Simulink® HDL Coder™ Release Notes

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™ Release Notes

Contents

Summary by Version

This table provides quick access to what's new in each version. For clarification, see "Using Release Notes" on page 1.

Version (Release)	New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Latest Version V1.7 (R2010a)	Yes Details	Yes Summary	Bug Reports	Printable Release Notes: PDF Current product documentation
V1.6 (R2009b)	Yes Details	Yes Summary	None	No
V1.5 (R2009a)	Yes Details	Yes Summary	None	No
V1.4 (R2008b)	Yes Details	Yes Summary	Bug Reports	No
V1.3 (R2008a)	Yes Details	Yes Summary	Bug Reports	No
V1.2 (R2007b)	Yes Details	Yes Summary	Bug Reports	No
V1.1 (R2007a)	Yes Details	No	Bug Reports	No

Using Release Notes

Use release notes when upgrading to a newer version to learn about:

- New features

- Changes
- Potential impact on your existing files and practices

Review the release notes for other MathWorks™ products required for this product (for example, MATLAB® or Simulink®). Determine if enhancements, bugs, or compatibility considerations in other products impact you.

If you are upgrading from a software version other than the most recent one, review the current release notes and all interim versions. For example, when you upgrade from V1.0 to V1.2, review the release notes for V1.1 and V1.2.

What Is in the Release Notes

New Features and Changes

- New functionality
- Changes to existing functionality

Version Compatibility Considerations

When a new feature or change introduces a reported incompatibility between versions, the **Compatibility Considerations** subsection explains the impact.

Compatibility issues reported after the product release appear under Bug Reports at The MathWorks™ Web site. Bug fixes can sometimes result in incompatibilities, so review the fixed bugs in Bug Reports for any compatibility impact.

Fixed Bugs and Known Problems

The MathWorks offers a user-searchable Bug Reports database so you can view Bug Reports. The development team updates this database at release time and as more information becomes available. Bug Reports include provisions for any known workarounds or file replacements. Information is available for bugs existing in or fixed in Release 14SP2 or later. Information is not available for all bugs in earlier releases.

Access Bug Reports using your MathWorks Account.

About Functions and Properties Being Removed

This section lists functions or properties removed or in the process of being removed. Functions and properties typically go through several stages across multiple releases before being completely removed. This provides time for you to make adjustments to your code.

- Announcement — The Release Notes announce the planned removal, but there are no functional changes; the function runs as it did before.
- Warning — When you run the function, it displays a warning message indicating it will be removed in a future release; otherwise the function runs as it did before.
- Error — When you run the function, it produces an error. The error message indicates the function was removed and suggests a replacement function, if one is available.
- Removal — When you run the function, it fails. The error message is the standard message when MATLAB does not recognize an entry.

Functions and properties might be in a stage for one or more releases before moving to another stage. Functions and properties are listed in the Functions and Properties Being Removed section only when they enter a new stage and their behavior changes. For example, if a function displayed a warning in the previous release and errors in this release, it appears on the list. If it continues to display a warning, it does not appear on the list because there was no change between the releases.

Not all functions and properties go through all stages. For example, a function's impending removal might not be announced, but instead, the first notification might be that the function displays a warning.

The Release Notes include actions you can take to mitigate the effects of function or property removal, such as adapting your code to use a replacement function.

Version 1.7 (R2010a) Simulink HDL Coder Software

This table summarizes what's new in Version 1.7 (R2010a):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports	Printable Release Notes: PDF Current product documentation

New features and changes introduced in this version are:

Simplified Syntax for Specification of Block Implementations in Control Files

In R2010a, the coder supports a simplified syntax for specifying block implementations in a control file. The new syntax lets you specify a block implementation using simple keywords, instead of `package.class` notation. The new implementation keywords are generic, rather than block-specific. This approach lets you use the same keyword to specify implementation types such as `Tree`, `Cascade`, or `Linear` for all blocks that support such implementations. For example, the following control file specifies that the coder uses a cascade implementation for all Sum blocks and all Product blocks in the model.

```
function cfg = controlFile
cfg = hdlnewcontrol(mfilename);


cfg.forEach('*',...
  'built-in/Sum', {},...
  'Cascade', {});
cfg.forEach('*',...
  'built-in/Product', {},...
  'Cascade', {});
```

To specify the the default implementation for any block, simply use the keyword `'default'`, as in the following example.

```
function cfg = controlFile
cfg = hdlnewcontrol(mfilename);
```

```
cfg.forEach( './Subsystem/MinMax', ...
  'built-in/MinMax', {}, ...
  'default');
```

Refer to "Summary of Block Implementations" in the Simulink® HDL Coder™ documentation for a complete listing of supported blocks and their implementations.

Compatibility Considerations

In previous releases, control files specified block implementations using `package.class` syntax. For example, the following control file specifies the cascade implementation for Sum blocks, using `package.class` syntax.

```
function cfg = controlFile
 cfg = hdlnewcontrol(mfilename);


 cfg.forEach('*',...
  'built-in/Sum', {},...
  'hdldefaults.SumCascadeHDLEmission', {});
```

The coder continues to support control files that use `package.class` syntax. However, we strongly recommend that you convert existing control files to the new syntax. To convert an existing control file:

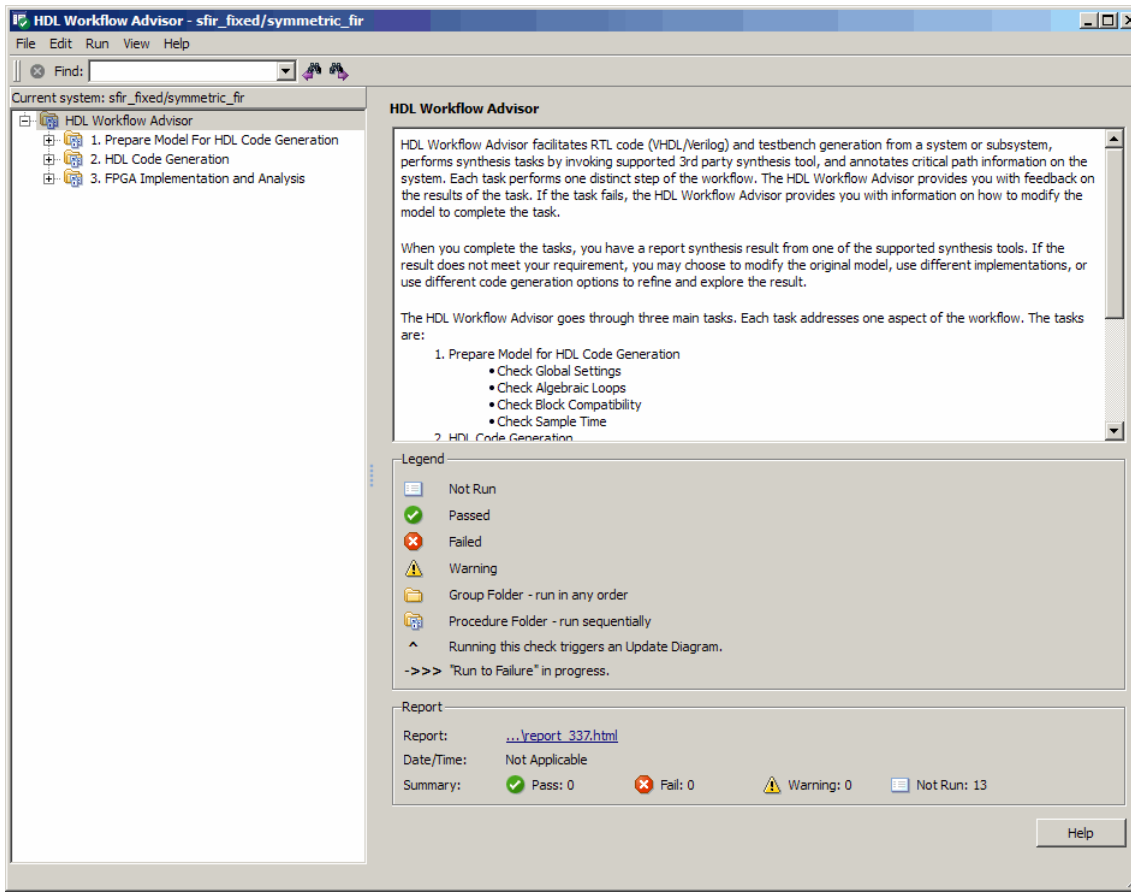
- Open a model that is linked to the control file.
- Open the Configuration Parameters dialog box and select the **HDL Coder** pane.
- Click **Generate** to generate HDL code for the model. The code generation process updates in-memory information that will be written to your updated control file.
- In the **Code generation control file** subpane, click **Save**. This overwrites the existing control file. The updated control file will use the new syntax.

HDL Workflow Advisor

The HDL Workflow Advisor is a GUI tool that supports all stages of the FPGA design process, including the following:

- Checking the Simulink model for HDL code generation compatibility
- HDL code and test bench generation
- Synthesis and timing analysis through integration with third-party synthesis tools (r2010a supports Xilinx® ISE)
- Back annotation of the Simulink model with critical path and other information obtained during synthesis.

The following figure shows the top-level HDL Workflow Advisor window.



See “Using the HDL Workflow Advisor” for further information.

Additional Simulink Blocks Supported for HDL Code Generation

The coder now supports the blocks listed in the following table for HDL code generation.

Block	Notes
simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled Resettable	
simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Resettable	
simulink/Math Operations/Trigonometric Function	See also "CORDIC Algorithm Supported for Trigonometric Functions (sin, cos, sincos)" on page 9.
Signal Processing Blockset/Signal Operations/Repeat	
Communications Blockset/Digital Baseband Modulation/PM: • PSK Modulators (BPSK,M-PSK,QPSK) • PSK Demodulators (BPSK,M-PSK,QPSK)	
Communications Blockset/Interleaving/Convolutional: • Convolutional Interleaver • Convolutional Deinterleaver	"Convolutional Interleaver and Deinterleaver Block Requirements and Restrictions"
Communications Blockset/Error Detection and Correction/Convolutional/Viterbi Decoder	"Viterbi Decoder Block Requirements and Restrictions"

"Summary of Block Implementations" in the Simulink HDL Coder documentation gives a complete listing of blocks that the coder supports for HDL code generation.

CORDIC Algorithm Supported for Trigonometric Functions (sin, cos, sincos)

The Simulink Trigonometric Function block now supports the CORDIC algorithm for the `sin`,`cos`, and `sincos` functions.

Simulink HDL Coder HDL Coder now supports HDL code generation for the Trigonometric Function block for the `sin`,`cos`, and `sincos` functions. To

generate HDL code for one these functions, select the Trigonometric Function block, you must set the **Approximation method** parameter to `CORDIC`.

See also “Trigonometric Function Block Requirements and Restrictions” in the Simulink HDL Coder documentation.

Option to Minimize Generation of Clock Enables

The new **Minimize clock enables** options lets you suppress generation of clock enable logic for single-rate designs, wherever possible. If your target device does not have registers with clock enables, you may want to consider selecting this option.

You can also use the command-line property `MinimizeClockEnable` to suppress generation of clock enable logic .

See also “Minimize clock enables” in the Simulink HDL Coder documentation.

VHDLArchitectureName Property Supports Specification of Architecture Name

The new `VHDLArchitectureName` property lets you specify the architecture name for generated HDL code. The default architecture name is `'rtl'`.

VHDLLibraryName Property Supports Specification of Target Library

The new `VHDLLibraryName` property lets you specify the name of the target library for generated HDL code. The default target library name is `'work'`.

Output Pipelining Now Supported for Subsystems

The coder now supports the `OutputPipeline` property for subsystems.

For detailed information, see “OutputPipeline” in the Simulink HDL Coder documentation.“Distributed Pipeline Insertion for Embedded MATLAB Function Blocks” in the Simulink HDL Coder documentation.

Distributed Pipelining Now Supported for Subsystems

In the previous release, the coder supported the `DistributedPipelining` property for Embedded MATLAB® Function blocks or Stateflow® charts within a subsystem.. In R2010a, the coder also supports this property for any subsystem.

For detailed information, see "Distributed Pipeline Insertion for Embedded MATLAB Function Blocks" in the Simulink HDL Coder documentation.

CSD and Factored CSD Optimizations for Constant Multiplications

You can now specify Canonic Signed Digit (CSD) and Factored Canonic Signed Digit (FCSD) techniques to optimize multiplication operations involving constants.

The `ConstMultiplierOptimization` implementation supports CSD and FCSD optimizations for the following blocks:

- Gain
- Stateflow chart
- Truth Table
- Embedded MATLAB

See also "ConstMultiplierOptimization".

Enhanced Gain Block Support

The coder now supports the following for HDL code generation for the Gain block:

- Use of `Matrix (k*u) (u vector)` mode for the Gain parameter.
- If you specify the implementation parameter `ConstMultiplierOptimization, 'auto'`) for the Gain block, the coder automatically selects CSD or FCSD implementations based on the number of required adders.

FIR Decimation Filter Supports Distributed Arithmetic Architecture

The code now supports distributed arithmetic (DA) filter implementations for the dspmlti4/FIR Decimation block. See “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” in the Simulink HDL Coder documentation for details.

Serial, Partly Serial and Cascade Serial Architectures Supported for FIR Filter Implementations

The coder now supports serial, partly serial and cascade serial architectures for the following blocks:

- dsparch4/Digital Filter (FIR structures only)
- simulink/Discrete/Discrete FIR Filter
- dspmlti4/FIR Decimation

You can specify serial architectures using the `SerialPartition` and `ReuseAccum` implementation parameters. See“Speed vs. Area Optimizations for FIR Filter Implementations” for further information.]

