Simulink® Coder™ Release Notes

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

R2017a

R2016b

R2016a

R2015aSP1

Bug Fixes

R2015b

R2015a

R2014b

R2013b

R2013a

R2012b

R2012a

R2011b

R2011a

R2017b

Version: 8.13

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

C++ Functions: Generate C++ code from Simulink functions, including functions that respond to initialize, reset, and terminate events

Generate C++ Code for Simulink functions

In R2014b, the code generator introduced C code generation support for Simulink Function and Function Caller blocks. R2017b adds C++ code generation support for these blocks. Using the blocks, you can:

- Define a function that can be invoked by a function caller.
- Call a function to compute output.
- Save and restore a state from nonvolatile memory.
- Provide entry-point functions that respond to external reset events.

You can invoke a C function that the code generator produces from a Simulink Function block with code generated from a Stateflow® chart, but not for a C++ function. Due to current scope limitations for generated C++ functions, you must invoke those functions with code generated from a Function Caller block.

For more information, see "Generate Code That Responds to Initialize, Reset, and Terminate Events" and the Simulink Function and Function Caller block reference pages.

Generate C++ Code for initialize, reset, and terminate events

In R2016b, the code generator introduced C code generation support for the blocks Initialize Function, Reset Function, and Terminate Function. R2017b adds C++ code generation support for these blocks. You can use these blocks to generate code that controls execution of a component in response to initialize, reset, and terminate events. For example, use these blocks to generate code that:

- Starts and stops an application component
- Calculates initial conditions

For more information, see"Simulink Function Blocks and Code Generation" and the Initialize Function, Reset Function, and Terminate Function block reference pages.

Data, Function, and File Definition

Tunable Parameters: Tune parameters in model workspace

A storage class causes a parameter object to appear in the generated code as a global variable whose value you can change during execution. For example, you can apply the storage class `ExportedGlobal` to a `Simulink.Parameter` object. Before R2017b, you could not apply a storage class other than `Auto` to parameter objects that you stored in a model workspace. In R2017b, you can apply a storage class other than `Auto`. However, if you store an `AUTOSAR.Parameter` object in a model workspace, the code generator ignores the storage class that you specify for the object.

A storage class yields a global variable (or some other global symbol such as a macro) in the generated code, which means the variable can have only one definition in the code. However, in a model reference hierarchy in Simulink, a parameter object in a model workspace can have the same name as a different parameter object in the model workspace of another model. If these parameter objects use storage classes other than `Auto`, when you attempt to generate code from the hierarchy, the definitions of the corresponding global variables conflict, preventing code generation. In general, you must make sure that each global symbol in a model hierarchy is unique.

Compatibility Considerations

Before R2017b, when you set the model configuration parameter **Default parameter behavior** to `Tunable`, a variable or parameter object that you stored in a model workspace did not appear in the generated code as a single, tunable entity that resides in memory (such as a field of the *model*_P structure). Instead, the code generator created a separate structure field for each block parameter that used the variable or object. In R2017b, the code generator creates a single structure field for each variable or parameter object.

When you generate code from a model that you created before R2017b, parameter diagnostics can generate new warnings or errors. For example, if the model uses such a variable or object in an expression that the code generator cannot preserve in the generated code, depending on the setting for the model configuration parameter **Detect loss of tunability**, attempting to generate code from the model can yield a new warning or error.

Virtual Buses Across Model Reference Boundaries: Check for large numbers of function arguments caused by virtual buses

When a virtual bus signal with many bus elements enters or exits a referenced model, the entry-point functions generated for the model (such as `model_step`) exchange data through separate arguments, one argument for each bus element that the model uses. In R2017b, you can use the new Model Advisor check **Check for large number of function arguments from virtual bus across model reference boundary** to identify such buses. To generate code that passes a structure pointer instead of many individual arguments, click **Update Models**. Simulink then makes the target buses nonvirtual by configuring Inport and Outport block parameters and inserting Signal Conversion blocks as necessary.

Code Generation

Configuration Parameters Dialog Box: View your model and code generation configuration parameters in unified dialog box with search capability

Previously, the Configuration Parameters dialog box contained two tabs: a tab for commonly used parameters and a tab that provided a searchable list of all available parameters. In R2017b, the Configuration Parameters dialog box combines these features in a unified dialog box with a search capability.

- View commonly used parameters on a category pane. Access advanced category parameters on the same pane.
- To quickly find a specific parameter in the dialog box, use the search tool.
- Right-click a parameter to get the parameter name to use in scripts, view parameter dependencies, and navigate to parameter documentation.

For more information, see "Configuration Parameters Dialog Box Overview" (Simulink).

Compatibility Considerations

- In R2017b, advanced parameters that were previously available only on the **All Parameters** tab can be found under the **Advanced Parameters** toggle of the relevant category pane. To access this toggle, hover over the ellipsis at the bottom of the pane. Alternatively, to find an advanced parameter, use the search tool at the top of the dialog box.
- If you use an `sl_customization.m` script to hide or disable parameters in the Configuration Parameters dialog box, the script requires updates to widget ID's and callback registrations. For example:

  - In R2017a:

    ```
    function sl_customization(cm)

    % Disable for standalone Configuration Parameters dialog box.
    cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBrowseButton)
    % Disable for Configuration Parameters dialog box
    cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBrowseButton)

    end
    ```

```
function disableRTWBrowseButton(dialogH)

  % Takes a cell array of widget Factory ID.
  dialogH.disableWidgets({'Tag_ConfigSet_RTW_Browse'})

end
```

- In R2017b:

```
function sl_customization(cm)

% Disable for all Configuration Parameters dialog boxes
configset.dialog.Customizer.addCustomization(@disableRTWBrowseButton,cm);

end

function disableRTWBrowseButton(dialogH)

    % Takes a cell array of widget Factory ID.
    dialogH.disableWidgets({'STF_Browser'})

end
```

For more information on getting widget ID's and customizing the dialog box, see "Disable and Hide Dialog Box Controls" (Simulink).

Simplified Build Folder Layout: Generate code for different hardware settings in separate folders

Specify separate folders for generated code from models that are configured for different target environments. Use the `Simulink.fileGenControl` option, `CodeGenFolderStructure`, or the Simulink preference, **Code generation folder structure**.

If you use this approach, do not manually specify the folder and subfolder locations for simulation and generated code files. Use the information that `RTW.getBuildDir` provides. Custom targets that do not use location information from `RTW.getBuildDir` might not support this approach.You can obtain the current value of `CodeGenFolderStructure` with this command:

```
Simulink.fileGenControl('get', 'CodeGenFolderstructure')
```

For more information, see:

- "Manage Build Process Folders"
- `Simulink.fileGenControl`

Warning Messages: Build process diagnostic warnings in Diagnostic Viewer

Previously, build process diagnostic warnings were in the build log. These warnings are now in the **Diagnostic Viewer**. This change increases the visibility of these warning messages.

