	Specify whether test bench forces clock input signals
`ForceClockEnable`	Specify whether test bench forces clock enable input signals
`ForceReset`	Specify whether test bench forces reset input signals
`HoldTime`	Specify hold time for input signals and forced reset input signals
`SimulatorFlags`	Specify simulator flags to apply to your generated test bench
`TestBenchPostFix`	Specify suffix to test bench name
`TestBenchReferencePostFix`	Specify string appended to names of reference signals generated in test bench code

Generated Model Properties

`CodeGenerationOutput`	Control production of generated code and display of generated model
`Generatedmodelname`	Specify name of generated model

`Generatedmodelnameprefix`	Specify prefix to name of generated model
`Highlightancestors`	Highlight ancestors of blocks in generated model that differ from original model
`Highlightcolor`	Specify color for highlighted blocks in generated model

11

Properties — Alphabetical List

BlockGenerateLabel

Purpose

Specify string to append to block labels used for HDL `GENERATE` statements

Settings

`'string'`

Specify a postfix string to append to block labels used for HDL `GENERATE` statements. The default string is `_gen`.

See Also

`InstanceGenerateLabel`, `OutputGenerateLabel`

Purpose

Enable or disable type casting of input values for addition and subtraction operations before execution of operation

Settings

`'on'`(default)

Typecast input values in addition and subtraction operations to the result type before operating on the values.

`'off'`

Preserve the types of input values during addition and subtraction operations and then convert the result to the result type.

See Also

`InlineConfigurations`, `LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`, `UseVerilogTimescale`

CheckHDL

Purpose

Check model or subsystem for HDL code generation compatibility

Settings

`'on'`

Check the model or subsystem for HDL compatibility before generating code, and report any problems encountered. This is equivalent to executing the `checkhdl` function before calling `makehdl`.

`'off'` (default)

Do not check the model or subsystem for HDL compatibility before generating code.

See Also

`checkhdl`, `makehdl`

Purpose

Name HDL port for model's clock enable input signals

Settings

`'string'`

The default name for the model's clock enable input port is `clk_enable`.

If you override the default with (for example) the string `'filter_clock_enable'` for the generating subsystem `filter_subsys`, the generated entity declaration might look as follows:

```
ENTITY filter_subsys IS
   PORT( clk                :  IN  std_logic;
         filter_clock_enable :  IN  std_logic;
         reset              :  IN  std_logic;
         filter_subsys_in   :  IN  std_logic_vector (15 DOWNTO 0);
         filter_subsys_out  :  OUT std_logic_vector (15 DOWNTO 0);
         );
END filter_subsys;
```

If you specify a string that is a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See `ReservedWordPostfix` for more information.

Usage Notes

The clock enable signal is asserted active high (1). Thus, the input value must be high for the generated entity's registers to be updated.

See Also

`ClockInputPort`, `InputType`, `OutputType`, `ResetInputPort`

ClockEnableOutputPort

Purpose

Specify name of clock enable output port

Settings

`'string'`

The default name for the generated clock enable output port is `ce_out`.

A clock enable output is generated when the design requires one.

Purpose

Specify period, in nanoseconds, during which test bench drives clock input signals high (1)

Settings

`ns`

The default is `5`.

The `ClockHighTime` and `ClockLowTime` properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Usage Notes

Simulink HDL Coder ignores this property if `ForceClock` is set to `'off'`.

See Also

`ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockInputPort

Purpose

Name HDL port for model's clock input signals

Settings

`'string'`

The default clock input port name is `clk`.

If you override the default with (for example) the string `'filter_clock'` for the generated entity `my_filter`, the generated entity declaration might look as follows:

```
ENTITY my_filter IS
   PORT( filter_clock   :  IN  std_logic;
         clk_enable     :  IN  std_logic;
         reset          :  IN  std_logic;
         my_filter_in   :  IN  std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         my_filter_out  :  OUT std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         );
END my_filter;
```

If you specify a string that is a VHDL reserved word, the code generator appends a reserved word postfix string to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See `ReservedWordPostfix` for more information.

See Also

`ClockEnableInputPort`, `InputType`, `OutputType`

Purpose

Specify period, in nanoseconds, during which test bench drives clock input signals low (0)

Settings

The default is `5` ns.

The `ClockHighTime` and `ClockLowTime` properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Usage Notes

The Simulink HDL Coder ignores this property if `ForceClock` is set to `'off'`.

See Also

`ClockHighTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockProcessPostfix

Purpose

Specify string to append to HDL clock process names

Settings

`'string'`

The default postfix is `_process`.

Simulink HDL Coder uses process blocks for register operations. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the coder derives the label `delay_pipeline_process` in the following block declaration from the register name `delay_pipeline` and the default postfix string `_process`:

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  .
  .
  .
```

See Also

`PackagePostfix`, `ReservedWordPostfix`

Purpose

Control production of generated code and display of generated model

Settings

`'string'`

`'GenerateHDLCode'` (default)

Generate code but do not display the generated model.

`'GenerateHDLCodeAndDisplayGeneratedModel'`

Generate both code and model, and display model when completed.

`'DisplayGeneratedModelOnly'`

Create and display generated model, but do not proceed to code generation.

See Also

“Defaults and Options for Generated Models” on page 5-8

EDAScriptGeneration

Purpose

Enable or disable generation of script files for third-party tools

Settings

`'on'` (default)

Enable generation of script files.

`'off'`

Disable generation of script files.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify base name string for internal clock enables in generated code

Settings

`'string'`

Specify the string used as the base name for internal clock enables and other flow control signals in generated code. The default string is `'enb'`.

Usage Notes

Where only a single clock enable is generated, `EnablePrefix` specifies the signal name for the internal clock enable signal.

In some cases multiple clock enables are generated (for example, when a cascade block implementation for certain blocks is specified). In such cases, `EnablePrefix` specifies a base signal name for the first clock enable that is generated. For other clock enable signals, numeric tags are appended to `EnablePrefix` to form unique signal names. For example, the following code fragment illustrates two clock enables that were generated when `EnablePrefix` was set to `'test_clk_enable'` :