InstancePostfix Property Allows Specification of Extension to Postfix String

In R2010a, the coder supports the `InstancePostfix`. `InstancePostfix` lets you specify a string appended after component instance names in generated code. The default value for `InstancePostfix` is `''`(no postfix added).

Version 1.6 (R2009b) Simulink HDL Coder Software

This table summarizes what's new in Version 1.6 (R2009b):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	None	None

New features and changes introduced in this version are:

Triggered Subsystems Support for HDL Code Generation

The coder now supports HDL code generation for triggered subsystems. See "Code Generation for Enabled and Triggered Subsystems" in the Simulink HDL Coder documentation for further information.

Stateflow Events Support for HDL Code Generation

The coder now supports a single input event and unlimited output events in Stateflow charts. for further information, see "Using Input and Output Events" in the Simulink HDL Coder documentation.

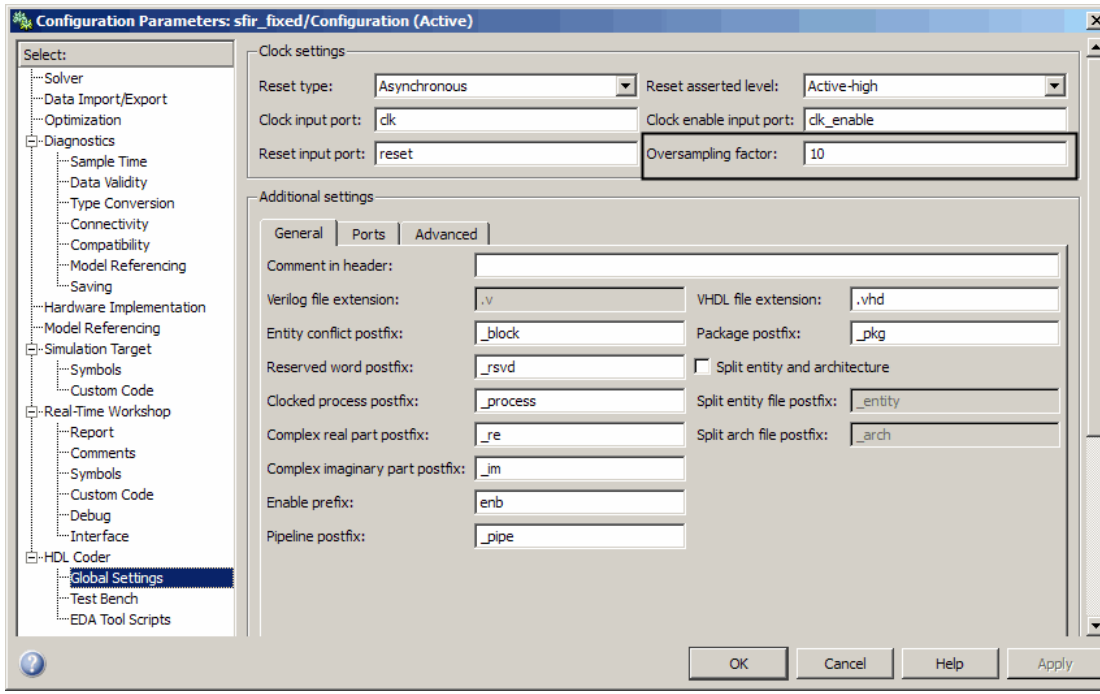
Support for Global Oversampling Clock

You can now generate global clock logic that allows you to integrate your DUT into a larger system easily, without using Upsample or Downsample blocks.

To generate global clock logic, you specify an *oversampling factor*. The oversampling factor expresses the desired rate of the global oversampling clock as a multiple of the base rate of the model. When you specify an oversampling factor, the coder generates the global oversampling clock. Then, it derives the required timing signals from the clock signal. Generation of the

global oversampling clock affects only generated HDL code. The clock does not affect the simulation behavior of your model.

You can specify the desired factor as the **Oversampling factor** option in the **Clock settings** section of the **Global Settings** pane of the Configuration Parameters dialog. The following figure shows the option. Alternatively, you can set the command-line property `'Oversampling'`.



See “Generating a Global Oversampling Clock” in the Simulink HDL Coder documentation for further information.

Test Bench GUI Reorganized

The new **Testbench generation output** section of the GUI contains three new options:

- **HDL test bench**: Selecting this option enables generation of an HDL test bench, and also enables all options in the **Configuration** section of the **Test Bench** pane.
- **Cosimulation blocks**: Selecting this option enables generation of a model containing HDL Cosimulation block for use in testing the DUT. Selecting this option also enables all options in the **Configuration** section of the **Test Bench** pane.
- **Cosimulation model for use with**: This option enables generation of a model containing an HDL Cosimulation block for use in testing with a selected cosimulation tool. Selecting this option also enables all options in the **Configuration** section of the **Test Bench** pane.

To configure test bench options and generate test bench code, select one or more of the options of the **Testbench generation output** section. If you deselect all three options of the **Testbench generation output** section, the coder disables all options in the **Configuration** section of the **Test Bench** pane.

MATLAB Editor Supports VHDL and Verilog Syntax Highlighting

The MATLAB Editor now supports syntax highlighting for VHDL and Verilog code. See “Highlighting Syntax to Help Ensure Correct Entries” in the MATLAB documentation for further information on syntax highlighting.

Hyperlinked Requirements Comments Included in HTML Code Generation Reports

The coder now renders requirements comments as hyperlinked comments within generated HTML code generation reports. See “Requirements Comments and Hyperlinks” in the Simulink HDL Coder documentation for further information.

HTML Code Generation Report from Root-Level Model Supported

In previous releases, the coder did not support generation of HTML code generation reports from the root-level model. R2009b removes this restriction.

You can now generate reports for the root-level model as well as for subsystems, blocks, Stateflow charts, or Embedded MATLAB blocks.

Generation of Simulink Model for Cosimulation of Generated HDL Code

The coder now supports generation of a Simulink model configured for:

- Simulink simulation of your design
- Cosimulation of your design with an HDL simulator

The generated model includes a behavioral model of your design and a corresponding HDL Cosimulation block, configured to cosimulate the design using EDA Simulator Link™. You can generate an HDL Cosimulation block for either of the following:

- EDA Simulator Link for use with Mentor Graphics®ModelSim®
- EDA Simulator Link for use with Cadence Incisive®

See "Generating a Simulink Model for Cosimulation with an HDL Simulator" for further information.

Additional Simulink Blocks Supported for HDL Code Generation

The coder now supports the blocks listed in the following table for HDL code generation.

Block	Implementation
hdldemolib/HDL Streaming FFT	hdldefaults.FFT
Ports & Subsystems/Trigger	hdldefaults.TriggerPort
simulink/Discrete/Discrete FIR Filter	hdldefaults.DiscreteFIRFilterHDLInstantiation
simulink/Lookup Tables/Direct Lookup Table (n-D)	hdldefaults.DirectLookupTable

Block	Implementation
simulink/Lookup Tables/Lookup Table (n-D)	hdldefaults.LookupTableND
simulink/Lookup Tables/Prelookup	hdldefaults.PreLookup

"Summary of Block Implementations" in the Simulink HDL Coder documentation gives a complete listing of blocks that the coder supports for HDL code generation.

New hdldemolib Block Supports Streaming FFT

The new hdldemolib/HDL Streaming FFT block supports a Radix-2 DIF streaming FFT algorithm.

See "HDL Streaming FFT" in the Simulink HDL Coder documentation for details.

Algebraic Loops Disallowed for HDL Code Generation

The coder now checks for algebraic loops during the compatibility checking phase of the code generation process. If `makehdl` detects an algebraic loop inside the DUT, the coder displays an error message and ends the code generation process.

Compatibility Considerations

Restructure any of your models that contain algebraic loops such that algebraic loops do not occur. It is also good practice to set the **Algebraic loop** diagnostic in the **Diagnostics** pane of the Configuration Parameters dialog box to `error`.

DUT Argument Required for checkhdl and makehdl Commands

R2009b requires that calls to the following functions must specify the device under test (DUT):

- `checkhdl`

- `makehdl`

When you call `checkhdl` or `makehdl`, specify the DUT as the initial argument to these functions, as in the following example:

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage', 'Verilog');
```

As in previous releases, you can specify the DUT in any of the following forms:

- `bdroot`: the current model.
- `'modelname'`: an explicitly specified model.
- `'modelname/subsys'`: explicitly specified path to a subsystem.
- `gcb`: the currently selected subsystem

This requirement avoids certain ambiguities that occurred in calls to `checkhdl` or `makehdl` that did not pass in an explicit DUT argument.

In R2009b, the coder displays a warning if it encounters a call to `checkhdl` or `makehdl` without the DUT argument. In future releases, the coder will generate an error if it encounters a call to either of these functions without the DUT argument.

See also the checkhdl and makehdl function reference pages in the Simulink HDL Coder documentation.

Compatibility Considerations

If your MATLAB files contain any calls to `checkhdl` or `makehdl` that do not specify the DUT, modify them to pass in the DUT as the initial argument.

AddClockEnablePort Implementation Parameter for RAM Blocks Deprecated

The `AddClockEnablePort` implementation parameter for the Dual Port RAM and Single Port RAM blocks is deprecated. The coder issues an error message if it detects a reference to `AddClockEnablePort` in a control file.

Compatibility Considerations

If you use the `AddClockEnablePort` in a control file to suppress to generation of a clock enable signal for RAM blocks:

- Remove all references to `AddClockEnablePort` from your control files.
- Use the generic RAM templates instead. The generic RAM templates do not use a clock enable signal for RAM structures. The generic RAM template implements clock enable with logic in a wrapper around the RAM. Consider the generic RAM style if
  - Your synthesis tool does not support RAM structures with a clock enable
  - Your synthesis tool cannot map generated HDL code to FPGA RAM resources.

To learn how to use generic style RAM for your design, see the new Getting Started with RAM and ROM in Simulink demo. To open the demo, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Additional Lookup Table Blocks Supported

The coder now supports the following lookup table (LUT) blocks for HDL code generation:

- simulink/Lookup Tables/Lookup Table (n-D)
- simulink/Lookup Tables/Prelookup
- simulink/Lookup Tables/Direct Lookup Table (n-D)

Expanded LUT functionality supported for these blocks includes:

- Tables of two dimensions
- Prelookup
- Interpolation
- Extrapolation

See “Using Lookup Table Blocks” in the Simulink HDL Coder documentation for details.

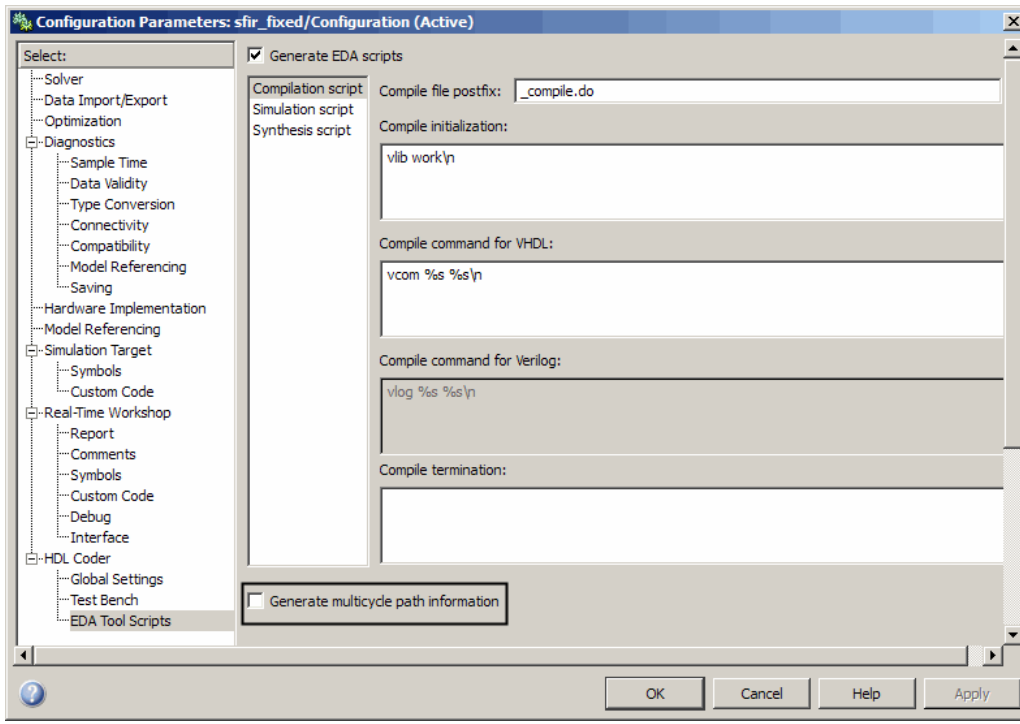
Discrete FIR Filter Supports Distributed Arithmetic Architecture

The code now supports distributed arithmetic (DA) filter implementations for the Discrete FIR Filter block. See “Distributed Arithmetic Implementation Parameters for Digital Filter Blocks” in the Simulink HDL Coder documentation for details.

Generation of Multicycle Path Constraint Information

The coder now supports generation of a text file that reports multicycle path constraint information. You can use this information with your synthesis tool.

To generate the file, select the **Generate multicycle path information** option in the **EDA Tool Scripts** pane of the Configuration Parameters dialog box. The following figure shows this option.



To generate a multicycle path constraint information file at the command line, set the `MulticyclePathInfo` property as shown in the following example.