Code Generation Advisor: Updates to parameter recommendations for objectives

In R2017b, when the Code Generation Advisor checks your model configuration settings against code generation objectives, these changes apply:

- For checks against safety precaution objectives, the Code Generation Advisor does not consider the parameter Conditional input branch execution (`ConditionallyExecuteInputs`).

- For checks against ROM efficiency objectives, the Code Generation Advisor considers the parameter Remove code that protects against division arithmetic exceptions (`NoFixptDivByZeroProtection`).

- For checks against ROM and execution efficiency objectives, the Code Generation Advisor considers the parameter Support long long (`ProdLongLongMode`).

Performance

Fast Fourier Transforms in a MATLAB Function Block: Generate code that takes advantage of the FFTW library

In previous releases, when you generated code for FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, and `ifftn`) in a MATLAB Function block, the code generator produced code for the FFT algorithms.

In R2017b, to improve the execution speed of code generated for FFT functions, the code generator can produce calls to an FFT library. To generate calls to a specific, installed FFTW library, provide an FFT library callback class. See "Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block".

For more information about the FFTW library, see www.fftw.org.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2017a

Version: 8.12

New Features

Bug Fixes

Model Architecture and Design

Subsystem Reuse Across Models: Reuse subsystems with naming control and global Data Store Memory blocks across models

In R2017a, the code generator can generate reusable code for the following modeling patterns:

- Subsystems across model reference boundaries that contain Data Store blocks that read from or write to a global data store. In the Data Store Block Parameters dialog box, you specify a global data store for the **Data store name** parameter. You define the global data store in the base workspace using a signal object.

- Reusable functions that have user-specified names. In the Subsystem Block Parameters dialog box, on the **Code Generation** tab, you specify a name by selecting `User specified` for the **Function name options** parameter and providing a name for the **Function name** parameter. The code generator no longer appends the user-specified name with a checksum.

The reusable code is in the shared utilities folder (`slprj/`)*target*`/_sharedutils`. Generating reusable code conserves ROM consumption and improves code execution speed. See Subsystems.

TMF and EXTMODE fields optional in TLC file

In R2017a, if you do not specify the `TMF` or `EXTMODE` fields in a system target TLC file, the file is still valid. To change the values for the parameters **Template makefile** (`TemplateMakefile`) and **External mode** (`ExtMode`), you can instead use the callback function specified by `rtwgensettings.SelectCallback`.

For more information, see Customize System Target Files.

Data, Function, and File Definition

Association of root-level Outport block with Simulink.Signal object

Before R2017a, you could not associate a root-level Outport block with a `Simulink.Signal` object.

In R2017a, you can use the Model Data Editor (see Configure Data Properties by Using a Table) to make this association.

MAT-file logging for root-level Outport blocks with storage class other than Auto

Before R2017a, when you used the Model Data Editor to apply a storage class or custom storage class other than `Auto` to a root-level Outport block, MAT-file logging (**Configuration Parameters > MAT-file logging**) did not support the Outport block.

In R2017a, MAT-file logging supports the Outport block unless the storage class is `ImportedExternPointer` or yields nonaddressable data in the generated code. For example, the custom storage class `GetSet` causes the Outport to appear in the generated code as a function call, which is not addressable.

Model Explorer accessibility for code generation settings of lookup table and breakpoint objects

Before R2017a, in the Model Explorer **Contents** pane, you were not able to access some code generation settings (properties) of `Simulink.LookupTable` and `Simulink.Breakpoint` objects, including:

- `StorageClass`
- `HeaderFile` (for the variable that appears in the generated code)
- `HeaderFileName` (for the structure type that appears in the generated code)
- `DefinitionFile`
- `Alignment`
- `SupportTunableSize`

Except for `SupportTunableSize`, these properties belong to the `Simulink.CoderInfo` and `Simulink.lookuptable.StructTypeInfo` objects that reside in the `CoderInfo` and `StructTypeInfo` properties of the lookup table or breakpoint object.

In R2017a, you can inspect and modify these code generation settings in the Model Explorer **Contents** pane.

Code Generation

Build Process Customization for S-Functions: Customize generated makefiles with RTW.BuildInfo functions in makecfg.m

To customize generated makefiles for S-functions, create `makecfg.m` and *yourSFunction*`_makecfg.m` files that use `RTW.BuildInfo` functions to specify:

- Additional source files and libraries
- Preprocessor macro definitions
- Compiler flags

For more information, see:

- Use makecfg to Customize Generated Makefiles for S-Functions
- Import Calls to External Code into Generated Code with Legacy Code Tool

Source file includes shared utility header file

In R2016b, the *model*`.h` file or the *subsystem*`.h` file contained the `#include` command for the header file that contained declarations for shared utility functions. In R2017a, the source file contains the `#include` command for this header file. Including the header file in the source file reduces compilation time and improves code readability because the *model*`.c` file or the *subsystem*`.c` file uses the shared utility functions. The *model*`.h` file or the *subsystem*`.h` file does not use the shared utility functions.

Generated code for Rate Transition block variables with volatile qualifier

In R2016b, for Rate Transition blocks, when you selected the **Ensure data integrity during data transfer** parameter and cleared the **Ensure deterministic data transfer (maximum delay)** parameter, some compilers performed optimizations that removed or reordered protection logic, which caused data integrity issues. Protection logic is particularly important for safety-critical system deployment. To avoid potential issues, users had to turn off compiler optimizations.

In R2017a, the generated code contains the `volatile` qualifier in variables in the `D_Work` structure. This qualifier indicates to most compilers not to perform optimizations

that remove or reorder protection logic because the value of these variables can possibly change outside control or detection of the program. The presence of the volatile qualifier means that users no longer have to turn off compiler optimizations and sacrifice performance for safety.