```
COMPONENT Timing_Controller
    PORT( clk                   :   IN    std_logic;
          reset                 :   IN    std_logic;
          clk_enable            :   IN    std_logic;
          test_clk_enable       :   OUT   std_logic;
          test_clk_enable_5_1_0 :   OUT   std_logic
          );
  END COMPONENT;
```

EntityConflictPostfix

Purpose

Specify string to append to duplicate VHDL entity or Verilog module names

Settings

`'string'`

The specified postfix resolves duplicate VHDL entity or Verilog module names. The default string is `_entity`.

For example, if Simulink HDL Coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_entity`.

See Also

`PackagePostfix`, `ReservedWordPostfix`

Purpose

Specify whether test bench forces clock input signals

Settings

`'on'` (default)

Specify that the test bench forces the clock input signals. When this option is set, the clock high and low time settings control the clock waveform.

`'off'`

Specify that a user-defined external source forces the clock input signals.

See Also

`ClockLowTime`, `ClockHighTime`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ForceClockEnable

Purpose

Specify whether test bench forces clock enable input signals

Settings

`'on'` (default)

Specify that the test bench forces the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value.

`'off'`

Specify that a user-defined external source forces the clock enable input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `HoldTime`

Purpose

Specify whether test bench forces reset input signals

Settings

`'on'` (default)

Specify that the test bench forces the reset input signals. If you enable this option, you can also specify a hold time to control the timing of a reset.

`'off'`

Specify that a user-defined external source forces the reset input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `HoldTime`

Generatedmodelname

Purpose

Specify name of generated model

Settings

`'string'`

By default, the name of a generated model is the same as that of the original model. Assign a string value to `Generatemodelname` to override the default.

See Also

"Defaults and Options for Generated Models" on page 5-8

Purpose

Specify prefix to name of generated model

Settings

`'string'`

The default prefix is `'gm_'`.

See Also

“Defaults and Options for Generated Models” on page 5-8

HDLCompileInit

Purpose

Specify string written to initialization section of compilation script

Settings

`'string'`

The default string is `'vlib work\n'`.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify string written to termination section of compilation script

Settings

`'string'`

The default is the null string (`''`).

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLCompileFilePostfix

Purpose

Specify postfix string appended to file name for generated ModelSim compilation scripts

Settings

`'string'`

The default postfix is `_compile.do`.

For example, if the name of the device under test or test bench is `my_design`, Simulink HDL Coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

Purpose

Specify command string written to compilation script for Verilog files

Settings

`'string'`

The default string is `'vlog %s %s\n'`.

The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLCompileVHDLCmd

Purpose

Specify command string written to compilation script for VHDL files

Settings

`'string'`

The default string is `'vcom %s %s\n'`.

The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Attach code generation control file to Simulink model

Settings

`{'string'}`

Pass in a cell array containing a string that specifies a control file to be attached to the current model. Defaults are

- File name extension: `.m`
- Path: the control file is assumed to be on the MATLAB path or in the current working directory. If the file is elsewhere, enter a full path name.

Note The current release supports specification of a single control file.

Usage Notes

To clear the property (so that no control file is invoked during code generation), pass in a cell array containing the null string, as in the following example:

```
makehdl(gcb,'HDLControlFiles',{''});
```

See Also

For a detailed description of the structure and use of control files, see Chapter 4, “Code Generation Control Files”.

HDLMapPostfix

Purpose

Specify postfix string appended to file name for generated mapping file

Settings

`'string'`

The default postfix is `'_map.txt'`.

For example, if the name of the device under test is `my_design`, Simulink HDL Coder adds the postfix `_map.txt` to form the name `my_design_map.txt`.

Purpose

Specify simulation command written to simulation script

Settings

`'string'`

The default string is `'vsim work.%s\n'`.

The implicit argument is the top-level module or entity name.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify string written to initialization section of simulation script

Settings

`'string'`

The default string is

```
['onbreak resume\n',...
'onerror resume\n']
```

See Also

Chapter 9, "Generating Scripts for HDL Simulators and Synthesis Tools"

Purpose

Specify postfix string appended to file name for generated ModelSim test bench simulation scripts

Settings

`'string'`

The default postfix is `_sim.do`.

For example, if the name of your test bench file is `my_design`, Simulink HDL Coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

HDLSimTerm

Purpose

Specify string written to termination section of simulation script

Settings

`'string'`

The default string is `'run -all\n'`.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify command written to synthesis script

Settings

`'string'`

The default string is `'add_file %s\n'`.

The implicit argument is the file name of the entity or module.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

HDLSynthInit

Purpose

Specify string written to initialization section of synthesis script

Settings

`'string'`

The default string is `'project -new %s.prj\n'`, which is a synthesis project creation command.

The implicit argument is the top-level module or entity name.

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Specify postfix string appended to file name for generated Synplify synthesis scripts

Settings

`'string'`

The default postfix is `_synplify.tcl`.

For example, if the name of the device under test is `my_design`, Simulink HDL Coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

HDLSynthTerm

Purpose

Specify string written to termination section of synthesis script

Settings

`'string'`

The default string is