```
makehdl(gcb,'MulticyclePathInfo', 'on');
```

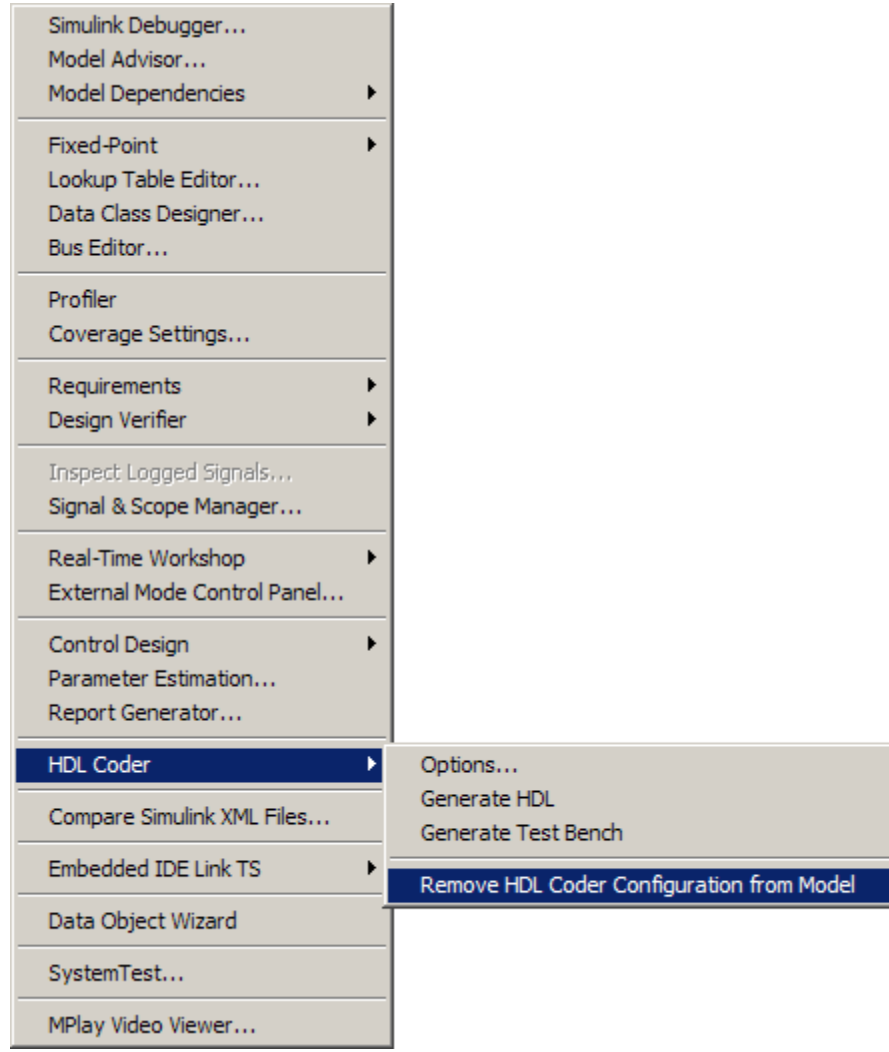
See “Generating Multicycle Path Information Files” in the Simulink HDL Coder documentation for detailed information.

Biquad Filter and Digital Filter Blocks Support Complex Input Data and Coefficients

The Biquad Filter and Digital Filter blocks now support complex input data and coefficients for all filter structures except decimators and interpolators.

Support for Adding or Removing HDL Configuration Component

The **HDL Coder** submenu of the **Tools** menu now supports addition or removal of the HDL Coder configuration component of a model. The following figure shows the **Remove HDL Configuration to Model** option.



See “Adding and Removing the HDL Configuration Component” Simulink HDL Coder documentation for more information.

Version 1.5 (R2009a) Simulink HDL Coder Software

This table summarizes what's new in Version 1.5 (R2009a):

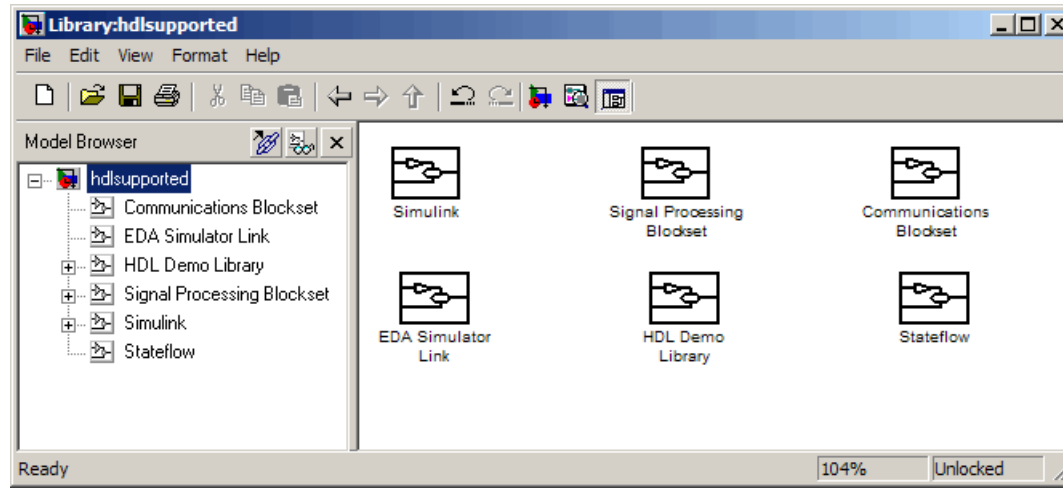
New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	None	None

New features and changes introduced in this version are:

- "Generating ROM with Lookup Table and Unit Delay Blocks" on page 35

hdlsupported Library Reorganized

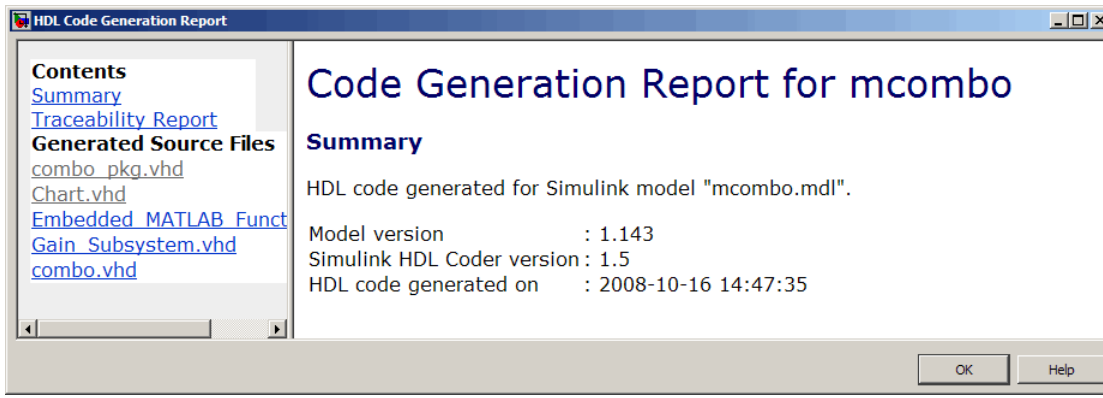
The `hdlsupported.mdl` block library has been reorganized into several sublibraries to help you locate the HDL-compatible blocks you need more easily. The following figure shows the top-level view of the `hdlsupported.mdl` library.



The set of supported blocks will change in future releases of the coder. To keep the `hdlsupported.mdl` current, you should rebuild the library each time you install a new release. See "Supported Blocks Library" in the Simulink HDL Coder documentation for further information.

HTML Code Generation Report

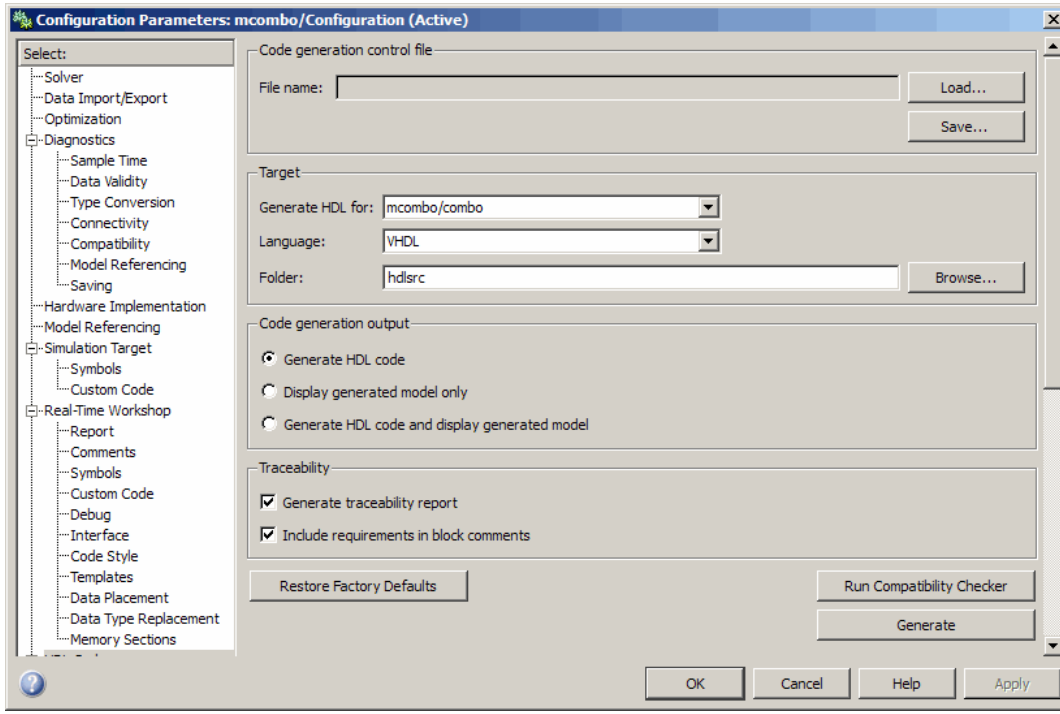
To help you navigate more easily between generated code and your source model, the coder provides a *traceability* option that lets you generate reports from either the GUI or the command line. When you enable traceability, the coder creates and displays an HTML code generation report during the code generation process. The following figure shows the top-level page of a typical report.



The report comprises several sections:

- The **Summary** section lists version and date information.
- The **Generated Source Files** table contains hyperlinks to that let you view generated HDL code in a MATLAB Web browser window. This view of the code includes hyperlinks that let you view the blocks or subsystems from which the code was generated. You can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. The report supports two types of linkage between the model and generated code:
  - Code-to-model hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click on the hyperlinks to view the relevant blocks or subsystems in a Simulink model window.
  - Model-to-code linkage lets you view the generated code for any block in the model. To highlight a block's generated code in the HTML report, right-click the block and select **HDL Coder > Navigate to Code** from the context menu.
- The **Traceability Report** allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / Embedded MATLAB Scripts**, providing a complete mapping between model elements and code.

To enable generation of the HTML code generation report, select **Generate traceability report** in the **HDL Coder** pane of the Configuration Parameters dialog box, as shown in the following figure.



See “Creating and Using a Code Generation Report” in the Simulink HDL Coder documentation for further information.

Additional Simulink Blocks Supported for HDL Code Generation

The coder now supports the blocks listed in the following table for HDL code generation.

Block	Implementation(s)
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Real World	default
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Increment Real World	default
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Decrement Store Integer	default
simulink/Additional Math & Discrete/ Additional Math: Increment - Decrement/Increment Store Integer	default
simulink/Discontinuties/Saturation Dynamic	default
simulink/Math Operations/Reciprocal Sqrt	default, SqrtFunction RecipSqrtNewton SqrtBitset SqrtNewton
Signal Routing/Go To	default
Signal Routing/From	default
dsparch4/Biquad Filter	default
Ports & Subsystems/Enable	default

See “Summary of Block Implementations” in the Simulink HDL Coder documentation for a complete listing of blocks that are currently supported for HDL code generation.

Enabled Subsystems Supported for HDL Code Generation

The code now supports code generation for enabled subsystems, provided that they are configured as described in “Code Generation for Enabled and Triggered Subsystems” in the Simulink HDL Coder documentation.

New Default HDL Implementations for Selected Blocks

The default HDL implementations for certain blocks has been changed. The following table lists these blocks, as well as their new default implementations and previous default implementations. All listed implementation classes belong to the package `hdldefaults`.

Block	Default Implementation Before R2009a	New Default Implementation
simulink/Commonly Used Blocks/Constant simulink/Commonly Used Blocks/Ground dspsrcs4/DSP Constant	ConstantHDLEmission	Constant
simulink/Commonly Used Blocks/Demux	DemuxHDLEmission	Demux
simulink/Commonly Used Blocks/Mux	MuxHDLEmission	Mux
simulink/Commonly Used Blocks/Switch	SwitchHDLEmission	SwitchRTW
simulink/Math Operations/Complex to Real-Imag	ComplexToRealImagHDLEmission	ComplexToRealImag
simulink/Math Operations/Real-Imag to Complex	RealImagtoComplexHDLEmission	RealImagtoComplex

See “Summary of Block Implementations” in the Simulink HDL Coder documentation for a complete listing of blocks that are currently supported for HDL code generation.

Compatibility Considerations

If your models use default HDL block implementations for the affected blocks, the coder now defaults to the new implementations. The new implementations

are compatible with the previous implementations and will produce identical results.

The older implementations for the listed blocks will be supported for a limited number of future releases. If your control files explicitly reference the previous default implementation for any of the affected blocks, the coder will continue to use the referenced implementation. You should consider removing or changing such references in your control files to use the new implementations.

New HDL Implementations for Selected Blocks

A number of HDL block implementations have been changed. The following table lists these blocks, as well as their new implementations and the earlier implementations that they replace. All listed implementation classes belong to the package `hdldefaults`.

Block	Implementation Before R2009a	New Implementation
simulink/Math Operations/MinMax dspstat3/Maximum dspstat3/Minimum	MinMaxCascadeHDLEmission	MinMaxCascade
simulink/Commonly Used Blocks/Sum simulink/Math Operations/Sum of Elements	SumTreeHDLEmission	SumTree
simulink/Commonly Used Blocks/Product simulink/Math Operations/Product of Elements	ProductTreeHDLEmission	ProductTree
simulink/Commonly Used Blocks/Sum simulink/Math Operations/Sum of Elements	SumCascadeHDLEmission	SumCascade
simulink/Commonly Used Blocks/Product simulink/Math Operations/Product of Elements	ProductCascadeHDLEmission	ProductCascade

See “Summary of Block Implementations” in the Simulink HDL Coder documentation for a complete listing of blocks that are currently supported for HDL code generation.

Compatibility Considerations

The new implementations are compatible with the previous implementations and will produce identical results.

The older implementations for the listed blocks will be supported for a limited number of future releases. If your control files explicitly reference the previous implementation for any of the affected blocks, the coder will continue to use the referenced implementation. You should consider removing or changing such references in your control files to use the new implementations.

Distributed Arithmetic Implementations for the Digital Filter Block

Distributed Arithmetic (DA) is a widely used technique for implementing sum-of-products computations without using multipliers. DA distributes multiply and accumulate operations across shifters, lookup tables (LUTs) and adders in such a way that conventional multipliers are not required. The coder now supports DA implementations for the following FIR structures of the Digital Filter block:

- `dfilt.dffir`
- `dfilt.dfsymfir`
- `dfilt.dfasymdir`

See “Block Implementation Parameters” in the Simulink HDL Coder documentation for further information.

Complex Data Supported for the Digital Filter Block

The coder supports complex coefficients and complex input signals for fully parallel FIR and CIC filter structures of the Digital Filter block. In many cases, you can use complex data and complex coefficients in combination. The following table shows the filter structures that support complex data and/or coefficients, and the permitted combinations.

Filter Structure	Complex Data	Complex Coefficients	Both Complex Data and Coefficients
`dfilt.dffir`	Y	Y	Y
`dfilt.dfsymfir`	Y	Y	Y
`dfilt.dfasymfir`	Y	Y	Y
`dfilt.dffirt`	Y	Y	Y
`mfilt.cicdecim`	Y	N/A	N/A
`mfilt.cicinterp`	Y	N/A	N/A
`mfilt.firdecim`	Y	Y	N
`mfilt.firinterp`	Y	Y	N

See “Blocks That Support Complex Data” for further information on how the coder supports use of complex data.

Requirements Comments Included in Generated Code

Requirements that you assign to Simulink blocks are now automatically included as comments in generated code. See the *Simulink® Verification and Validation™ User's Guide* in the Simulink HDL Coder documentation for further information on requirements comments.

Restriction on fi and fimath Rounding Modes in Embedded MATLAB Function Block Removed

In previous releases, the coder did not support the `convergent` and `round` modes for the `fi` and `fimath` functions in Embedded MATLAB Function blocks.

This restriction has been removed; the coder now supports all `fi` and `fimath` rounding modes.

See also “Generating HDL Code with the Embedded MATLAB Function Block” in the Simulink HDL Coder documentation.

Restriction on for Loop Increment in Embedded MATLAB Function Block Removed