For example, the model rtwdemo_ratetrans contains six Rate Transition blocks. In the block parameters dialog boxes for IntegOnlyF2S and IntegOnlyS2F, the **Ensure data integrity during data transfer** parameter is selected and the **Ensure deterministic data transfer (maximum delay)** parameter is cleared.
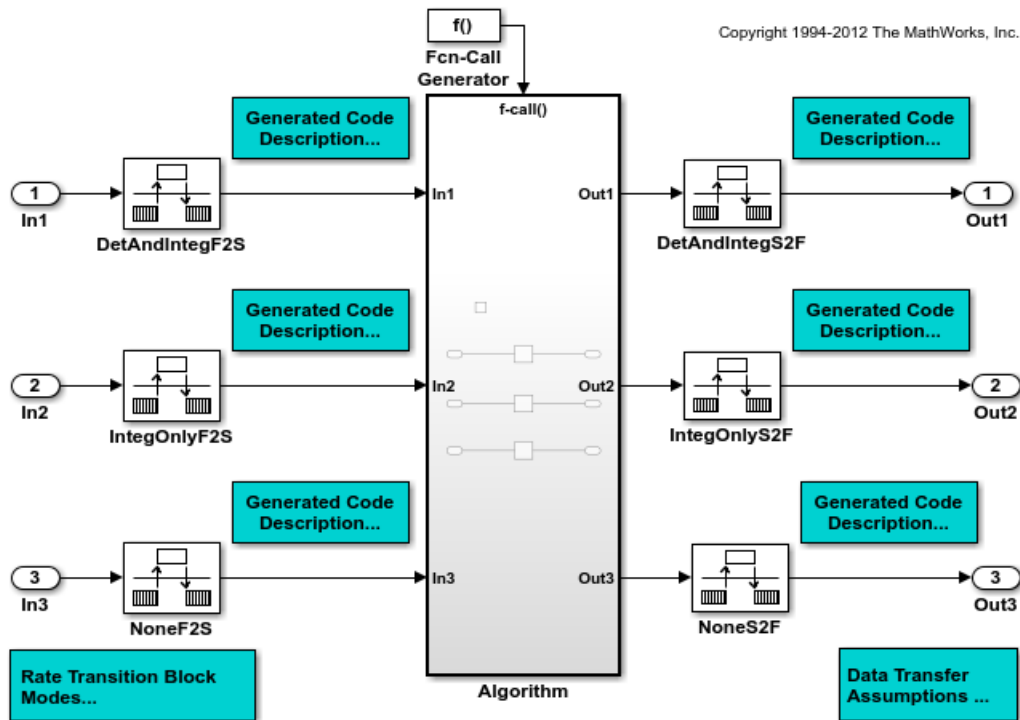
In R2016b, the rtwdemo_ratetrans.h file contained this code:

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T Integrator1_DSTATE[20];      /* '<S1>/Integrator1' */
  real_T Integrator2_DSTATE[20];      /* '<S1>/Integrator2' */
  real_T Integrator3_DSTATE[20];      /* '<S1>/Integrator3' */
  real_T DetAndIntegS2F_Buffer0[20];  /* '<Root>/DetAndIntegS2F' */
```

```
  real_T IntegOnlyS2F_Buffer[40];      /* '<Root>/IntegOnlyS2F' */
  real_T IntegOnlyF2S_Buffer0[20];     /* '<Root>/IntegOnlyF2S' */
  uint32_T Algorithm_ELAPS_T[2];       /* '<Root>/Algorithm' */
  uint32_T Algorithm_PREV_T[2];        /* '<Root>/Algorithm' */
  int8_T IntegOnlyS2F_ActiveBufIdx;    /* '<Root>/IntegOnlyS2F' */
  int8_T IntegOnlyF2S_semaphoreTaken;  /* '<Root>/IntegOnlyF2S' */
} DW_rtwdemo_ratetrans_T;
```

The Rate Transition block variables did not contain the `volatile` qualifier.

In R2017a, the `rtwdemo_ratetrans.h` file contains this code:

```
/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T Integrator1_DSTATE[20];       /* '<S1>/Integrator1' */
  real_T Integrator2_DSTATE[20];       /* '<S1>/Integrator2' */
  real_T Integrator3_DSTATE[20];       /* '<S1>/Integrator3' */
  real_T DetAndIntegS2F_Buffer0[20];   /* '<Root>/DetAndIntegS2F' */
  volatile real_T IntegOnlyS2F_Buffer[40];/* '<Root>/IntegOnlyS2F' */
  volatile real_T IntegOnlyF2S_Buffer0[20];/* '<Root>/IntegOnlyF2S' */
  uint32_T Algorithm_ELAPS_T[2];       /* '<Root>/Algorithm' */
  uint32_T Algorithm_PREV_T[2];        /* '<Root>/Algorithm' */
  volatile int8_T IntegOnlyS2F_ActiveBufIdx;/* '<Root>/IntegOnlyS2F' */
  volatile int8_T IntegOnlyF2S_semaphoreTaken;/* '<Root>/IntegOnlyF2S' */
} DW_rtwdemo_ratetrans_T;
```

The variables corresponding to Rate Transition blocks that have the **Ensure data integrity during data transfer** parameter selected and the **Ensure deterministic data transfer (maximum delay)** parameter cleared contain the `volatile` qualifier. These variables have the `IntegOnly` comment.

IncludeMdlTerminateFcn not checked against efficiency objectives

In R2017a, when the Code Generation Advisor checks your model configuration settings against code generation efficiency objectives, it does not consider the parameter **Terminate function required** (`IncludeMdlTerminateFcn`).

More information in code generation report summary

Additional fields in the code generation report **Summary** page provide information on your model and the generated code, including:

- **Author**
- **Last Modified By**
- **Tasking Mode** (except for exported models)
- **System Target File**
- **Hardware Device Type**
- **Type of Build**
- **Code Generation Advisor** (if you run Code Generation Advisor as part of the build process, it provides link to **Code Generation Advisor Report**)
- **Code Reuse Exception** (if exceptions exist, it links to **Subsystem Report**)

For more information on code generation reports, see Reports for Code Generation.

Deployment

NXP FRDM-K64F Board: Create models using Analog Output, Quadrature Encoder, Serial, and UDP blocks

This table lists the support for these new blocks.

Block	Usage
Analog Output	Send an analog signal to DAC0_OUT pin
Quadrature Encoder	Measure the rotation of the encoder in ticks
Serial Receive	Read data from the UART port
Serial Transmit	Send data to the UART port
UDP Receive	Receive UDP packets from another UDP host
UDP Send	Send UDP packets to another UDP host

Support for new board STMicroelectronics Nucleo F746ZG

You can use the Simulink Coder Support Package for STMicroelectronics® Nucleo Boards to generate code for STMicroelectronics Nucleo F746ZG board.

You must install the Simulink Coder Support Package for STMicroelectronics Nucleo Boards to use this support.

Support for new board STMicroelectronics Nucleo F411RE

You can use the Simulink Coder Support Package for STMicroelectronics Nucleo Boards to generate code for STMicroelectronics Nucleo F411RE board.

You must install the Simulink Coder Support Package for STMicroelectronics Nucleo Boards to use this support.

Gyroscope and LCD blocks added to ARM Cortex-based VEX Microcontroller

This table lists the support for these new blocks.

Block	Usage
Gyroscope	Measure the yaw rotation in degrees
LCD Button	Read the state of the selected button
LCD Screen	Display text and numbers on the display

Performance

Dynamic Memory Allocation for MATLAB Function Block: Generate C code that uses dynamic memory allocation

In R2017a, simulation and C/C++ code generation support dynamic memory allocation for arrays in a MATLAB Function block, a Stateflow chart, or a System object™ associated with a MATLAB System block. Dynamic memory allocation allocates memory as needed at run time, instead of allocating memory statically on the stack. Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

By default, dynamic memory allocation for MATLAB Function blocks is enabled for GRT-based targets and disabled for ERT-based targets. To change the setting, in the Configuration Parameters dialog box, on the **All Parameters** tab, in the **Simulation Target > Advanced parameters** category, clear or select the **Dynamic memory allocation in MATLAB Function blocks** check box.

When dynamic memory allocation is enabled, the code generator uses dynamic memory allocation for arrays whose size is equal to or greater than a threshold. The default value of this threshold is 64 kilobytes. To change the threshold, in the Configuration Parameters dialog box, on the **All Parameters** tab, in the **Simulation Target > Advanced parameters** category, set the **Dynamic memory allocation threshold in MATLAB Function blocks** parameter.

In the generated C/C++ code, the code generator represents dynamically allocated data as a structure type called `emxArray`. The code generator produces utility functions that the generated code uses to manage the `emxArrays`. If you have Embedded Coder®, you can customize the identifiers for `emxArrays` and the utility functions.

- To customize `emxArray` identifiers, use the **EMX array types identifier format** parameter in the **Simulation Target > Advanced parameters** category of the **All Parameters** tab.
- To customize the utility function identifiers, use the **EMX array utility functions identifier format** parameter in the **Simulation Target > Advanced parameters** category of the **All Parameters** tab.

Dynamic memory allocation does not apply to:

- Input and output signals. Variable-size input and output signals must have an upper bound.
- Parameters or global variables. Parameters and global variables must be fixed-size.
- Fields of bus arrays. Bus arrays cannot have variable-size fields.
- Discrete state properties of System objects associated with a MATLAB System block.

See Control Memory Allocation for Variable-Size Arrays in a MATLAB Function Block.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2016b

Version: 8.11

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Initialize Function and Terminate Function Blocks: Generate code for initialize, reset, and terminate events

R2016b introduces the blocks Initialize Function and Terminate Function. You can use these blocks to generate code that controls execution of a component in response to initialize, reset, or terminate events. For example, use them to generate code that:

- Starts and stops an application component.
- Calculates initial conditions.
- Saves and restores state from nonvolatile memory.
- Provides entry-point functions that respond to external reset events.

For more information, see Generate Code That Responds to Initialize, Reset, and Terminate Events and descriptions of the Initialize Function and Terminate Function blocks.

State Reader and State Writer Blocks: Generate code that reads or writes state values to set terminal or initial conditions

R2016b introduces State Reader and State Writer blocks. Use these blocks with the new Initialize Function and Terminate Function blocks to generate code that controls execution of a component in response to initialize, reset, or terminate events.

By default, the Initialize Function block includes a State Writer block. The Terminate Function block includes a State Reader block. Set up the State Writer block or the State Reader block to write the state to or read the state from a given state owner block in your model or subsystem. When the function is triggered, the value of the state variable is read from or written to the specified block. The code generator uses unique state names configured for the blocks to identify the reusable function code for a given read or write operation.

Supported state owner blocks include:

- Delay
- Discrete Filter

- Discrete State-Space
- Discrete-Time Integrator
- Discrete Transfer Fcn
- Discrete Zero-Pole
- S-Function
- Trigger
- Unit Delay

For more information, see Generate Code That Responds to Initialize, Reset, and Terminate Events and descriptions of the Initialize Function, Terminate Function, Event Listener, State Reader, and State Writer blocks.

Updates to protected model message identifiers

In R2016b, protected model error message identifiers have been updated.

Compatibility Considerations

If you have protected model code, such as a switch expression, that depends on specific protected model error message identifiers, update this code with the new identifiers.

Data, Function, and File Definition

Name and Storage Class for Outport: Configure name and storage class for code generation directly on root-level Outport blocks

At the root level of a model, Outport blocks represent outputs that other systems can consume as inputs. Prior to R2016b, to configure code generation for an Outport block, you could not apply a name or storage class directly to the block. Instead, you applied a name and storage class to the signal line that entered the block. Optimizations eliminated the Outport block from the generated code, instead allocating memory for the signal line.

In R2016b, use the Model Data Editor (see "Model Data Editor for applying storage classes to Inport blocks, Outport blocks, signals, and Data Store Memory blocks" on page 3-7) to apply a name and storage class directly to an Outport block. You can now:

- Configure system inputs and outputs (Inport and Outport blocks) before you develop the internal algorithm of the system.
- Store the name and storage class specifications in the block. When you delete the signal line that enters the block, you do not lose these specifications.
- Distribute a single signal value to multiple system outputs by branching a signal line to multiple Outport blocks.

To programmatically apply storage classes to Outport blocks, use the new parameters `SignalName`, `StorageClass`, and `SignalObject`.

ASAP2 file generation for bus signals and parameters

R2016b enhances ASAP2 file generation to support bus signals and bus parameters. The following model structures now can be exported as measurements and characteristics and used with ASAP2 based tools to calibrate models:

- Bus type signals and discrete states that are associated with `Simulink.Signal` or `mpt.Signal` derived objects with compatible storage classes
- Bus type test points
- Nested buses and structures for nonlookup parameters
- Nested buses for signals and test points

- Arrays of buses for signals and test points

With nested structure support, you can structure parameters and access each field for calibration. You also can calibrate model reference parameters that are stored in structures.

ASAP2 file generation for nested structures involves additional post-code generation steps, which require:

- A compiler that generates `elf` files
- A `readelf` utility
- Compiling with the debug option

Compatibility Considerations

To support ASAP2 file generation with nested structures, R2016b requires additional post-code generation steps. Also, if you modified a version of the ASAP2 user template `asap2scalar.tlc` from a previous release, R2016b requires minor API and algorithm revisions.

Perform additional steps after code generation for nested structures

C code generation generates model bus parameters and bus signals in variables with nested structures. A map file is not sufficient to retrieve the address of individual fields for each signal or parameter. ASAP2 file generation now uses DWARF debug information to collect structure layout information and emit correct addresses in the `a2l` file. The procedure requires a `readelf` utility.

To calibrate nested structures, perform the following extra steps after code generation.

1 Create a `dwarf` file. Execute the following command in the MATLAB® Command Window.

```
>> !readelf -wi model.elf > model.dwarf;
```

2 If you generated code for referenced models, each model reference build generates an `a2l` file. Merge the files using the `rtw.asap2MergeMdlRefs` function.

```
>> rtw.asap2MergeMdlRefs('topmodel','merged.a2l');
```

3 To add addresses to the `a2l` file, use the `rtw.asap2SetAddress` function.