```
['set_option -technology VIRTEX2\n',...
'set_option -part XC2V500\n',...
'set_option -synthesis_onoff_pragma 0\n',...
'set_option -frequency auto\n',...
'project -run synthesis\n']
```

See Also

Chapter 9, “Generating Scripts for HDL Simulators and Synthesis Tools”

Purpose

Highlight ancestors of blocks in generated model that differ from original model

Settings

`'on'` (default)

Highlight blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy.

`'off'`

Highlight only the blocks in a generated model that differ from the original model without highlighting their ancestor (parent) blocks in the model hierarchy.

See Also

“Defaults and Options for Generated Models” on page 5-8

Highlightcolor

Purpose

Specify color for highlighted blocks in generated model

Settings

`'string'`

The default color specification is `'cyan'`.

Specify the color as one of the following color string values:

- `cyan`
- `yellow`
- `magenta`
- `red`
- `green`
- `blue`
- `white`
- `black`

See Also

"Defaults and Options for Generated Models" on page 5-8

Purpose

Specify hold time for input signals and forced reset input signals

Settings

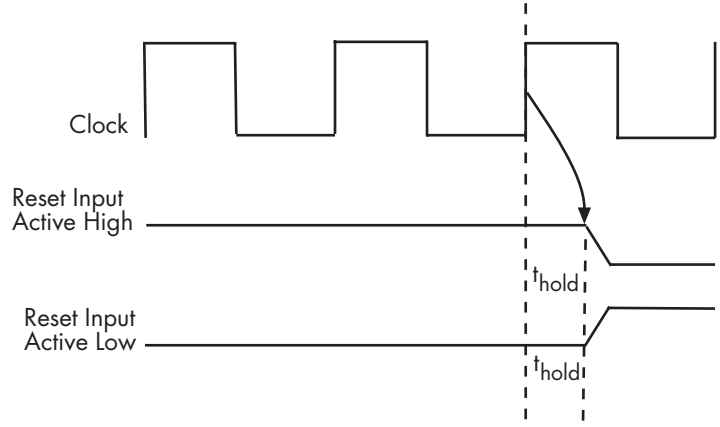
`ns`

Specify the number of nanoseconds (a positive integer) during which the model's data input signals and forced reset input signals are held past the clock rising edge. The default is `2`.

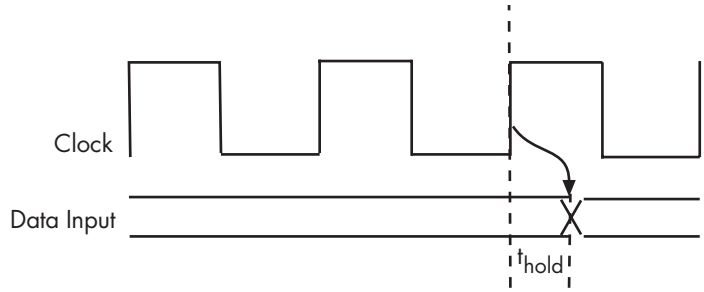
This option applies to reset input signals only if forced resets are enabled.

Usage Notes

The hold time is the amount of time that reset input signals and input data are held past the clock rising edge. The following figures show the application of a hold time (t_{hold}) for reset and data input signals when the signals are forced to active high and active low.



Hold Time for Reset Input Signals



Hold Time for Data Input Signals

Note A reset signal is always asserted for two cycles plus t_{hold}.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`

Purpose

Specify whether generated VHDL code includes inline configurations

Settings

`'on'` (default)

Include VHDL configurations in any file that instantiates a component.

`'off'`

Suppress the generation of configurations and require user-supplied external configurations. Use this setting if you are creating your own VHDL configuration files.

Usage Notes

VHDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, the Simulink HDL Coder includes configurations for a model within the generated VHDL code. If you are creating your own VHDL configuration files, you should suppress the generation of inline configurations.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`

InputType

Purpose

Specify HDL data type for model's input ports

Settings

`'std_logic_vector'`

Specifies VHDL type `STD_LOGIC_VECTOR` for the model's input ports.

`'signed/unsigned'`

Specifies VHDL type `SIGNED` or `UNSIGNED` for the model's input ports.

`'wire'` (Verilog)

If the target language is Verilog, the data type for all ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, , `OutputType`

Purpose

Specify string to append to instance section labels in VHDL `GENERATE` statements

Settings

`'string'`

Specify a postfix string to append to instance section labels in VHDL `GENERATE` statements. The default string is `_gen`.

See Also

`BlockGenerateLabel`, `OutputGenerateLabel`

InstancePrefix

Purpose

Specify string prefixed to generated component instance names

Settings

`'string'`

Specify a string to be prefixed to component instance names in generated code. The default string is `u_`.

Purpose

Specify whether VHDL `FOR` and `GENERATE` loops are unrolled and omitted from generated VHDL code

Settings

`'on'`

Unroll and omit `FOR` and `GENERATE` loops from the generated VHDL code.

In Verilog code, loops are always unrolled.

If you are using an electronic design automation (EDA) tool that does not support `GENERATE` loops, you can enable this option to omit loops from your generated VHDL code.

`'off'` (default)

Include `FOR` and `GENERATE` loops in the generated VHDL code.

Usage Notes

The setting of this option does not affect results obtained from simulation or synthesis of generated VHDL code.

See Also

`InlineConfigurations`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`

OutputGenerateLabel

Purpose

Specify string that labels output assignment block for VHDL `GENERATE` statements

Settings

`'string'`

Specify a postfix string to append to output assignment block labels in VHDL `GENERATE` statements. The default string is `outputgen`.

See Also

`BlockGenerateLabel`, `OutputGenerateLabel`

Purpose

Specify HDL data type for model's output ports

Settings

`'std_logic_vector'` (VHDL default)

Output ports have VHDL type `STD_LOGIC_VECTOR`.

`'signed/unsigned'`

Output ports have type `SIGNED` or `UNSIGNED`.

`'wire'` (Verilog)

If the target language is Verilog, the data type for all ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, `InputType`

PackagePostfix

Purpose

Specify string to append to specified model or subsystem name to form name of VHDL package file

Settings

`'string'`

The coder applies this option only if a package file is required for the design. The default string is `_pkg`.

See Also

`ClockProcessPostfix`, `EntityConflictPostfix`, `ReservedWordPostfix`

Purpose

Specify string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words

Settings

`'string'`

The default postfix is `_rsvd`.

The reserved word postfix is applied to signals and constants that have names conflicting with VHDL or Verilog reserved words. For example, if your generating model contains a signal named `mod`, Simulink HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

See Also

`ClockProcessPostfix`, `EntityConflictPostfix`, `ReservedWordPostfix`

ResetAssertedLevel

Purpose

Specify asserted (active) level of reset input signal

Settings

`'active-high'` (default)

Specify that the reset input signal must be driven high (1) to reset registers in the model. For example, the following code fragment checks whether `reset` is active high before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
.
.
.
```

`'active-low'`

Specify that the reset input signal must be driven low (0) to reset registers in the model. For example, the following code fragment checks whether `reset` is active low before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '0' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
.
.
.
```

See Also

`ResetType`

Purpose

Name HDL port for model's reset input

Settings

`'string'`

The default name for the model's reset input port is `reset`.

If you override the default with (for example) the string `'chip_reset'` for the generating system `myfilter`, the generated entity declaration might look as follows:

```
ENTITY myfilter IS
   PORT( clk             :  IN  std_logic;
         clk_enable      :  IN  std_logic;
         chip_reset      :  IN  std_logic;
         myfilter_in     :  IN  std_logic_vector (15 DOWNTO 0);
         myfilter_out    :  OUT std_logic_vector (15 DOWNTO 0);
         );
END myfilter;
```

If you specify a string that is a VHDL reserved word, the code generator appends a reserved word postfix string to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See `ReservedWordPostfix` for more information.

Usage Notes

If the reset asserted level is set to active high, the reset input signal is asserted active high (1) and the input value must be high (1) for the entity's registers to be reset. If the reset asserted level is set to active low, the reset input signal is asserted active low (0) and the input value must be low (0) for the entity's registers to be reset.

See Also

`ClockEnableInputPort`, `InputType`, `OutputType`

Purpose

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers

Settings

`'async'` (default)

Use asynchronous reset logic. The following process block, generated by a Unit Delay block, illustrates the use of asynchronous resets. When the reset signal is asserted, the process block performs a reset, without checking for a clock event.

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
  END PROCESS Unit_Delay1_process;
```

`'sync'`

Use synchronous reset logic. Code for a synchronous reset follows. The following process block, generated by a Unit Delay block, checks for a clock event, the rising edge, before performing a reset:

```
Unit_Delay1_process : PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
      IF reset = '1' THEN
        Unit_Delay1_out1 <= (OTHERS => '0');
      ELSIF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
```

```
    END PROCESS Unit_Delay1_process;
```

See Also

ResetAssertedLevel

ResetValue

Purpose

Specify constant value to which test bench forces reset input signals

Settings

`'active high'` (default)

Specify that the test bench set the reset input signal to active high (1).

`'active low'`

Specify that the test bench set the reset input signal to active low (0).

Usage Notes

The setting for this option must match the setting of the reset asserted level specified for the test bench. Simulink HDL Coder ignores the setting of this option if forced resets are disabled.

See Also

`ForceReset`, `ResetType`, `ResetAssertedLevel`

Purpose

Specify syntax for concatenated zeros in generated VHDL code

Settings

`'on'` (default)

Use the type-safe syntax, `'0' & '0'`, for concatenated zeros. Typically, this syntax is preferred.

`'off'`

Use the syntax "`000000...`" for concatenated zeros. This syntax can be easier to read and is more compact, but it can lead to ambiguous types.

See Also

`LoopUnrolling`, `UseAggregatesForConst`, `UseRisingEdge`

SimulatorFlags

Purpose

Specify simulator flags to apply to your generated test bench

Settings

`'string'`

Specify options that are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag `-93`.

Usage Notes

The flags you specify with this option are added to the `vsim` command in generated ModelSim `.do` test bench files.

Purpose

Specify string to append to specified name to form name of file containing model's VHDL architecture

Settings

`'string'`

The default is `_arch`. This option applies only if you direct Simulink HDL Coder to place the generated VHDL entity and architecture code in separate files.

Usage Notes

The option applies only if you direct the Simulink HDL Coder to place the filter's entity and architecture in separate files.

See Also

`SplitEntityArch`, `SplitEntityFilePostfix`

SplitEntityArch

Purpose

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

`'on'`

Write the generated VHDL code to a single file.

`'off'`(default)

Write the code for the generated VHDL entity and architecture to separate files.

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch `) are appended to the base file name. You can override the default and specify your own postfix string.

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

Note This property is specific to VHDL code generation. It does not apply to Verilog code generation and should not be enabled when generating Verilog code.

See Also

`SplitArchFilePostfix`, , `SplitEntityFilePostfix`

Purpose

Specify string to append to specified model name to form name of generated VHDL entity file

Settings

`'string'`

The default is `_entity`. This option applies only if you direct Simulink HDL Coder to place the generated VHDL entity and architecture code in separate files.

See Also

`SplitArchFilePostfix`, `SplitEntityArch`

TargetDirectory

Purpose

Identify directory into which generated output files are written

Settings

`'string'`

Specify a subdirectory under the current working directory into which generated files are written. The string can specify a complete path name. The default string is `hdlsrc`.

If the target directory does not exist, Simulink HDL Coder creates it.

See Also

`VerilogFileExtension`, `VHDLFileExtension`

Purpose

Specify HDL language to use for generated code

Settings

`'VHDL'` (default)

Generate VHDL filter code.

`'verilog'`

Generate Verilog filter code.

TestBenchPostFix

Purpose

Specify suffix to test bench name

Settings

`'string'`

The default postfix is `'_tb'`.

For example, if the name of your DUT is `my_test`, Simulink HDL Coder adds the postfix `_tb` to form the name `my_test_tb`.

Purpose

Specify string appended to names of reference signals generated in test bench code

Settings

`'string'`

The default postfix is `'_ref'`.

Reference signal data is represented as arrays in the generated test bench code. The string specified by `TestBenchReferencePostFix` is appended to the generated signal names.

UseAggregatesForConst

Purpose

Specify whether all constants are represented by aggregates, including constants that are less than 32 bits

Settings

`'on'`

Specify that all constants, including constants that are less than 32 bits, be represented by aggregates. The following VHDL constant declarations show scalars less than 32 bits being declared as aggregates:

```
CONSTANT coeff1 :signed(15 DOWNTO 0) := (4 DOWNTO 2 => '0', 0 =>'0', OTHERS => ', ');
CONSTANT coeff2 :signed(15 DOWNTO 0) := (6 => '0', 4 DOWNTO 3 => '0',OTHERS => ', ');
```

`'off'` (default)

Specify that the coder represent constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. The following VHDL constant declarations are examples of declarations generated by default for values less than 32 bits:

```
CONSTANT coeff1 :signed(15 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
CONSTANT coeff2 :signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
```

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseRisingEdge`

Purpose

Specify comment line in header of generated model and test bench files

Settings

`'string'`

For example, you might use this property to add the revision control tag `$Revision:  1.1.4.7 $` to the header. The resulting header comment block for subsystem `symmetric_fir` would appear as follows:

```
-- -------------------------------------------------------------
--
-- Module: symmetric_fir
-- Simulink Path: symmetric_fixed_symmetric_fir
-- Created: 2005-03-30 16:18:09
-- Hierarchy Level: 0
-- $Revision: 1.1.4.1 $
--
-- -------------------------------------------------------------
```