In previous releases, the use of `for` loops with an increment other than `1` in an Embedded MATLAB Function Block was not supported for HDL code generation.

This restriction has been removed. The coder now allows use of any increment in a `for` loop in an Embedded MATLAB Function Block.

See also "Generating HDL Code with the Embedded MATLAB Function Block" in the Simulink HDL Coder documentation.

Generic RAM Template Supports RAM Without a Clock Enable Signal

The `hdldemolib` library provides three type of RAM blocks:

- Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM

These blocks (see "RAM Blocks" in the Simulink HDL Coder documentation) implement RAM structures using HDL templates that include a clock enable signal.

However, some synthesis tools do not support RAM inference with a clock enable. As an alternative, the coder now provides a generic style of HDL templates that do not use a clock enable signal for the RAM structures. The generic RAM template implements clock enable with logic in a wrapper around the RAM.

You may want to use the generic RAM style if your synthesis tool does not support RAM structures with a clock enable, and cannot map generated HDL code to FPGA RAM resources. To learn how to use generic style RAM for your design, see the new Getting Started with RAM and ROM in Simulink demo. To open the demo, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Generating ROM with Lookup Table and Unit Delay Blocks

Simulink HDL Coder does not provide a ROM block, but you can easily build one using basic Simulink blocks. The new Getting Started with RAM and ROM in Simulink demo includes an example in which a ROM is built using a Lookup Table block and a Unit Delay block. To open the demo, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Version 1.4 (R2008b) Simulink HDL Coder Software

This table summarizes what's new in Version 1.4 (R2008b):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports	No

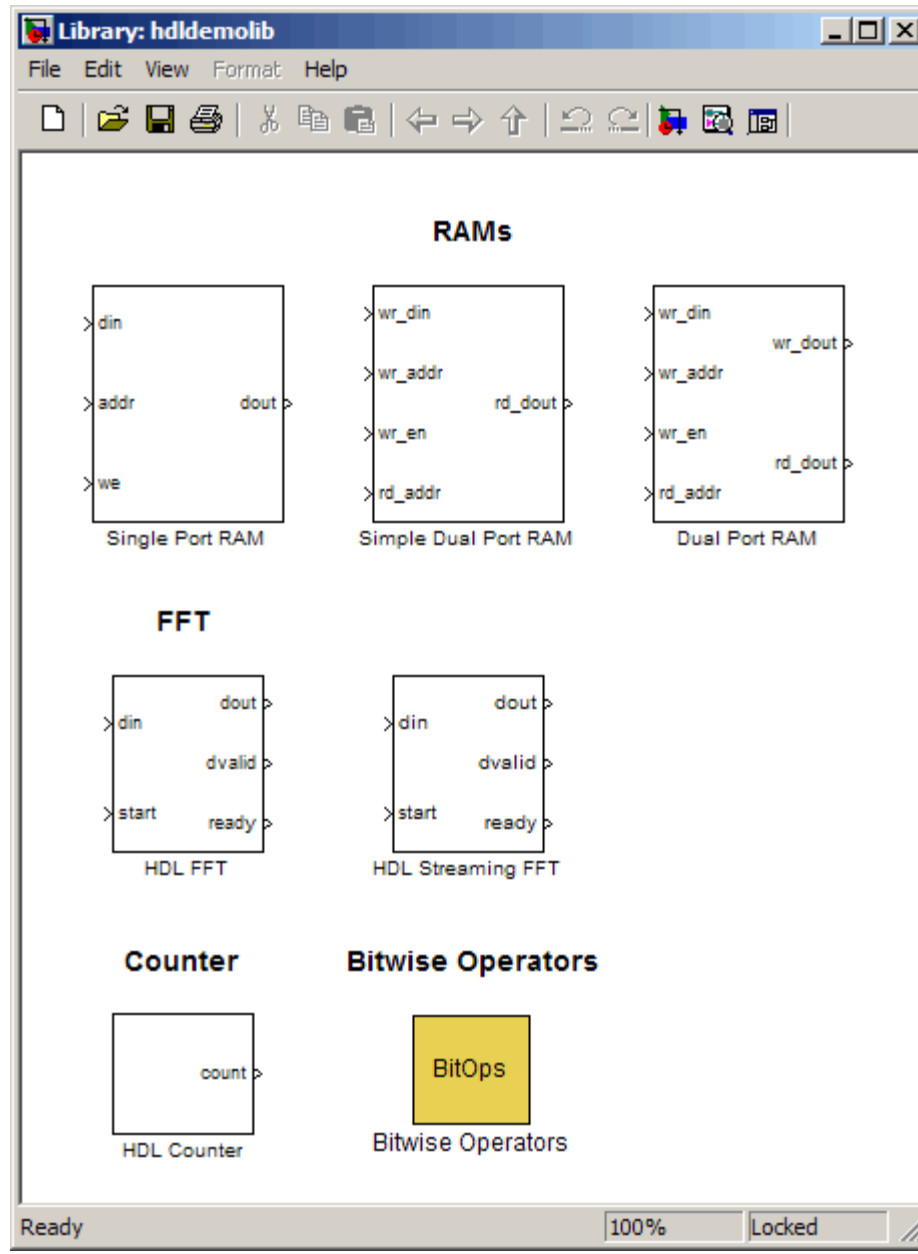
New features and changes introduced in this version are:

New hdldemolib Blocks Support FFT, HDL Counter, and Bitwise Operators

The `hdldemolib` library now includes HDL-specific block implementations supporting simulation and code generation for:

- Counter with count-limited and free-running modes (see "HDL Counter" in the Simulink HDL Coder documentation)
- Minimum resource FFT (see "HDL FFT" in the Simulink HDL Coder documentation)
- Bitwise operations, including bit slice, bit reduction, bit concatenation, bit shift, and bit rotation (see "Bitwise Operators" in the Simulink HDL Coder documentation)

The following figure shows the `hdldemolib` library window. See "The hdldemolib Block Library" in the Simulink HDL Coder documentation for more information about the library.


Library: hdldemolib
File Edit View Format Help
RAMs
din
addr
we
dout
Single Port RAM
wr_din
wr_addr
wr_en
rd_addr
rd_dout
Simple Dual Port RAM
wr_din
wr_addr
wr_en
rd_addr
wr_dout
rd_dout
Dual Port RAM
FFT
din
start
dout
dvalid
ready
HDL FFT
din
start
dout
dvalid
ready
HDL Streaming FFT
Counter
count
HDL Counter
Bitwise Operators
BitOps
Bitwise Operators
Ready
100%
Locked

Additional Simulink Blocks Supported for HDL Code Generation

The coder now supports the following blocks for HDL code generation:

- Signal Processing Blockset/Multirate Filters/CIC Interpolation
- Signal Processing Blockset/Multirate Filters/FIR Interpolation

  (See the demo "Digital Down Converter for HDL Code Generation" for an example of the use of this block.)
- Signal Processing Blockset/Filtering /Adaptive Filters/LMS Filter

  (See the demo "Adaptive Noise Canceler with LMS Filter" for an example of the use of this block.)
- simulink/Logic and Bit Operations/Extract Bits
- simulink/Math Operations/Math Function (now supports `hermitian`, and `transpose` functions for HDL code generation)
- simulink/Model-Wide Utilities/DocBlock
- Stateflow Truth Table

In addition, several HDL-specific block implementations have been added to the `hdldemolib` library. See "New hdldemolib Blocks Support FFT, HDL Counter, and Bitwise Operators" on page 37.

See "Summary of Block Implementations" in the Simulink HDL Coder documentation for a complete listing of blocks that are currently supported for HDL code generation.

Complex Signals Supported for Additional Blocks

In the previous release, the coder introduced support for use of complex signals with a limited set of blocks. In R2008b, the coder supports complex signals for these additional blocks:

- dspadpt3/LMS Filter
- dspsigattribs/Frame Conversion
- dspsigops/Delay (`DSPDelayHDLEmission` implementation)

- hdldemolib/Dual Port RAM
- hdldemolib/Simple Dual Port RAM
- hdldemolib/Single Port RAM
- hdldemolib/HDL FFT
- simulink/Commonly Used Blocks/Relational Operator (~= and == operators only)
- simulink/Discrete/Memory
- simulink/Discrete/Zero-Order Hold
- simulink/Logic and Bit Operations/Compare To Constant
- simulink/Logic and Bit Operations/Compare To Zero
- simulink/Lookup Tables/Lookup Table (`LookupHDLInstantiation` implementation)
- simulink/Math Operations/Assignment
- simulink/Math Operations/Math Function (`hermitian`, `transpose`)
- simulink/Signal Attributes/Signal Specification

See "Blocks That Support Complex Data" in the Simulink HDL Coder documentation for a complete listing of blocks that support complex signals.

Code Annotation Support

The coder now lets you add text annotations to generated code, in the form of comments. There are two ways to add annotations to your code:

- Enter text directly on the block diagram as Simulink annotations.
- Place a DocBlock at the desired level of your model and enter text comments.

See "Annotating Generated Code with Comments and Requirements" in the Simulink HDL Coder documentation for further information.

New Constant Block Implementation Indicates Hi-Z or Unknown States

The coder now supports an implementation for the built-in/Constant block (`hdldefaults.ConstantSpecialHDLEmission`), which you can use to indicate when a constant signal is in high-impedance (`'Z'`) or unknown (`'X'`) state. The implementation provides the `{Value}` parameter to indicate the state, as follows:

- `{Value, 'Z'}`: If the signal is in a high-impedance state, the Constant block emits the character `'Z'` for each bit in the signal. For example, for a 4-bit signal, `'ZZZZ'` would be emitted.

  `{Value, 'Z'}` is the default value for this implementation.

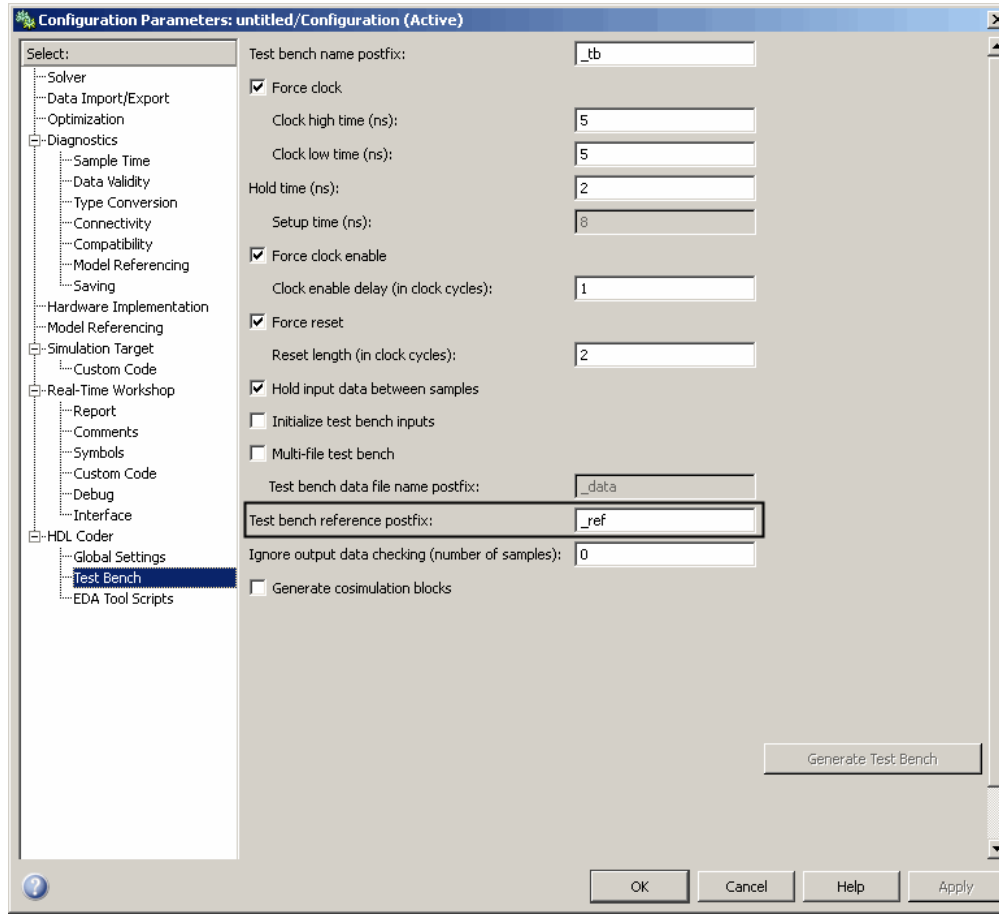
- `{Value, 'X'}`: If the signal is in an unknown state, the Constant block emits the character `'X'` for each bit in the signal. For example, for a 4-bit signal, `'XXXX'` would be emitted.

`hdldefaults.ConstantSpecialHDLEmission` does not support the `double` data type.

See also “Blocks with Multiple Implementations” in the Simulink HDL Coder documentation.

New Test Bench Reference Postfix Option

The new **Test bench reference postfix** option (shown in the following figure) lets you customize the names of reference signals generated in test bench code by specifying a string to be appended to reference signal names. The default string is`'_ref'`.



If you generate test bench code via the `makehdltb` function, use the `Testbenchreferencepostfix` property (see TestBenchReferencePostFix in the in the Simulink HDL Coder documentation) to specify the postfix string.

New Default HDL Implementations for Selected Blocks

The default HDL implementations for certain blocks has been changed. The following table lists these blocks, as well as their new default implementations and previous default implementations. All listed implementation classes belong to the package `hdldefaults`.

Block	Default Implementation Before Release R2008b	New Default Implementation
simulink/Commonly Used Blocks/Data Type Conversion	DataTypeConversionHDLEmission	DataTypeConversionRTW
simulink/Commonly Used Blocks/Product	ProductLinearHDLEmission	ProductRTW
simulink/Math Operations/Divide	ProductLinearHDLEmission	ProductRTW
simulink/Math Operations/Product of Elements	ProductLinearHDLEmission	ProductRTW
simulink/Commonly Used Blocks/Sum	SumLinearHDLEmission	SumRTW
simulink/Math Operations/Add	SumLinearHDLEmission	SumRTW
simulink/Math Operations/Sum of Elements	SumLinearHDLEmission	SumRTW
simulink/Math Operations/Subtract	SumLinearHDLEmission	SumRTW
simulink/Commonly Used Blocks/Unit Delay	UnitDelayHDLEmission	UnitDelayRTW
simulink/Math Operations/MinMax	MinMaxTreeHDLEmission	MinMaxTree
dspstat3/Maximum	MinMaxTreeHDLEmission	MinMaxTree
dspstat3/Minimum	MinMaxTreeHDLEmission	MinMaxTree

Compatibility Considerations

If your models use default HDL block implementations for the affected blocks, the coder will now default to the new implementations. The new implementations are compatible with the previous implementations and will produce identical results.

The older implementations for the listed blocks will be supported for a limited number of future releases. If your control files explicitly reference the previous default implementation for any of the affected blocks, the coder will continue to use the referenced implementation. You should consider removing or changing such references in your control files to use the new implementations.

Default Entity Conflict Postfix Changed

The default value for the **Entity conflict postfix** property (and the corresponding CLI property, `EntityConflictPostfix`) has been changed from `'_entity'` to `'_block'`.

Compatibility Considerations

If your models or scripts rely on the previous default value (`'_entity'`) for the **Entity conflict postfix** property, you will need to explicitly set the property value to `'_entity'`.

New DistributedPipelining Implementation Parameter for Embedded MATLAB Function Blocks and Stateflow Charts

In the previous release, the coder introduced automatic pipeline insertion, a special optimization for HDL code generated from Embedded MATLAB Function blocks or Stateflow charts. This optimization was enabled implicitly by specifying the `{'OutputPipeline', nStages}` parameter in a control file for these blocks.

In the current release, the new `DistributedPipelining` parameter lets you explicitly enable or disable pipeline insertion, independently from the `OutputPipeline` parameter. The control file listed in the following example specifies two pipeline registers, with `DistributedPipelining` enabled.