```
>> rtw.asap2SetAddress('model.a2l','model.dwarf');
```

The extra steps can be integrated into an automated build process. Step 1, `dwarf` file creation, can be included in a template makefile or build tool integration file, following the link command that generates the `elf` file. Alternatively, Step 1 can be included in a post-code generation command.

Steps 2 and 3 can be integrated in a build process hook method in an *STF*_make_rtw_hook file. For example:

```matlab
function ert_make_rtw_hook(hookMethod,modelName,rtwroot,templateMakefile,buildOpts,buildArgs,buildInfo)
% ert_make_rtw_hook - Sample hook file to automate A2L merge and address population
%
  switch hookMethod

   case 'after_make'
    % Called after make process is complete.

    % Merge A2L files for model reference.
    mergea2l( modelName,buildInfo );

  end
end


function mergea2l(modelName,buildInfo)
% Merge the A2L files
% When using model reference, an A2L file is created for each model.
% Merge them into one file.

mdlRefTargetType = get_param(modelName,'ModelReferenceTargetType');
isNotModelRefTarget = strcmp(mdlRefTargetType,'NONE');

if strcmp(get_param(modelName,'GenerateASAP2'),'on')
    if isNotModelRefTarget
        if ~isempty(buildInfo.ModelRefs)
            rtw.asap2MergeMdlRefs(modelName, [modelName '.a2l']);
        end
        rtw.asap2SetAddress([modelName '.a2l'], [modelName '.dwarf']);
    end
end

end
```

Revise ASAP2 user template for nested structures

ASAP2 file generation of bus signals and bus parameters can result in nested C structures and a generated ECU address with multiple levels of nesting. Changes have been made to the ASAP2 user template `asap2scalar.tlc`, which is used to customize how CHARACTERISTICs are emitted in the `a2l` file. If you modified a version of this template from a previous release, incorporate minor revisions to an API and an algorithm in your template.

- API revision — The function `ASAP2UserFcnWriteStructCharacteristic_Scalar` has added a `parentName` parameter.

```
%function ASAP2UserFcnWriteStructCharacteristic_Scalar(param, parentName) Output
```

`parentName` passes the name of the enclosing structure. When using library functions `LibASAP2GetSymbolForBusElement` and `LibASAP2GetAddressForBusElement` to access the symbol and address of the CHARACTERISTIC, reference the new parameter.

```
%assign characteristicName = LibASAP2GetSymbolForBusElement(data,busIdx,"",parentName)
%assign characteristicAddress = LibASAP2GetAddressForBusElement(data,busIdx,"",parentName)
```

The third parameter, `dataIdx`, is omitted, because ASAP2 file generation does not support arrays of busses in CHARACTERISTICs.

- Algorithm revision — In templates from previous releases, you could not recurse inside a structure to add CHARACTERISTICs for nested structures. The following line in the template guarded against recursion:

```
%if !LibIsStructDataType(dtId)
```

You can now add the following code to support recursion. Insert the code immediately before the closing `%endif` statement.