UseRisingEdge

Purpose

Specify VHDL coding style used to check for rising edges when operating on registers

Settings

`'on'`

Use the VHDL `rising_edge` function to check for rising edges when operating on registers. The following code, generated from a Unit Delay block, tests `rising_edge` as shown in the following `PROCESS` block:

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
  END PROCESS Unit_Delay1_process;
```

`'off'` (default)

Check for clock events when operating on registers. The following code, generated from a Unit Delay block, checks for a clock event as shown in the `ELSIF` statement of the following `PROCESS` block:

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
```

```
    END PROCESS Unit_Delay1_process;
```

Usage Notes

The two coding styles have different simulation behavior when the clock transitions from `'X'` to `'1'`.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`

UseVerilogTimescale

Purpose

Use compiler `timescale directives in generated Verilog code

Settings

`'on'` (default)

Use compiler `timescale directives in generated Verilog code.

`'off'`

Suppress the use of compiler `timescale directives in generated Verilog code.

Usage Notes

The `timescale directive provides a way of specifying different delay values for multiple modules in a Verilog file. This setting does not affect the generated test bench.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`

Purpose

Specify string prefixed to vector names in generated code

Settings

`'string'`

Specify a string to be prefixed to vector names in generated code. The default string is `vector_of_`.

Verbosity

Purpose

Specify level of detail for messages displayed during code generation

Settings

`n`

The default for `n` is `0` (minimal messages displayed).

When `Verbosity` is set to `0`, minimal code generation progress messages are displayed in the MATLAB window. When `Verbosity` is set to `1`, more detailed progress messages are displayed.

Purpose

Specify file type extension for generated Verilog files

Settings

`'string'`

The default file type extension for generated Verilog files is `.v`.

See Also

`TargetDirectory`

VHDLFileExtension

Purpose

Specify file type extension for generated VHDL files

Settings

`'string'`

The default file type extension for generated VHDL files is `.vhd`.

See Also

`TargetDirectory`

12

Functions — Alphabetical List

checkhdl

Purpose

Check subsystem or model for compatibility with HDL code generation

Syntax

```
checkhdl
checkhdl(bdroot)
checkhdl('modelname')
checkhdl('modelname/subsys')
checkhdl(gcb)
output = checkhdl(arg)
```

Description

`checkhdl` is a utility that checks a subsystem or model for compatibility with HDL code generation. If any incompatibilities are detected (for example, use of unsupported blocks or illegal data type usage), `checkhdl` displays information on the blocks and potential problems in an HTML report.

`checkhdl` examines (by default) the current Simulink model for compatibility with HDL code generation.

`checkhdl(bdroot)` examines the current Simulink model for compatibility with HDL code generation.

`checkhdl('modelname')` examines the Simulink model explicitly specified by `'modelname'`for compatibility with HDL code generation.

`checkhdl('modelname/subsys')` examines a specified subsystem within the Simulink model specified by `'modelname'` for compatibility with HDL code generation.

`'subsys'` specifies the name of the subsystem to be checked. In the current release, `'subsys'` must be at the top (root) level of the current Simulink model; it cannot be a subsystem nested at a lower level of the model hierarchy.

`checkhdl(gcb)` examines the currently selected subsystem within the current Simulink model for compatibility with HDL code generation.

`checkhdl` generates an HTML HDL Code Generation Check Report. The report file-naming convention is *`system`*`_report.html`, where *`system`* is the name of the subsystem or model that was passed in to

`checkhdl`. The report is written to the target directory. `checkhdl` also displays the report in a browser window.

The report is in table format. Each entry in the table is hyperlinked to a block or subsystem that caused a problem. When you click the hyperlink, Simulink highlights and displays the block of interest (provided that the model referenced by the report is open).

If no errors are encountered, the report contains only a hyperlink to the subsystem or model that was checked.

Alternatively, you can also specify an output argument, using the following syntax:

```
output = checkhdl(arg)
```

where *arg* specifies a model or subsystem in any of the forms described above.

When an output argument is specified, `checkhdl` returns a `1xN` MATLAB struct array with one entry for each error, warning or message. In this case, no report is generated (see "Examples" on page 12-4).

The MathWorks strongly recommends that you use `checkhdl` to check your subsystems or models before generating HDL code.

`checkhdl` reports three levels of compatibility problems:

- *Errors*: Errors will cause `makehdl` to error out. These issues must be fixed before HDL code can be generated. A typical error would be the use of an unsupported data type. For example, the current release does not support complex numbers.
- *Warnings*: Warnings may cause problems in the generated code, but generally allow HDL code generation to continue. For example, the presence of an unsupported block in the model would raise a warning. In this case, the code generator attempts to proceed as if the block were not present in the design. This could lead to errors later in the code generation process, which would then terminate.

- *Messages*: Messages are indications that the HDL code generator may treat data types in a way that differs from what might be expected. For example, single-precision floating-point data types are automatically converted to double-precision because neither VHDL nor Verilog support single-precision data types.

Note If a model or subsystem passes `checkhdl` without errors, that does *not* imply that `makehdl` will complete successfully, since not all block parameters are verified in this release. However, if `checkhdl` reports an error, `makehdl` will not successfully complete HDL code generation.

For convenience, `checkhdl` also takes the same property-value pairs as `makehdl` and `makehdltb`.

Examples

The following example checks the subsystem `symmetric_fir` within the model `sfir_fixed` for HDL code generation compatibility. If problems are encountered, an HTML report is generated.

```
checkhdl('sfir_fixed/symmetric_fir')
```

The following example checks the subsystem `symmetric_fir_err` within the model `sfir_fixed_err` for HDL code generation compatibility. Information on problems encountered is returned in the struct `output`. The first element of `output` is then displayed.

```
output = checkhdl('sfir_fixed_err/symmetric_fir_err')
### Starting HDL Check.
...
### HDL Check Complete with 4 errors, warnings and messages.

output =

1x4 struct array with fields:
    path
    type
```

```
    message
    level


output(1)

ans =

       path: 'sfir_fixed_err/symmetric_fir_err/Product'
       type: 'block'
    message: 'Unhandled mixed double and non-double datatypes at ports of block'
      level: 'Error'
```

See Also

makehdl

hdllib

Purpose

Create a Simulink library of blocks that are supported for HDL code generation

Syntax

```
hdllib
```

Description

`hdllib` creates a library of blocks that are supported for HDL code generation. The library is named `hdlsupported.mdl`. After the library is generated, you must save it to a directory of your choice.