```
function c = pipeline_control

c = hdlnewcontrol(mfilename);

c.forEach('*',...
    'eml_lib/Embedded MATLAB Function', {},...
 'hdlstateflow.StateflowHDLInstantiation', {'OutputPipeline', 2, 'DistributedPipelining', 'on'});
```

The `DistributedPipelining` property applies only to Embedded MATLAB Function blocks or Stateflow charts within a subsystem.

For detailed information, see "Distributed Pipeline Insertion for Embedded MATLAB Function Blocks" in the Simulink HDL Coder documentation.

Compatibility Considerations

If your existing control files specified automatic pipelining implicitly using the `OutputPipeline` parameter, you should change your control files to specify automatic pipelining explicitly as in the following code excerpt:

```
c.forEach('*',...
    'eml_lib/Embedded MATLAB Function', {},...
 'hdlstateflow.StateflowHDLInstantiation', {'OutputPipeline', 2, 'DistributedPipelining', 'on'});
```

Coefficient Multiplier Optimization for Digital Filter, FIR Decimation, and FIR Interpolation Filters

The `CoeffMultipliers` implementation parameter lets you specify use of canonic signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in code generated for certain filter blocks. Specify the `CoeffMultipliers` parameter in a control file using the following syntax:

- `{'CoeffMultipliers', 'csd'}`: Use CSD techniques to replace multiplier operations with shift and add operations. CSD techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This decreases the area used by the filter while maintaining or increasing clock speed.

- `{'CoeffMultipliers', 'factored-csd'}`: Use factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the coefficients. This option lets you achieve a greater filter area reduction than CSD, at the cost of decreasing clock speed.
- `{'CoeffMultipliers', 'multipliers'}` (default): Retain multiplier operations.

The coder supports `CoeffMultipliers` for the filter block implementations shown in the following table.

Block	Implementation
dsparch4/Digital Filter	hdldefaults.DigitalFilterHDLInstantiation
dspmlti4/FIR Decimation	hdldefaults.FIRDecimationHDLInstantiation
dspmlti4/FIR Interpolation	hdldefaults.FIRInterpolationHDLInstantiation

See also “Block Implementation Parameters” in the Simulink HDL Coder documentation.

hdlnewblackbox Function Generates Black Box Control Statements

The `hdlnewblackbox` function provides a simple way to create the control file statements that are required to generate black box interfaces for one or more subsystems.

Given a selection of one or more subsystems from your model, `hdlnewblackbox` returns the following as string data in the MATLAB workspace for each selected subsystem:

- A `forEach` call coded with the correct `modelscope`, `blocktype`, and default implementation class (`SubsystemBlackBoxHDLInstantiation`) arguments for the block.
- (Optional) A cell array of strings enumerating the available implementations classes for the subsystem, in `package.class` form.
- (Optional) A cell array of cell arrays of strings enumerating the names of implementation parameters (if any) corresponding to the implementation

classes. `hdlnewblackbox` does not list data types and other details of implementation parameters.

For further information, see "Generating a Black Box Interface for a Subsystem" in the Simulink HDL Coder documentation.

hdlnewcontrolfile Function Optionally Returns Result to String

The `hdlnewcontrolfile` function (optionally) now can return control statements to a string variable.

To return control statements as text in the string variable `t`, instead of returning a control file, use the following syntax:

```
t = hdlnewcontrolfile(...)
```

See also hdlnewcontrolfile in the Simulink HDL Coder documentation.

-novopt Flag Added to Default Simulation Command in Generated Compilation Scripts

For improved operation with the ModelSim (version 6.2 and later) simulator, the default values of the `HDLSimCmd` property string (and the **Simulation Command** GUI option) now includes the `-novopt` flag, as follows:

```
'vsim -novopt work.%s\n'
```

The `-novopt` flag directs the ModelSim simulator not to perform optimizations that remove signals from the simulation view.

Compatibility Considerations

If you are using ModelSim 6.0 or an earlier version, you should set the `HDLSimCmd` property string (or the **Simulation Command** GUI option) to omit the `-novopt` option, as follows:

```
'vsim work.%s\n'
```

Version 1.3 (R2008a) Simulink HDL Coder Software

This table summarizes what's new in V1.3 (R2008a):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports	No

New features and changes introduced in this version are:

• "Limited Variable-Step Solver Support" on page 63

Complex Data Type Support

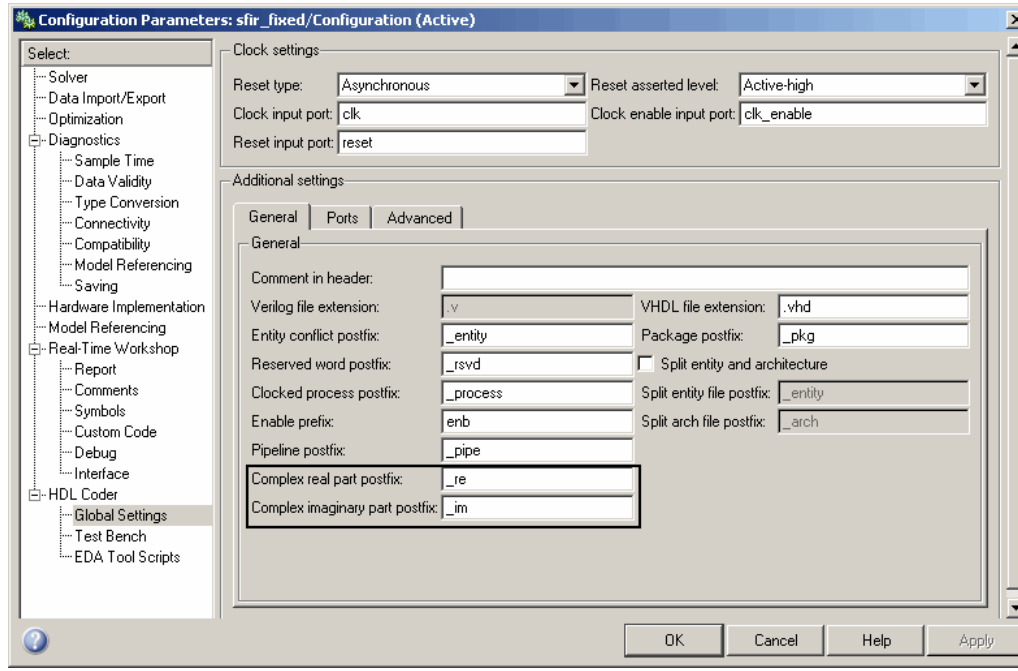
The coder now supports use of signals of complex data type.

You can use complex signals in the test bench without restriction.

In the device under test (DUT) selected for HDL code generation, support for complex signals is limited to a subset of the blocks supported by the coder. Some restrictions apply for some of these blocks. These blocks are listed in "Blocks That Support Complex Data".

New Options Supporting Complex Data Types

Two new code generation options have been added to help you customize naming conventions for the real and imaginary components of complex signals in generated HDL code. These options are available to the **Global Settings / General** pane in the **HDL Coder** pane of the Configuration Parameters dialog box, as shown in the following figure.



The **Complex real part postfix** option (and the corresponding `ComplexRealPostfix` CLI property) specifies a string to be appended to the names generated for the real part of complex signals. The default postfix is `'_re'`. See also “Complex real part postfix”.

The **Complex imaginary part postfix** option (and the corresponding `ComplexImagPostfix` CLI property) specifies a string to be appended to the names generated for the imaginary part of complex signals. The default postfix is `'_im'`. See also “Complex imaginary part postfix”.

Test Bench Enhancements

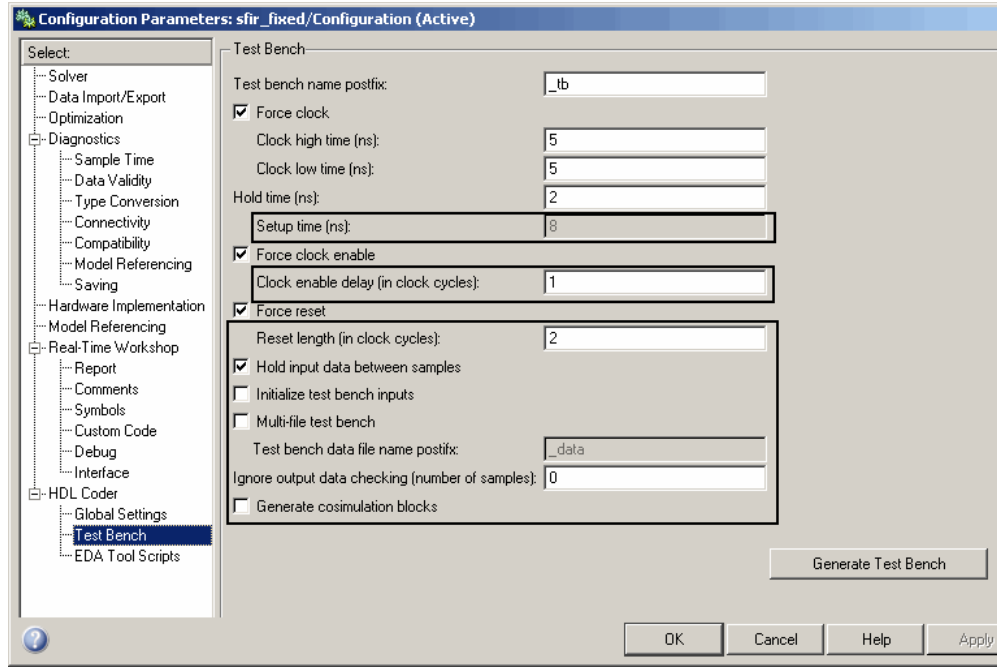
This release includes significant enhancements to test bench generation.

Test Bench Supports Complex Data Type

You can use complex signals in the test bench without restriction. Use of complex signals within the DUT is limited to a subset of supported blocks. See also “Complex Data Type Support” on page 49.

New Test Bench Options and Properties

A number of options have been added to the **HDL Coder / Test Bench** pane of the Configuration Parameters dialog box, as shown in the following figure.



Most of the new options have a corresponding command-line property. The following table lists the new options and their corresponding CLI properties, and provides hyperlinks to the relevant documentation.

GUI Option	Command-line Property
Setup time: See "Setup time (ns)"	This is a display-only field. It does not have a corresponding user-settable command-line property.
Clock enable delay (in clock cycles): See "Clock enable delay (in clock cycles)"	`TestBenchClockEnableDelay`
Reset length : See "Reset length (in clock cycles)"	`ResetLength`
Hold input data between samples: See "Hold input data between samples"	`HoldInputDataBetweenSamples`
Initialize test bench inputs: See "Initialize test bench inputs"	`InitializeTestBenchInputs`
Multi-file test bench : See "Multi-file test bench"	`MultifileTestBench`
Test bench data file name postfix : See "Test bench data file name postfix"	`TestBenchDataPostFix`
Ignore test bench data checking: See "Ignore output data checking (number of samples)"	`IgnoreDataChecking`
Generate cosimulation blocks: See "Cosimulation blocks"	`GenerateCoSimBlock`

Additional Blocks Supported for HDL Code Generation

The coder now supports the following blocks for HDL code generation:

- Communications Blockset/Comm Sources/Sequence Generators/PN Sequence Generator

  (This block requires Communications Blockset™.)

- Signal Processing Blockset/Multirate Filters/CIC Decimation
- Signal Processing Blockset/Multirate Filters/FIR Decimation
- Signal Processing Blockset/Signal Operations/NCO
- Signal Processing Blockset/Signal Processing Sources/Sine Wave
- Simulink/Discontinuities/Saturation
- Simulink/Discrete/Discrete-Time Integrator
- Simulink/Math Operations/Real-Imag to Complex
- Simulink/Math Operations/Complex to Real-Imag
- Simple Dual Port RAM (see also "Additional RAM Blocks" on page 55.)
- Single Port RAM (see also "Additional RAM Blocks" on page 55.)

See"Summary of Block Implementations" for a complete listing of blocks that are currently supported for HDL code generation.

Enhanced Pipelining Support

In the previous release, the coder introduced output pipelining support for many block implementations (see "OutputPipeline"). In this release, pipelining support has been significantly expanded and enhanced. The following sections discuss new pipelining features.

Input Pipelining

You can now specify generation of input pipeline registers for selected blocks. To do this, invoke the new block implementation parameter `{'InputPipeline', nStages}` in a control file. The parameter value (`nStages`) specifies the number of input pipeline stages (pipeline depth) in the generated code. See "InputPipeline" for further information.

Most HDL block implementations support `InputPipeline`. See "Summary of Block Implementations" for a complete list of block implemenations and their parameters.

Automatic Pipeline Insertion for Embedded MATLAB Function Block and Stateflow Chart

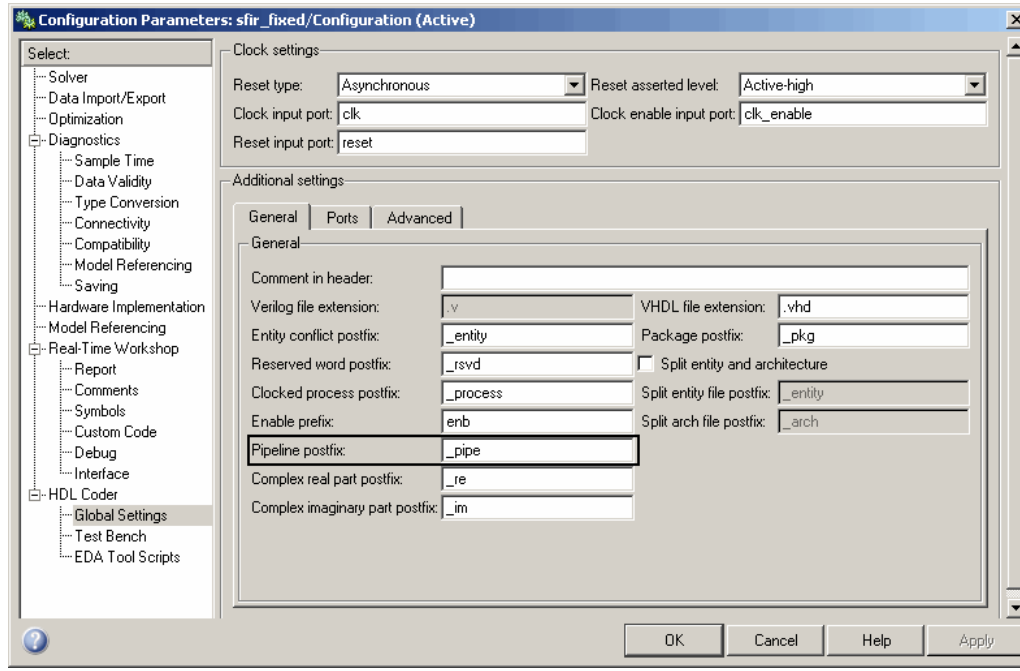
In this release, the coder introduces *automatic pipeline insertion*, a special optimization for HDL code generated from Embedded MATLAB Function blocks or Stateflow charts. Automatic pipeline insertion is performed when the `{'OutputPipeline', nStages}` parameter is specified for these blocks. When you specify `OutputPipeline`, the coder inserts internal pipeline stages into the HDL code generated for these blocks (rather than at the output of the HDL code) whenever possible. The `nStages` argument defines the number of pipeline stages to be inserted.

Automatic pipeline insertion lets you achieve higher clock rates in your HDL applications, at the cost of some latency caused by the introduction of pipeline registers.

See "Distributed Pipeline Insertion for Embedded MATLAB Function Blocks" for a detailed description of this feature.

Customizable Pipeline Register Names

When generating code for pipeline registers, the coder appends a postfix string to names of input or output pipeline registers. The default postfix string is `_pipe`. You can now customize the postfix string. To specify the postfix, use the **Pipeline postfix** option in the **Global Settings / General** pane in the **HDL Coder** pane of the Configuration Parameters dialog box (see the following figure). Alternatively, you can pass the desired postfix string in the `makehdl` property `PipelinePostfix`. See "Pipeline postfix" for an example.



Additional RAM Blocks

The coder now supports two new RAM blocks, supplementing the previously supported Dual Port RAM block:

- Simple Dual Port RAM: This block is identical to the Dual Port RAM , but does not have a data output at the write port. If data output at the write port is not required, you can achieve better RAM inferring with synthesis tools by using the Simple Dual Port RAM block rather than the Dual Port RAM block.
- Single Port RAM: This block provides data input, write address and write enable, and data output ports. The block GUI includes a **Output data during write** drop-down menu, providing options that control how the generated RAM handles data that is read into the RAM during a write operation.

See “RAM Blocks” for detailed information on RAM blocks.

Enhanced Math Function and Divide Block Support

The coder now supports a wider range of functions and algorithms for the Math Function and Divide blocks, as follows:

- The Math Function block `reciprocal` operation is now supported. Implementations using either hardware divide (HDL / operator) or iterative Newton algorithm are available.
- The Math Function block `conj` function is now supported.
- The Math Function block `sqrt` function implementations now support a choice of multiply/add, bitset shift/addition, or iterative Newton algorithms.
- The Math Operations/Divide block `reciprocal` operation now supports implementations using either hardware divide (HDL / operator) or the iterative Newton algorithm.

See “Math Function Block Implementations” and “Divide Block Implementations” for further information.

Optional Suppression of Reset Logic Generation for Selected Delay Blocks

The new `{'ResetType','None'}` block implementation parameter lets you suppress generation of reset logic for selected delay blocks. The following blocks support this parameter:

- Integer Delay
- Tapped Delay
- Unit Delay
- Unit Delay Enabled

The following control file specifies suppression of reset logic for a specific unit delay block within the subsystem `resetnone_examp/HDLSubsystem`.

```
function c = resetnone_examp

% Control file for resetnone_examp
c = hdlnewcontrol(mfilename);
c.generateHDLFor('resetnone_examp/HDLSubsystem');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Suppress reset logic for Unit Delay block