```
%else
   %% Write out CHARACTERISTIC for child structure
   %<ASAP2UserFcnWriteStructCharacteristic_Scalar(data.StructInfo.BusElement[busIdx], parentGrpName)>
```

Model Data Editor for applying storage classes to Inport blocks, Outport blocks, signals, and Data Store Memory blocks

To control the representation of individual signals and Data Store Memory blocks in the generated code, you apply storage classes and custom storage classes. The signals and data stores appear in the generated code as global data that you can access through your custom code.

In R2016b, you can use the Model Data Editor to apply storage classes to these data items. You can view and edit the items in a list that you can sort, group, and filter. Use this technique to inspect and configure the data interface of your model at a high level.

For more information about the Model Data Editor, see Model Data Editor: Configure model data properties using a table within the Simulink Editor. For an example, see Design Data Interface by Configuring Inport and Outport Blocks.

Storage of lookup tables for calibration according to ASAP2 and AUTOSAR standards

Use the new classes `Simulink.LookupTable` and `Simulink.Breakpoint` to store table and breakpoint set data in Simulink. If you have Embedded Coder, use these classes to prepare the data for calibration by packaging it in the generated code according to the ASAP2 (STD_AXIS or COM_AXIS) and AUTOSAR (for example, CURVE or MAP) standards:

- For STD_AXIS, store all of the data in a single `Simulink.LookupTable` object. Use the object in an n-D Lookup Table block.

  The data appear in the generated code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.

- For COM_AXIS, store each unique set of table data in a `Simulink.LookupTable` object and each unique breakpoint vector in a `Simulink.Breakpoint` object. Use each `Simulink.LookupTable` object in an Interpolation Using Prelookup block and each `Simulink.Breakpoint` object in a Prelookup block. You can reduce memory consumption by sharing breakpoint data between lookup tables.

  Each set of table data appears in the generated code as a separate variable. Each breakpoint vector appears as an array or, optionally, as a structure with one field to store the breakpoint data and one field to store the length of the vector. The second field enables you to tune the effective size of the table.

You use these classes in approximately the same way that you use the `Simulink.Parameter` class. For example, you can apply storage classes and custom storage classes. However, you can use these classes only in lookup table blocks.

Tunable Table Size

Prior to R2016b, to tune the effective size of the table in the generated code, in an n-D Lookup Table block, you selected the parameter **Support tunable table size in code generation**. When you used Prelookup and Interpolation Using Prelookup blocks, you could not enable a tunable table size.

In R2016b, you can enable a tunable table size by using the properties of `Simulink.LookupTable` and `Simulink.Breakpoint` objects. Therefore, you can enable a tunable table size whether you use n-D Lookup Table blocks or Prelookup and Interpolation Using Prelookup blocks.

Calibration

To store lookup table data for calibration according to the ASAP2 or AUTOSAR standards (for example, STD_AXIS, COM_AXIS, or CURVE), you can use `Simulink.LookupTable` and `Simulink.Breakpoint` objects. However, some limitations apply. See `Simulink.LookupTable`.

More explicit purpose for SimulinkGlobal storage class

Before R2016b, applying the storage class `SimulinkGlobal` to a signal achieved the same effect as configuring the signal as a test point and applying the default storage class, `Auto`. For example:

- If you configured multiple signals to use the same name and to use `SimulinkGlobal`, the code generator mangled the name of each corresponding structure field to avoid identifier conflicts.
- The model configuration parameter **Ignore test point signals** (Embedded Coder) affected signals that used `SimulinkGlobal` and test points.

If you configured block states to use the same name and `SimulinkGlobal`, the code generator mangled names. Data items that used `SimulinkGlobal` were sometimes subject to code generation optimizations, which possibly removed the data from the code.

There was an overlap of purpose between `SimulinkGlobal` and test point signals due to their similarity. The name mangling made it more difficult to access the data through your custom code. For all kinds of data item, there was an overlap of purpose between `SimulinkGlobal` and `Auto`.

In R2016b, `SimulinkGlobal` represents an explicit specification, similar to other storage classes such as `ExportedGlobal`.

Compatibility Considerations

You can no longer apply the same name to multiple signals or states that use `SimulinkGlobal` because the code generator no longer mangles names. Specify a unique name for each signal and state. Correct existing models that:

- Use the Signal Properties dialog box or block dialog boxes to apply the same name to multiple signals or states that use `SimulinkGlobal`.

- Resolve multiple signal lines or block states to a single `Simulink.Signal` object that uses the storage class `SimulinkGlobal`.

When you apply `SimulinkGlobal` to a data item, optimizations cannot eliminate the data from the generated code. When you select **Ignore test point signals**, optimizations such as the model configuration parameter **Signal storage reuse** do not eliminate signals that use `SimulinkGlobal`.

Additional tunability support for expressions

Previously, to maintain tunability of expressions in the generated code, the data type of workspace variables such as MATLAB variables and `Simulink.Parameter` objects had to be of type `double`. In R2016b, you can specify any data type for these variables and objects. If the data type of these variables and objects and the data type of the corresponding block parameters are the same or a combination of one data type and `double`, the code generator can preserve tunability.

Previously, for blocks that accessed parameter data through pointer or reference in the generated code, you could not specify a math expression that contained workspace variables or used a data type that required an implicit data type conversion. In R2016b, you can specify a math expression or use a data type that is different from the data type of the block parameter. In these cases, the code generator creates an expression that is not addressable to perform the computation. This operation requires a data copy. For large data sets, this data copy can potentially significantly increase RAM consumption and slow down execution speed. For example, Lookup Table blocks often access large vectors or matrices through pointer or reference in the generated code. For maximally efficient code, match the data types of block parameters and workspace variables and specify parameter expressions that are addressable. For example, the name of a single global variable or the field of a structure is addressable.

For more information, see Block Parameter Representation in the Generated Code, Parameter Data Types in the Generated Code, and Optimize Generated Code for Lookup Table Blocks.

Code Generation

Data Exchange Interface: Use independent controls to configure C API, ASAP2, and external mode

Previously, the Simulink Configuration Parameters dialog box allowed you to select only one data exchange interface for your model – C API, ASAP2, or external mode. Selecting a second data exchange interface required using MATLAB `set_param` commands, and the command-line selections were not displayed in the Configuration Parameters dialog box.

In R2016b, the **Code Generation** > **Interface** pane provides separate configuration controls for each data exchange interface. You can configure what your application requires and view the settings together. For example, you can configure the ASAP2 and external mode data exchange interfaces together.