Note that `hdllib` loads many Simulink libraries during the creation of the `hdlsupported` library. (This will cause a license checkout.) When `hdllib` completes generation of the library, it does not unload Simulink libraries.

The `hdlsupported` library affords quick access to all supported blocks. By constructing models using blocks from this library, you can ensure block-level compatibility of your model with Simulink HDL Coder.

The set of supported blocks will change in future releases of Simulink HDL Coder. To keep the `hdlsupported.mdl` current, The MathWorks recommends that you rebuild the library and table each time you install a new release.

Purpose

Generate `forEach` calls for insertion into code generation control files

Syntax

```
hdlnewforeach
hdlnewforeach('blockpath')
hdlnewforeach({'blockpath1','blockpath2',...})
[cmd, impl] = hdlnewforeach
[cmd, impl] = hdlnewforeach('blockpath')
[cmd, impl] = hdlnewforeach({'blockpath1','blockpath2',...})
```

Description

Simulink HDL Coder provides the `hdlnewforeach` utility to help you construct `forEach` calls for use in code generation control files. Given a selection of one or more blocks from your model, `hdlnewforeach` returns the following for each selected block, as string data in the MATLAB workspace:

- A `forEach` call coded with the correct `modelscope`, `blocktype`, and default implementation arguments for the block
- (Optional) A cell array of strings enumerating the available implementations for the block, in `package.class` form

`hdlnewforeach` returns a `forEach` call for each selected block in the model. Each call is returned as a string.

`hdlnewforeach('blockpath')` returns a `forEach` call for a specified block in the model. The call is returned as a string.

The `'blockpath'` argument is a string specifying the full Simulink path to the desired block.

`hdlnewforeach({'blockpath1','blockpath2',...})` returns a `forEach` call for each specified block in the model. Each call is returned as a string.

The `{'blockpath1','blockpath2',...}` argument is a cell array of strings, each of which specifies the full Simulink path to a desired block.

`[cmd, impl] = hdlnewforeach` returns a `forEach` call for each selected block in the model to the string variable `cmd`. In addition, the

call returns a cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.

`[cmd, impl] = hdlnewforeach('blockpath')` returns a `forEach` call for a specified block in the model to the string variable `cmd`. In addition, the call returns a cell array of strings (`impl`) enumerating the available implementations for the block.

The `'blockpath'` argument is a string specifying the full Simulink path to the desired block.

`[cmd, impl] = hdlnewforeach({'blockpath1','blockpath2',...})` returns a `forEach` call for each specified block in the model to the string variable `cmd`. In addition, the call returns a cell array of cell arrays of strings (`impl`) enumerating the available implementations for the block.

The `{'blockpath1','blockpath2',...}` argument is a cell array of strings, each of which specifies the full Simulink path to a desired block.

Usage Notes

Before invoking `hdlnewforeach`, you must run `checkhdl` or `makehdl` to build in-memory information about the model. If you do not run `checkhdl` or `makehdl`, `hdlnewforeach` will display an error message indicating that you should run `checkhdl` or `makehdl`.

`hdlnewforeach` returns an empty string for blocks that do not have an HDL implementation. `hdlnewforeach` also returns an empty string for subsystems, which are a special case. Subsystems do not have a default implementation class, but special-purpose subsystems implementations are provided (see Chapter 7, “Interfacing Subsystems and Models to HDL Code”).

After invoking `hdlnewforeach`, you will generally want to insert the `forEach` calls returned by the function into a control file, and use the implementation information returned to specify a nondefault block implementation. See “Generating Selection/Action Statements with the hdlnewforeach Function” on page 4-16 for a worked example.

Examples

The following example generates `forEach` commands for two explicitly specified blocks.

```
hdlnewforeach({'sfir_fixed/symmetric_fir/Add4',...
'sfir_fixed/symmetric_fir/Product2'})

ans =

c.forEach('sfir_fixed/symmetric_fir/Add4',...
 'built-in/Sum', {},...
 'hdldefaults.SumLinearHDLEmission', {});

c.forEach('sfir_fixed/symmetric_fir/Product2',...
 'built-in/Product', {},...
 'hdldefaults.ProductLinearHDLEmission', {});
```

The following example generates a `forEach` command for an explicitly specified Sum block. The implementation information is listed after the `forEach` command.

```
[cmd,impl] = hdlnewforeach('sfir_fixed/symmetric_fir/Add4')

cmd =

c.forEach('sfir_fixed/symmetric_fir/Add4',...
 'built-in/Sum', {},...
 'hdldefaults.SumLinearHDLEmission', {});

impl =

    {3x1 cell}

>> impl{1}

ans =

    'hdldefaults.SumTreeHDLEmission'
    'hdldefaults.SumLinearHDLEmission'
    'hdldefaults.SumCascadeHDLEmission'
```

hdlsetup

Purpose

Set Simulink model parameters for HDL code generation

Syntax

```
hdlsetup
hdlsetup('model')
```

Description

`hdlsetup` changes the parameters of the current Simulink model (`bdroot`) to values that are commonly used for HDL code generation.

`hdlsetup('model')` changes the parameters of the Simulink model specified by the `'model'` argument to values that are commonly used for HDL code generation.

A model should be open in Simulink before you invoke the `hdlsetup` command.

The `hdlsetup` command uses the Simulink `set_param` function to set up models for HDL code generation quickly and consistently. The model parameters settings provided by `hdlsetup` are intended as useful defaults, but they may not be appropriate for all your applications.

To view the complete set of model parameters affected by `hdlsetup`, view `hdlsetup.m` in the MATLAB editor.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of user-settable model parameters.