c.forEach('resetnone_examp/HDLSubsystem/Unit Delay',...
 'built-in/UnitDelay', {},...
 'hdldefaults.UnitDelayHDLEmission', {'ResetType','none'});
```

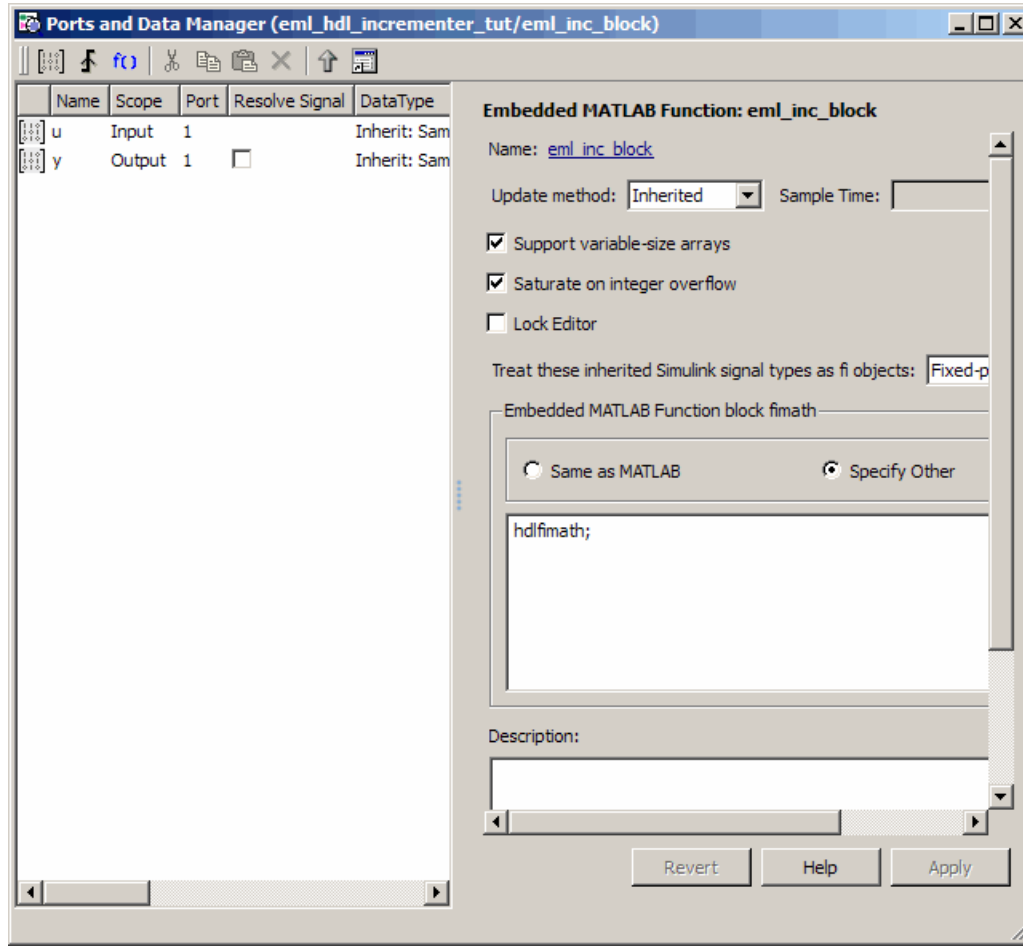
See ResetType for further information.

Enhanced Embedded MATLAB Function Block Support

HDL code generation support for the Embedded MATLAB Function block has been enhanced in Release 2008a, as discussed in the following sections.

hdlfimath Utility for Configuring Optimized FIMATH Settings

In this release, the coder provides the function `hdlfimath`, a utility that defines a FIMATH specification that is optimized for HDL code generation. When you configure an Embedded MATLAB Function Block for HDL code generation, it is strongly recommended that you replace the default **FIMATH for fixed-point signals** specification with a call to the `hdlfimath` function, as shown in the following figure.



See “Use the hdlfimath Utility for Optimized FIMATH Settings” for further information.

Support for Complex Data Type

Embedded MATLAB Function block now supports use of complex data type for HDL code generation. All operators that support complex data types can be used in a Embedded MATLAB Function block code, subject to some restrictions. See the `eml_hdl_design_patterns` library for numerous

examples showing applications of complex arithmetic in Embedded MATLAB Function blocks.

Support for Compiled External MATLAB Functions on the Embedded MATLAB Path

You can now generate HDL code from Embedded MATLAB Function blocks that include compiled external MATLAB functions. This feature lets you write reusable code that can be called from multiple Embedded MATLAB Function blocks.

Such functions must be defined in files that are on the Embedded MATLAB path, and must include the `%#eml` compilation directive. See "Adding the Compilation Directive %#eml" in the Embedded MATLAB documentation for complete details.

Support for Non-Tunable Parameter Arguments

An Embedded MATLAB function argument can be declared as a *parameter argument* by setting its **Scope** to `Parameter` in the Ports and Data Manager GUI.

Parameter arguments for Embedded MATLAB Function blocks do not appear as signal ports on the block. Parameter arguments do not take their values from signals in the Simulink model. Instead, their values come from parameters defined in a parent Simulink masked subsystem or variables defined in the MATLAB base workspace.

Only *nontunable* parameter arguments are supported for HDL code generation. If you declare parameter arguments in Embedded MATLAB function code that is intended for HDL code generation, be sure to clear the **Tunable** option for each parameter argument.

See also "Parameter Arguments in Embedded MATLAB Functions" in the Simulink documentation.

Enhanced Support for Fixed-Point Functions

Rounding Functions. The Embedded MATLAB Function block now supports the following Fixed-Point Toolbox™ rounding functions for HDL code generation:

- `ceil`
- `fix`
- `floor`
- `nearest`

See also "Supported Functions and Limitations of the Fixed-Point Embedded MATLAB Subset" in the Fixed-Point Toolboxdocumentation.

Other Functions. The Embedded MATLAB Function block now supports the following for HDL code generation:

- The `bitreplicate` function
- The `bitconcat` function now supports:
  - single-vector argument:

    ```
    bitconcat([a_vector]);
    ```

  - variable argument list:

    ```
    bitconcat(a,b,c,...);
    ```

For general information on these functions, see "Supported Functions and Limitations of the Fixed-Point Embedded MATLAB Subset" in the Fixed-Point Toolboxdocumentation.

Stateflow Chart Support Supports Complex Data Type

Stateflow charts now support the use of complex data types for HDL code generation. All operators that support complex data types can be used in a chart, without restriction.

See also “Stateflow HDL Code Generation Support”.

hdlnewcontrolfile Function Generates Control Files Automatically

The coder provides the new `hdlnewcontrolfile` utility to help you construct code generation control files. Given a selection of one or more blocks from your model, `hdlnewcontrolfile` generates a control file containing `forEach` statements and comments providing information about all supported implementations and parameters, for all selected blocks. The generated control file is automatically opened in the MATLAB Editor for further customization. See hdlnewcontrolfile for details.

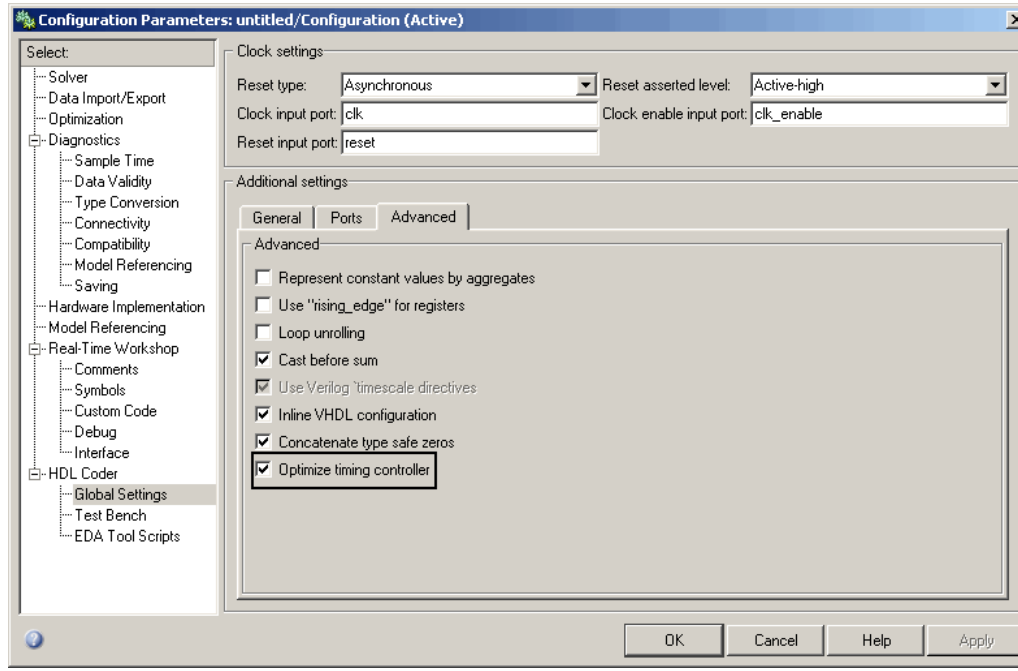
Integrating FPGA Vendor Tools with Simulink HDL Coder

You can now integrate Simulink HDL Coder with third-party FPGA vendor tools for HDL code generation. For detailed information on how to do this, see the Simulink HDL Coder Technical literature page: http://www.mathworks.com/products/slhdlcoder/technicalliterature.html.

Timing Controller Optimization for Multirate Models

The new **Optimize timing controller** option (and the corresponding `OptimizeTimingController` CLI property) optimizes generated `TimingController` entities for speed and code size by generating multiple counters (one counter for each rate in the model) in the timing controller code. The benefit of this optimization is that it generates faster logic, and reduces generated code size.

By default, the **Optimize timing controller** option is selected, as shown in the following figure.



See “Optimize timing controller” for further details.

Enhanced modelscope Syntax Increases Portability of Control Files

The `modelscope` argument to the `forEach` and `forAll` control file methods has been enhanced to allow use of the period (.) to represent the root-level model. This lets you represent the current model as an abstraction, instead of explicitly coding the model name, as in the following example:

```
cfg.forEach( './Subsystem/MinMax', ...
   'built-in/MinMax', {}, ...
   'hdldefaults.MinMaxCascadeHDLEmission');
```

If you represent the model in this way, and then save the model under a different name, the control file does not require any change. Using the period to represent the root-level model makes the modelscope independent of the model name, and therefore more portable.

See also “Representation of the Root Model in modelscopes” in the Simulink HDL Coder User’s Guide.

Compatibility Considerations

When you save HDL code generation settings to a control file, the control file contains a `generateHDLFor` statement that specifies the path to the DUT specified in the **Generate HDL for** field. In previous releases, the root-level model in this path was represented by an explicit model name reference. In release 2008a, by default, the root-level model is represented by the period, as described above.

If you resave model settings to an existing control file, be aware that such explicit references to root-level model name will be changed to the period syntax, in accordance with this new default. This will not affect the operation of your existing control files in any way.

Limited Variable-Step Solver Support

In previous releases, only fixed-step solvers were supported for HDL code generation. In the current release, you can select a variable-step **Solver type** for your model, under the following limited conditions:

- The device under test (DUT) is single-rate.
- The sample times of all signals driving the DUT are greater than 0.

Version 1.2 (R2007b) Simulink HDL Coder Software

This table summarizes what's new in V1.2 (R2007b):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports	No

New features and changes introduced in this version are:

HDL Code Generation for Single-Clock Multirate Models

The coder now supports HDL code generation for single-clock, single-tasking multirate models. Your model can include blocks running at multiple sample rates:

- Within in the device under test (DUT)

- In the test bench driving the DUT
- In both the test bench and the DUT

Multirate code generation support is described in detail in "Generating HDL Code for Multirate Models" in the documentation.

Additional Blocks Supported for Multirate Code Generation

The following blocks, frequently used in construction of multirate models, are now supported for HDL code generation:

- Signal Attributes/Rate Transition
- Signal Processing Blockset/Signal Operations/Downsample
- Signal Processing Blockset/Signal Operations/Upsample

New Property Added in Support of Multirate Code Generation

To support multirate code generation, a new `makehdl` property, `HoldInputDataBetweenSamples`, has been added. This property determines how long (in terms of base rate clock cycles) data values for subrate signals are held in a valid state. See `HoldInputDataBetweenSamples` for details.

Requirements and Restrictions for Multirate Code Generation

Certain requirements and restrictions apply to the use of multirate models for HDL code generation. See "Configuring Multirate Models for HDL Code Generation" for further information.

Additional Blocks Supported for HDL Code Generation

The coder now supports the following blocks for HDL code generation:

- Additional Math & Discrete/Additional Discrete/Unit Delay Enabled
- Math Operations/Divide
- Math Operations/Math Function (`sqrt` function only)
- Signal Attributes/Rate Transition

- Signal Processing Blockset/Signal Operations/Downsample
- Signal Processing Blockset/Signal Operations/Upsample
- Dual Port RAM (For information on this new block, see also "Dual Port RAM Block Supported for Simulation and Code Generation" on page 66.)

See "Summary of Block Implementations" for a complete listing of blocks that are currently supported for HDL code generation.

Dual Port RAM Block Supported for Simulation and Code Generation

The coder now provides the Dual Port RAM Block for use in simulation and code generation.

The Dual Port RAM block lets you:

- Simulate the behavior of a dual-port RAM with registered outputs in your model.
- Generate an interface to the inputs and outputs of the RAM in HDL code.

See "RAM Blocks" for full details.

Block Implementation Parameters Include Output Pipelining

The coder now supports *block implementation parameters*, which let you control details of the code generated for specific block implementations. Block implementation parameters are passed as property/value pairs to `forEach` or `forAll` calls in a code generation control file.

Supported Block Implementation Parameters

Block implementation parameters supported in the current release include:

- `'OutputPipeline', nStages`: This parameter lets you specify a pipelined implementation for selected blocks. The parameter value (`nStages`) specifies the number of pipeline stages (pipeline depth) in the generated

code. `OutputPipeline` is supported by most Simulink HDL Coder HDL Coder block implementations.

- Interface generation parameters let you customize features of an interface generated for the following block types:
  - simulink/Ports & Subsystems/Model
  - built-in/Subsystem
  - lfilinklib/HDL Cosimulation
  - modelsimlib/HDL Cosimulation

  For example, you can specify generation of a black box interface for a subsystem, and pass in parameters that specify the generation and naming of clock, reset, and other ports in HDL code. Interface generation parameters are described in "Customizing the Generated Interface".

For more information on block implementation parameters, see the following sections in the documentation:

- "Specifying Block Implementations and Parameters in the Control File"
- "Block Implementation Parameters"
- "Summary of Block Implementations"

Using hdlnewforeach to Find Block Implementation Parameters

Given a selection of one or more blocks from your model, the `hdlnewforeach` function returns information about the available HDL implementations for each block.

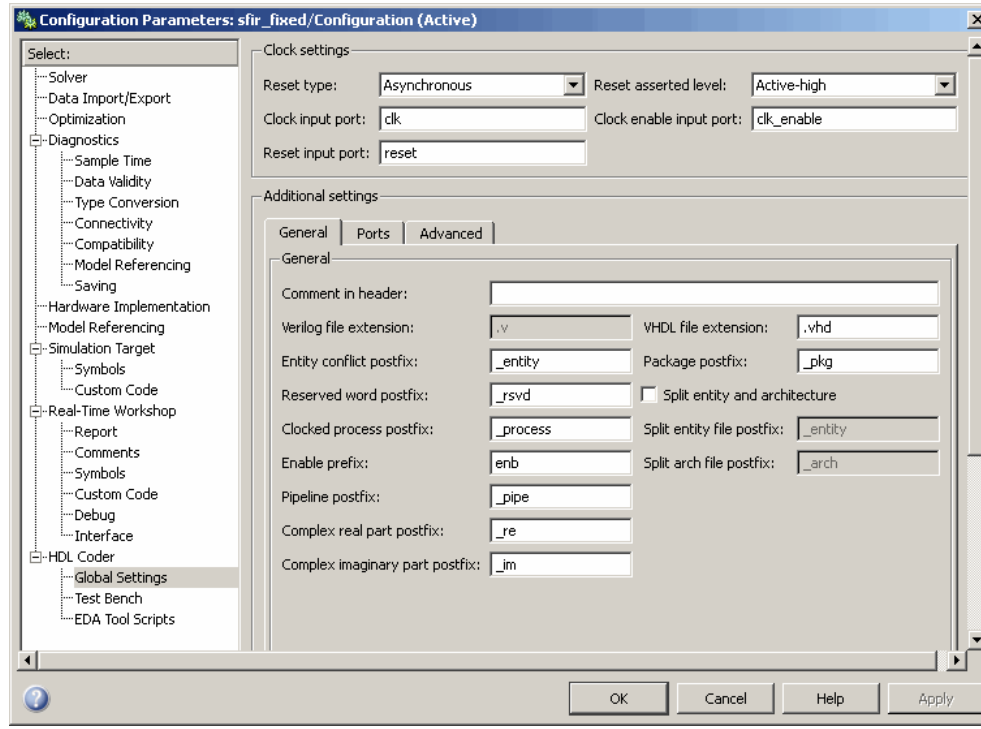
In the current release, the information returned by `hdlnewforeach` has been expanded. `hdlnewforeach` now returns an optional cell array of strings specifying the parameter(s) corresponding to each block implementation.

See "Generating Selection/Action Statements with the hdlnewforeach Function" for details.

Summary of GUI Updates

The following updates have been made to the Simulink HDL Coder GUI:

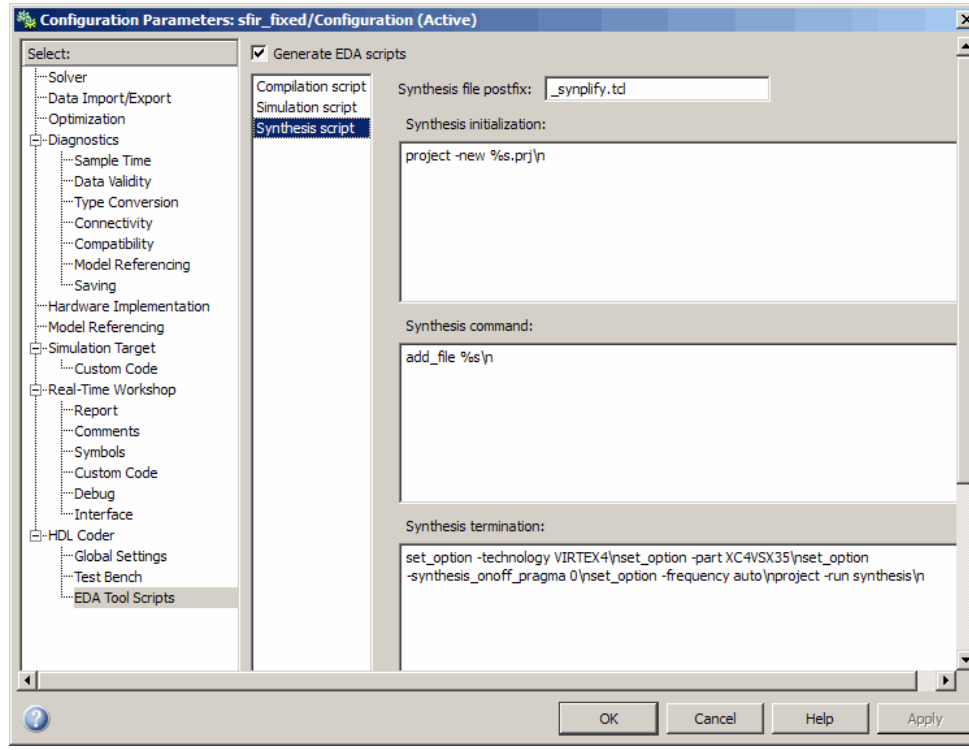
- The **Enable prefix** option is now supported by the GUI as well as by the `EnablePrefix` command-line property. See “Enable prefix” for details on this option.



- The default value for the **Synthesis termination** field of the EDA Tool Scripts dialog box has changed, as shown in the following figure. The default hardware target string in generated synthesis scripts now specifies
  - `technology` option: `VIRTEX4`

    In previous releases, this option defaulted to `VIRTEX2`.
  - `part` option: `XC4VSX35`

    In previous releases, this option defaulted to `XC2V500`.



See also “Default Hardware Target for Synthesis Scripts Updated to Virtex-4 ” on page 70.

Digital Filter Block Restriction Removed

In previous releases, Filter Design HDL Coder™ software was required to generate HDL code for the Digital Filter block when the **Dialog parameters** option was selected in the **Coefficient source** option group. This requirement has been removed.

In the current release, the HDL code generation requirements for the Digital Filter block vary according to the **Coefficient source** option you select, as follows:

- **Dialog parameters**: No additional toolboxes or blocksets required for HDL code generation.

- **Discrete-time filter object**: Filter Design HDL Coder software required.
- **Input port(s)**: This option is not supported for HDL code generation.

Support for New Embedded MATLAB Bitwise Functions

The code supports the new Embedded MATLAB fixed-point bitwise functions introduced in R2007b. Many of these functions map directly to HDL bitwise operators, resulting in very efficient HDL code. See "Using Fixed-Point Bitwise Functions" for examples of the use of these functions in HDL code generation.

For general information on Embedded MATLAB bitwise functions, see "Bitwise Operations" in the Fixed-Point Toolbox documentation.

Compatibility Considerations

In previous releases, the return type of the `bitget` function was `ufix8`. For more efficient HDL code generation, the return data type of the `bitget` function has been changed to `ufix1`. If your existing Embedded MATLAB code performs type casts to adapt values returned from `bitget` for HDL code generation, you may be able to eliminate these type casts.

Default Hardware Target for Synthesis Scripts Updated to Virtex-4

The default hardware target string in generated synthesis scripts now specifies

- `technology` option: `VIRTEX4`

  In previous releases, this option defaulted to `VIRTEX2`.

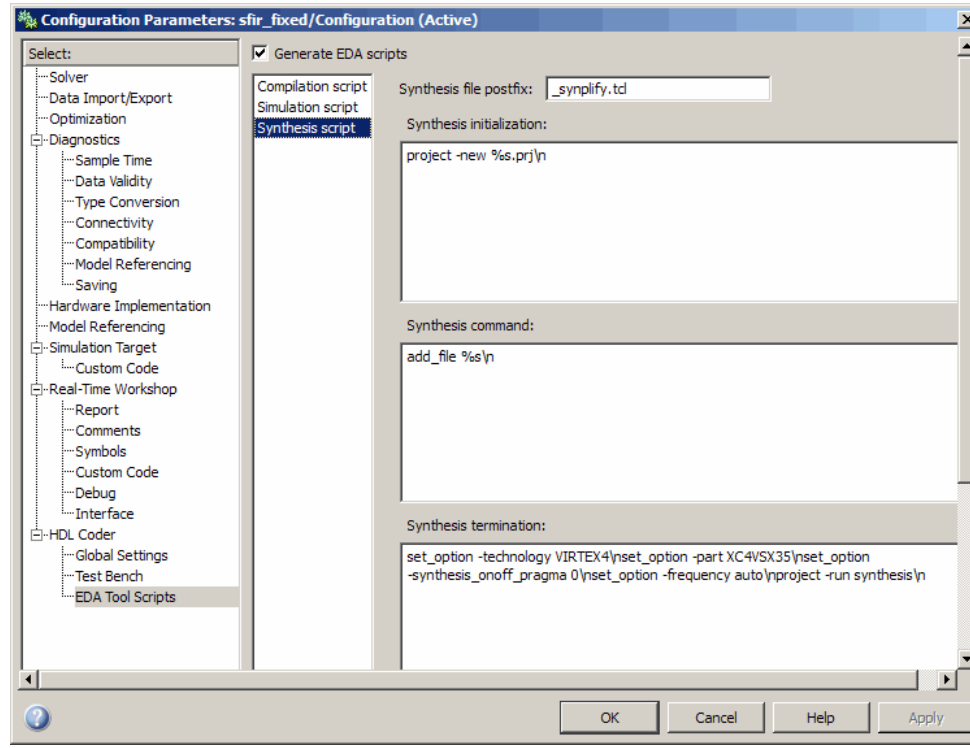
- `part` option: `XC4VSX35`

  In previous releases, this option defaulted to `XC2V500`.

These updates affect the default value for the `HDLSynthTerm` property. The default is:

```
['set_option -technology VIRTEX4\n',...
'set_option -part XC4VSX35\n',...
'set_option -synthesis_onoff_pragma 0\n',...
'set_option -frequency auto\n',...
'project -run synthesis\n']
```

The default value for the `HDLSynthTerm` property appears in the **Synthesis termination** field of the EDA Tool Scripts dialog box, as shown in the following figure.



See also “Generating Scripts for HDL Simulators and Synthesis Tools”.

Compatibility Considerations

If you have existing models that generate synthesis scripts using the previous defaults for `technology` or `part`, you may want to update your models and regenerate scripts.

Version 1.1 (R2007a) Simulink HDL Coder Software

This table summarizes what's new in V1.1 (R2007a):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	Bug Reports	No

New features and changes introduced in this version are

Sign Block Supported for HDL Code Generation

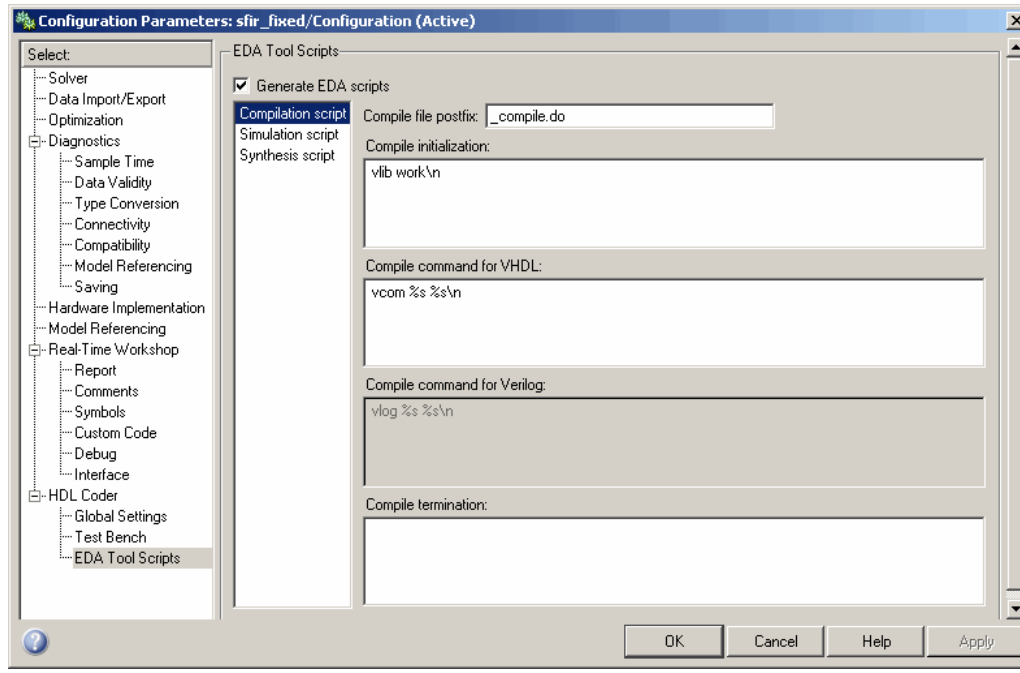
The Sign block (Simulink/Math Operations/Sign) is now supported for HDL code generation.

Link for Cadence Incisive HDL Cosimulation Block Supported

The coder now supports HDL code generation for the Link for Cadence® Incisive® HDL Cosimulation Block. You can use the HDL Cosimulation block with the coder to generate an interface to your manually written or legacy HDL code. When an HDL Cosimulation block is included in a model, the coder generates a VHDL or Verilog interface, depending on the selected target language. See "Code Generation for HDL Cosimulation Blocks" for details.

GUI Support for Generation of EDA Tool Scripts

The new **EDA Tool Scripts** pane of the GUI (shown in the following figure) lets you set all options that control generation of script files for third-party electronic design automation (EDA) tools. In previous releases, script generation options were available only through `makehdl` and `makehdltb` properties.



The list on the left of the **EDA Tool Scripts** pane lets you select from the following categories of options:

- **Compilation script**: Options related to customizing scripts for compilation of generated VHDL or Verilog code.
- **Simulation script**: Options related to customizing scripts for HDL simulators.
- **Synthesis script**: Options related to customizing scripts for synthesis tools.

See “Generating Scripts for HDL Simulators and Synthesis Tools” for detailed information on the **EDA Tool Scripts** options and on script generation in general.

Embedded MATLAB Function Block Supported for HDL Code Generation

The coder now supports synthesizable HDL code generation from the Embedded MATLAB Function block. See “Generating HDL Code with the Embedded MATLAB Function Block” for detailed information.

Stateflow HDL Code Generation Updates

This section describes some limitations on the use of Stateflow charts in HDL code generation have been removed in the current release. These are:

Restriction on Reading from Output Ports Removed

In the previous release, reading from output ports was disallowed. This restriction has been relaxed. You can now read from output ports if outputs are registered. (Outputs are registered if the **Initialize Outputs Every Time Chart Wakes Up** option is deselected.)

Stateflow Charts Fully Support Fixed Point Data Type

In the previous release, fixed-point data type support for Stateflow HDL code generation was limited to fixed point without scaling. This limitation has been removed. You can now use fixed-point data types without restriction in Stateflow charts intended for HDL code generation.

Compatibility Summary for Simulink HDL Coder Software

This table summarizes new features and changes that might cause incompatibilities when you upgrade from an earlier version, or when you use files on multiple versions. Details are provided in the description of the new feature or change.

Version (Release)	New Features and Changes with Version Compatibility Impact
Latest Version V1.7 (R2010a)	See the **Compatibility Considerations** subheading for this new feature or change: • "Simplified Syntax for Specification of Block Implementations in Control Files" on page 5
V1.6 (R2009b)	See the **Compatibility Considerations** subheading for this new feature or change: • "DUT Argument Required for checkhdl and makehdl Commands" on page 18 • "Algebraic Loops Disallowed for HDL Code Generation" on page 18 • "AddClockEnablePort Implementation Parameter for RAM Blocks Deprecated" on page 19

Version (Release)	New Features and Changes with Version Compatibility Impact
V1.5 (R2009a)	See the **Compatibility Considerations** subheading for this new feature or change: • "New Default HDL Implementations for Selected Blocks" on page 30 • "New HDL Implementations for Selected Blocks" on page 31
V1.4 (R2008b)	See the **Compatibility Considerations** subheading for this new feature or change: • "Default Entity Conflict Postfix Changed" on page 44 • "-novopt Flag Added to Default Simulation Command in Generated Compilation Scripts" on page 47 • "New DistributedPipelining Implementation Parameter for Embedded MATLAB Function Blocks and Stateflow Charts" on page 44
V1.3 (R2008a)	See the **Compatibility Considerations** subheading for this new feature or change: • "Enhanced modelscope Syntax Increases Portability of Control Files" on page 62

Version (Release)	New Features and Changes with Version Compatibility Impact
V1.2 (R2007b)	See the **Compatibility Considerations** subheading for this new feature or change: • "Default Hardware Target for Synthesis Scripts Updated to Virtex-4 " on page 70 • "Support for New Embedded MATLAB Bitwise Functions" on page 70
V1.1 (R2007a)	None