Purpose

Generate HDL RTL code from Simulink model or subsystem

Syntax

```
makehdl()
makehdl(bdroot)
makehdl('modelname')
makehdl('modelname/subsys')
makehdl(gcb)
makehdl('PropertyName', PropertyValue,...)
makehdl(bdroot, 'PropertyName', PropertyValue,...)
makehdl('modelname', 'PropertyName', PropertyValue,...)
makehdl('modelname/subsys','PropertyName',PropertyValue,...)
makehdl(gcb, 'PropertyName', PropertyValue,...)
```

Description

`makehdl` generates HDL RTL code (VHDL or Verilog) from a Simulink model or subsystem. We will refer to a model or subsystem from which code is generated as the *device under test (DUT)*.

`makehdl()` generates HDL code from the current Simulink model (by default), using default values for all properties.

`makehdl(bdroot)` generates HDL code from the current Simulink model, using default values for all properties.

`makehdl('modelname')` generates HDL code from the Simulink model explicitly specified by `'modelname'`, using default values for all properties.

`makehdl('modelname/subsys')` generates HDL code from a subsystem within the Simulink model specified by `'modelname'`, using default values for all properties.

`'subsys'` specifies the name of the subsystem. In the current release, this must be a subsystem at the top (root) level of the current Simulink model; it cannot be a subsystem nested at a lower level of the model hierarchy.

`makehdl(gcb)` generates HDL code from the currently selected subsystem within the current Simulink model, using default values for all properties.

`makehdl('PropertyName', PropertyValue,...)` generates HDL code from the current Simulink model (by default), explicitly specifying one or more code generation options as property/value pairs.

`makehdl(bdroot, 'PropertyName', PropertyValue,...)` generates HDL code from the current Simulink model, explicitly specifying one or more code generation options as property/value pairs.

`makehdl('modelname', 'PropertyName', PropertyValue,...)` generates HDL code from the Simulink model explicitly specified by `'modelname'`, explicitly specifying one or more code generation options as property/value pairs.

`makehdl('modelname/subsys','PropertyName',PropertyValue,...)` generates HDL code from a subsystem within the Simulink model specified by `'modelname'`, explicitly specifying one or more code generation options as property/value pairs.

`'subsys'` specifies the name of the subsystem. In the current release, this must be a subsystem at the top (root) level of the current Simulink model; it cannot be a subsystem nested at a lower level of the model hierarchy.

`makehdl(gcb, 'PropertyName', PropertyValue,...)` generates HDL code from the currently selected subsystem within the current Simulink model, explicitly specifying one or more code generation options as property/value pairs.

Property/value pairs are passed in the form

```
'PropertyName', PropertyValue
```

These property settings determine characteristics of the generated code, such as HDL element naming and whether certain optimizations are applied. The next section, “HDL Code Generation Defaults” on page 12-13, summarizes the default actions of the code generator.

For detailed descriptions of each property and its effect on generated code, see Chapter 11, “Properties — Alphabetical List” and Chapter 10, “Properties — By Category”.

HDL Code Generation Defaults

This section summarizes the default actions of the code generator. Most defaults can be overridden by passing in appropriate property/value settings to `makehdl`. Chapter 11, "Properties — Alphabetical List" describes all `makehdl` properties in detail.

Target Language, File Packaging and Naming

- The `TargetLanguage` property determines whether VHDL or Verilog code is generated. The default is VHDL.
- `makehdl` writes generated files to `hdlsrc`, a subdirectory of the current working directory. This directory is called the *target directory*. `makehdl` creates a target directory if it does not already exist.
- `makehdl` generates separate HDL source files for the DUT and each subsystem within it. In addition, `makehdl` generates script files for HDL simulation and synthesis tools. File names derive from the Simulink names of the DUT. File names are assigned by Simulink HDL Coder and are not user-assignable. The following table summarizes file-naming conventions.

File	Name
Verilog source code	*system*`.v`, where *system* is the name of the DUT.
VHDL source code	*system*`.vhd`, where *system* is the name of the DUT.

File	Name
Timing controller code	`Timing_Controller.vhd` (VHDL) or `Timing_Controller.v` (Verilog). This file contains a module defining timing signals (clock, reset, external clock enable inputs and clock enable output) in a separate entity or module. Timing controller code is generated if required by the design; a purely combinatorial model does not generate timing controller code.
ModelSim compilation script	*system*`_compile.do`, where *system* is the name of the DUT.
Synplify synthesis script	*system*`_synplify.tcl`, where *system* is the name of the DUT.
VHDL package file	*system*`_pkg.vhd`, where *system* is the name of the DUT. A package file is generated only if the design requires a VHDL package.
Mapping file	*system*`_map.txt`, where *system* is the name of the DUT. This report file maps generated entities (or modules) to the Simulink subsystems that generated them. See "Code Tracing Using the Mapping File" on page 6-5.

Entities, Ports, and Signals

- Unique names are assigned to generated VHDL entities or Verilog modules. Entity or module names are derived from the

names of the DUT. Name conflicts are resolved by the use of a postfix string.

- HDL port names are assigned according to the following conventions:

HDL Port	Name
Input	Same as corresponding port name on the DUT (name conflicts resolved according to rules of the target language)
Output	Same as corresponding port name on the DUT (name conflicts resolved according to rules of the target language)
Clock input	`clk`
Clock enable input	`clk_enable`
Clock enable output	`ce_out`
Reset input	`reset`

- HDL port directions and data types
  - Port direction: `IN` or `OUT`, corresponding to the port on the DUT.
  - Clock, clock enable, and reset port data types: VHDL type `STD_LOGIC_VECTOR` or Verilog type `wire`.
  - Input and output port data types: VHDL type `STD_LOGIC_VECTOR` or Verilog type `wire`. Port widths are determined by Simulink.
- HDL signal names and data types:
  - HDL signals generated from named Simulink signals retain their signal names.

- For unnamed Simulink signals, HDL signal names are derived from the concatenated names of the block and port connected to the signal in the DUT: *`blockname_portname`*. Conflicting names are made unique according to VHDL or Verilog rules.
- Signal data types are determined by the data type of the corresponding Simulink signal. Each signal declaration is annotated with a comment indicating the Simulink data type.

General HDL Code Settings

- VHDL-specific defaults:
  - Generated VHDL files include both entity and architecture code.
  - VHDL configurations are placed in any file that instantiates a component.
  - VHDL code checks for rising edges via the logic `IF clock'event AND clock='1' THEN...` , when operating on registers.
  - When creating labels for VHDL `GENERATE` statements, `makehdl` appends `_gen` to section and block names. `makehdl` names output assignment block labels with the string `outputgen`.
- A type-safe representation is used for concatenated zeros: `'0' & '0'`...
- Generated code for registers uses asynchronous reset logic with an active-high (1) reset level.
- The postfix string `_process` is appended to process names.

Code Optimizations

- In general, generated HDL code produces results that are bit-true and cycle-accurate with respect to the original Simulink

model (that is, the HDL code exactly reproduces simulation results from the Simulink model).

However, note that some block implementations generate code that includes certain block-specific performance and area optimizations. These optimizations can produce numeric results or timing differences that differ from those produced by the original Simulink model (see Chapter 5, "Generating Bit-True Cycle-Accurate Models").

Examples

- The following call to `makehdl` generates Verilog code for the subsystem `symmetric_fir` within the model `sfir_fixed`.

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage', 'Verilog')
```

- The following call to `makehdl` generates VHDL code for the current model. Code is generated into the target directory `hdlsrc`, with all code generation options set to default values.

```
makehdl(bdroot)
```

- The following call to `makehdl` directs the HDL compatibility checker (see `checkhdl`) to check the subsystem `symmetric_fir` within the model `sfir_fixed` before code generation starts. If no compatibility errors are encountered, `makehdl` generates VHDL code for the subsystem `symmetric_fir`. Code is generated into the target directory `hdlsrc`, with all code generation options set to default values.

```
makehdl('sfir_fixed/symmetric_fir','CheckHDL','on')
```

See Also

`makehdltb`, `checkhdl`

Purpose

Generate HDL test bench from Simulink model.

Syntax

```
makehdltb('modelname/subsys')
makehdltb('modelname/subsys', 'PropertyName', PropertyValue,
   ...)
```

Description

`makehdltb('modelname/subsys')` generates an HDL test bench from the specified subsystem within a Simulink model, using default values for all properties. The *`modelname`*`/`*`subsys`* argument gives the Simulink path to the subsystem under test. This subsystem must be at the top (root) level of the current Simulink model. The generated test bench is designed to interface to and validate HDL code generated from *`subsys`* (or from a subsystem with a functionally identical public interface).

A typical practice is to generate HDL code for a subsystem, followed immediately by generation of a test bench to validate the same subsystem (see “Examples” on page 12-20).

Note If `makehdl` has not previously executed successfully within the current MATLAB session, `makehdltb` generates model code before generating the test bench code.

Properties passed in to `makehdl` persist after `makehdl` executes, and (unless explicitly overridden) will be passed in to subsequent `makehdltb` calls during the same MATLAB session.

`makehdltb('modelname/subsys', 'PropertyName', PropertyValue,...)` generates an HDL test bench from the specified subsystem within a Simulink model, explicitly specifying one or more code generation options as property/value pairs.

Property/value pairs are passed in the form

```
'PropertyName', PropertyValue
```

These property settings determine characteristics of the test bench code. Many of these properties are identical to those for `makehdl`, while others are specific to test bench generation (for example, options for generation of test bench stimuli). The next section, "Defaults for Test Bench Code Generation" on page 12-19, summarizes the defaults that are specific to generated test bench code.

For detailed descriptions of each property and its effect on generated code, see Chapter 11, "Properties — Alphabetical List" and Chapter 10, "Properties — By Category".

Generating Stimulus and Output Data

`makehdltb` generates test data from Simulink signals connected to inputs of the subsystem under test. Sample values for each stimulus signal are computed and stored for each time step of the simulation. The signal data is represented as arrays in the generated test bench code.

To help you validate generated HDL code, `makehdltb` also generates output data from Simulink signals connected to outputs of the subsystem under test. Like input data, sample values for each output signal are computed and stored for each time step of the simulation. The signal data is represented as arrays in the generated test bench code.

The Simulink total simulation time (set by the model's **Stop Time** parameter) determines the size of the stimulus and output data arrays. Computation of sample values can be time-consuming. Consider speeding up generation of signal data by entering a shorter **Stop Time**.

Defaults for Test Bench Code Generation

This section describes defaults that apply specifically to generation of test bench code. `makehdltb` has many properties and defaults in common with `makehdl`. See "HDL Code Generation Defaults" on page 12-13 for a summary of these common properties and defaults.

File Packaging and Naming

`makehdltb` generates an HDL source file containing test bench code and arrays of stimulus and output data. In addition, `makehdltb` generates script files that let you execute a ModelSim

simulation of the test bench and the HDL entity under test. Generated test bench file names (like `makehdl` generated file names) are based on the name of the DUT. The following table summarizes the default test bench file-naming conventions.

File	Name
Verilog test bench	*system*`_tb.v`, where *system* is the name of the system under test
VHDL test bench	*system*`_tb.vhd`, where *system* is the name of the system under test
ModelSim compilation script	*system*`_tb_compile.do`, where *system* is the name of the DUT
ModelSim simulation script	*system*`_tb_sim.do`, where *system* is the name of the DUT

Other Test Bench Settings

- The test bench forces clock, clock enable, and reset input signals.
- The test bench forces clock enable and reset input to active high (1).
- The clock input signal is driven high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- The test bench forces reset signals.
- The test bench applies a hold time of 2 nanoseconds to reset and data input signals.

Examples

In the following example, `makehdl` generates VHDL code for the subsystem `symmetric_fir`. After Simulink HDL Coder indicates successful completion of code generation, `makehdltb` generates a VHDL test bench for the same subsystem.

```
makehdl('sfir_fixed/symmetric_fir')
### Applying HDL Code Generation Control Statements


### Begin VHDL Code Generation
### Working on sfir_fixed/symmetric_fir as hdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
makehdltb('sfir_fixed/symmetric_fir')
### Begin TestBench Generation
### Generating Test bench: hdlsrc\symmetric_fir_tb.vhd
### Please wait ...

### HDL TestBench Generation Complete.
```

See Also

makehdl

A

Examples

Use this list to find examples in the documentation.

Generating HDL Code Using MATLAB Commands

Generating HDL Code in the Simulink Environment

Verifying Generated HDL Code in an HDL Simulator

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Z