Standard math library changes

These changes apply to standard math library configurations:

- When you create a model or configuration set, the default standard math library setting is ISO®/IEC 9899:1999 C (`C99 (ISO)`). Previously, the default standard math library was ISO/IEC 9899:1990 C (`C89/C90 (ANSI)`). If you are using a compiler that does not support ISO/IEC 9899:1999 C (`C99 (ISO)`), set the **Standard math library** (`TargetLangStandard`) parameter to `C89/C90 (ANSI)`.

- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.
- When you change the value of the **Language** parameter, the standard math library updates to ISO/IEC 9899:1999 C (`C99 (ISO)`) for C and ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) for C++. Previously, you adjusted the standard math library to match your programming language selection.

For more information, see Configure Standard Math Library for Target System and Standard math library.

Compatibility Considerations

As of R2016b, if you create a model or open an existing model with a script that creates a configuration set without setting the standard math library parameter `TargetLangStandard` explicitly, the parameter defaults to ISO/IEC 9899:1999 C (`C99 (ISO)`). If the specified toolchain is not compatible with that standard math library, a warning occurs during code generation and the build process continues. To avoid the warning, set `TargetLangStandard` to a standard math library that is compatible with your toolchain.

For more information, see Standard math library and Toolchain.

SupportVariableSizeSignals not checked against efficiency objectives

In R2016b, when the Code Generation Advisor checks your model configuration settings against code generation efficiency objectives, it does not consider the parameter **Support: variable-size signals** (`SupportVariableSizeSignals`).

Use default installation folder on Windows system with ReFS file system

In previous releases, on Windows systems, the code generator relied on 8.3 name or short file name generation to operate from the default installation folder (for example, `C:\Program Files\MATLAB\R2015b`).

The Windows ReFS (Resilient File System) does not permit 8.3 name or short file name generation. ReFS differs from Windows NTFS (New Technology File System), which–by default–provides short file name support.

To support the default MATLAB installation folder on Windows systems with the ReFS file system or when NTFS short file name support is disabled, the code generation software maps a drive corresponding to the MATLAB installation folder.

For more information, see Enable Build Process When Folder Names Have Spaces.

Deployment

Simulink Coder Target Support Packages: Generate code for NXP Freedom boards and STMicroelectronics Nucleo boards

You can use the following new support packages to generate code for the NXP Freedom boards and the STMicroelectronics Nucleo boards.

·   Simulink Coder Support Package for NXP FRDM-KL25Z Board

·   Simulink Coder Support Package for NXP FRDM-K64F Board.

·   Simulink Coder Support Package for STMicroelectronics Nucleo Boards User Guide

Generate code for STMicroelectronics Nucleo boards

You can use the Simulink Coder Support Package for STMicroelectronics Nucleo Boards to generate code for these STMicroelectronics Nucleo boards:

·   STM32 Nucleo F031K6

·   STM32 Nucleo F103RB

·   STM32 Nucleo F302R8

·   STM32 Nucleo F401RE

·   STM32 Nucleo L053R8

·   STM32 Nucleo L476RG

You can use processor-in-the-loop (PIL) execution to verify generated code that you deploy to all the supported Nucleo boards (except NUCLEO-F031K6 due to memory constraint) with an Embedded Coder license. By using PIL with hardware, you can more effectively generate code for your hardware by profiling speed and algorithm performance.

Support for I2C and PWM blocks for FRDM-KL25Z board

You can use the I2C Master Read and I2C Master Write blocks from the Simulink Coder Support Package for NXP™ FRDM-KL25Z Board library for reading and writing data from and to an I2C slave device.

To generate square waveform on the specified output pin, use the PWM Output block from the library.

Support for new blocks for FRDM-K64F board

From the Simulink Coder Support Package for NXP FRDM-K64F Board block library, you can use the following blocks.

- I2C Master Read and I2C Master Write blocks for reading and writing data from and to an I2C slave device.
- Push Button block to read the logical state of a push button.
- FXOS8700CQ 6–Axes Sensor block to measure linear acceleration and magnetic field along the X, Y, and Z axes.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2016a

Version: 8.10

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Variants: Generate code for active variant choice as specified with Variant Sink and Variant Source blocks

Previously, you used model variants and variant subsystems to make parts of a model conditional. In R2016a, you can make parts of a model conditional without placing blocks inside variant subsystems or model variants. A Variant Source block enables variant choices at the source of a signal. For the Variant Source block, you can specify one or no active input port. A Variant Sink block enables variant choices at the destination of a signal. For the Variant Sink block, you can specify one or no active output port. During simulation, Simulink ignores blocks that connect to inactive ports.

When you generate code, you generate code for only the active variant choice. If you use Embedded Coder, you can generate code with preprocessor conditionals that defer the choice of active variant until compilation time. You can also generate preprocessor conditionals that allow for no active variant choice.

If you use Embedded Coder, see Compile-Time Variants: Generate compiler directives (#if) for variant choices specified with Variant Source and Variant Sink blocks and Represent Variant Source and Sink Blocks in Generated Code for more information.

Protected Model Callbacks: Define callbacks for customized protected models

Customize the behavior of your protected model by using protected model callbacks. You can specify code to execute when a user views, simulates, or generates code for the protected model. If you are using a protected model, you cannot view or modify a callback.

Callback objects specify:

- The code to execute for the callback. The code can be a string of MATLAB commands or a script on the MATLAB path. The code can include protected model functions or any MATLAB command that does not require loading the model. You can use the `Simulink.ProtectedModel.getCallbackInfo` function in callback code to get information on the protected model. The function provides the protected model name and the names of submodels. If the callback is specified for `'CODEGEN'` functionality

and a `'Build'` event, the function provides the target identifier and model code interface type (`'Top model'` or `'Model reference'`).

- The event that triggers the callback. The event can be `'PreAccess'` or `'Build'`.

- The protected model functionality that the event applies to. The functionality can be `'CODEGEN'`, `'SIM'`, `'VIEW'`, or `'AUTO'`. If you select `'AUTO'`, and the event is `'PreAccess'`, the callback is applied to each functionality. If you select `'AUTO'`, and the event is `'Build'`, the callback is applied only to `'CODEGEN'` functionality. If no functionality is selected, the default behavior is `'AUTO'`.

- The option to override the protected model build. This option applies only to `'CODEGEN'` functionality.

To create a protected model with callbacks:

1 Define `Simulink.ProtectedModel.Callback` objects for each callback.

2 To create your protected model, call the `Simulink.ModelReference.protect` function. To specify a cell array of callbacks to include in the model, use the `'Callbacks'` option.

For example, to create a protected model that specifies a callback for simulation:

```
callbackForSim = Simulink.ProtectedModel.Callback('PreAccess', ...
'SIM', 'disp(myTestSim)')
Simulink.ModelReference.protect('myModel','Callbacks',{callbackForSim});
```

When you simulate the protected model, the callback is triggered before extraction of the simulation MEX-file:

```
sim('myModel')
```

```
myTestSim
```

To create a protected model that specifies a callback for code generation:

```
callbackForCodeGen = Simulink.ProtectedModel.Callback('Build', ...
'CODEGEN', 'disp(myTestCodeGen)')
Simulink.ModelReference.protect('myModel', 'Mode', 'CodeGeneration',...
'Callbacks', {callbackForCodeGen});
```

Configure the callback to specify the override option:

```
callbackForCodeGen.OverrideBuild = true
```

4-3

When you generate code for the protected model, the callback is triggered during the build of the ERT target. The build does not occur due to the override defined in the callback:

```
rtwbuild('myModel')

myTestCG
```

For more information, see Define Callbacks for Protected Model.

Simulink Coder Student Access: Obtain Simulink Coder as student-use add-on product or with MATLAB Primary and Secondary School Suite

Starting with R2016a, Simulink Coder is available for purchase as an add-on product for student-use software: MATLAB and Simulink Student Suite™ and MATLAB Student. Student-use software provides the same tools that professional engineers and scientists use. Students use the software to develop skills that help them excel in courses and prepare for careers.

Starting with R2016a, Simulink Coder is included in the MATLAB Primary and Secondary School Suite.

Model Block Virtual Buses: Interface to Model blocks by using virtual buses, reducing data copies in the generated code

In Simulink, you can create virtual bus signals to exchange signal data between a referenced model and the parent model. You can use a virtual bus as an input to the Model block or as a root-level output of the referenced model.

Previously, Simulink converted the virtual bus to a nonvirtual bus. In the code that you generated from the parent model, the parent model algorithm passed the input signal data to the referenced model `step` or `output` function as a structure. The parent algorithm copied the individual outputs of the upstream block calculations to the structure fields before calling the referenced model function. Similarly, the parent algorithm created and passed a separate structure to store the bus output of the referenced model.

In R2016a, the models exchange the signal data through multiple variables or pointers, each corresponding to a signal element of the bus, instead of a structure. This interface improves the efficiency of the generated code by eliminating the memory consumption of

the structure. The code appears and functions as it would if you used multiple signal lines instead of a virtual bus.

For example, suppose that you created a parent model `myTopModel` and a referenced model `mySubModel` in R2015b.
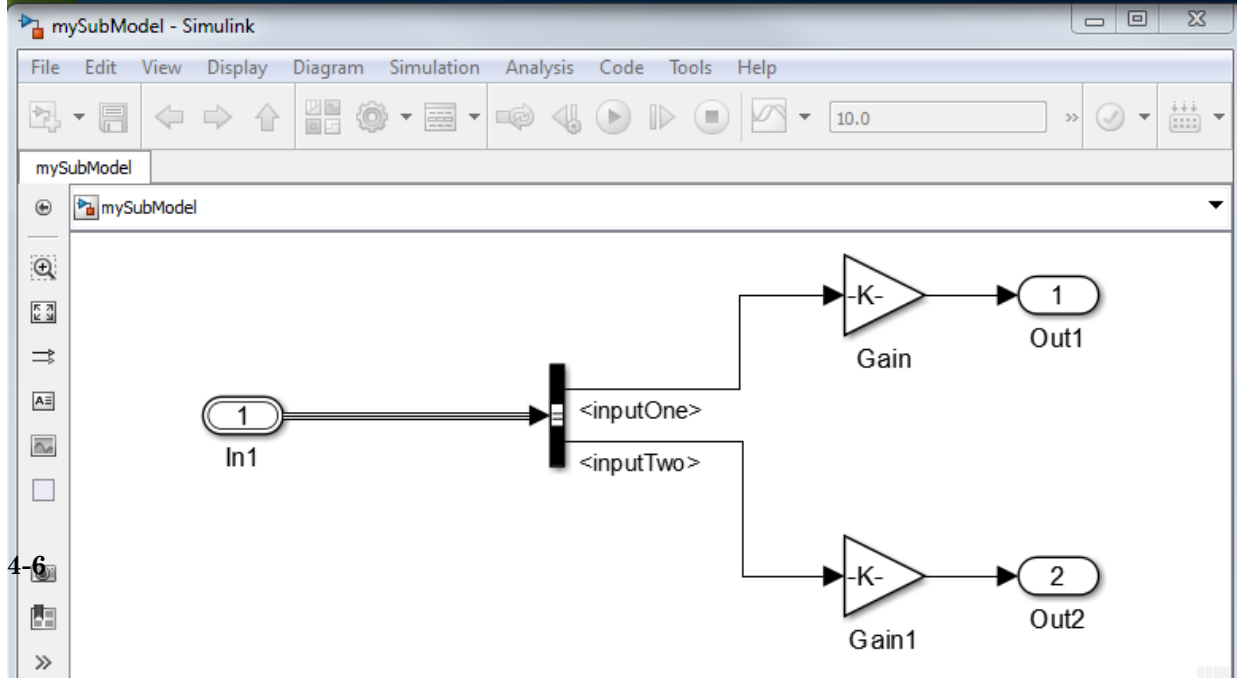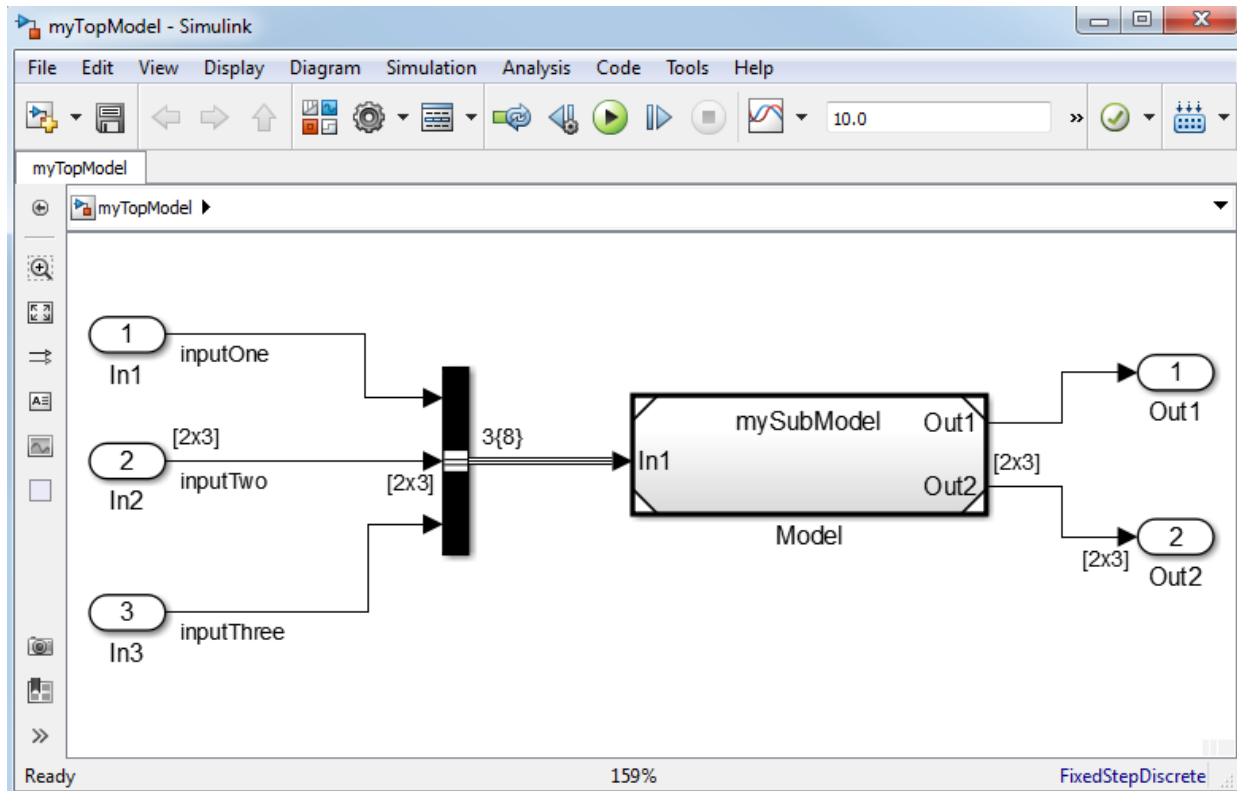
In R2015b and in R2016a, when you use a bus signal as the input to a referenced model, you must use a `Simulink.Bus` object as the output data type of the Inport block in the referenced model. Suppose that you created a bus object named `myBusType` in the base workspace.

When you generated code from the parent model, the generated algorithm copied the signal data from the Inport blocks to the fields of a local structure variable, `rtb_BusConversion_InsertedFor_M`, and passed the structure to the referenced model `step` function.

```
/* Model step function */
void myTopModel_step(void)
{
  myBusType rtb_BusConversion_InsertedFor_M;
  int32_T i;

  rtb_BusConversion_InsertedFor_M.inputOne = myTopModel_U.inputOne;

  for (i = 0; i < 6; i++) {
    rtb_BusConversion_InsertedFor_M.inputTwo[i] = myTopModel_U.inputTwo[i];
  }

  rtb_BusConversion_InsertedFor_M.inputThree = myTopModel_U.inputThree;

 mySubModel(&rtb_BusConversion_InsertedFor_M, &myTopModel_Y.Out1,
            &myTopModel_Y.Out2[0]);
}
```

The code is inefficient because:

- The local structure variable consumes redundant memory for storing the input signal data, including all of the elements of the nonscalar signal `inputTwo`.
- Even though the referenced model algorithm does not require the signal `inputThree`, the structure consumes memory for storing the field `inputThree`.

In R2016a, the parent model algorithm passes the signals `inputOne` and `inputTwo` to the referenced model as individual arguments. The code does not allocate memory for a structure variable.

```
void myTopModel_step(void)
{
```

```
  mySubModel(&myTopModel_U.inputOne, &myTopModel_U.inputTwo[0],
               &myTopModel_Y.Out1, &myTopModel_Y.Out2[0]);
}
```

In general, a virtual bus is a modeling convenience that does not affect the generated code. To package signals into a structure in the generated code, use a nonvirtual bus.

For information about changes to modeling in Simulink, including information about how to upgrade models to R2016a, see Virtual Bus Signals Across Model Reference Boundaries: Use virtual bus signals as inputs or outputs of a referenced model.

Compatibility Considerations

In R2015b and in R2016a, the code that you generate from a model represents root-level input and output virtual buses as structures. In R2016a, when you generate code from a parent model, the referenced model `step` or `output` function exchanges virtual bus signal data by passing individual arguments instead of structures. When you use a model as a referenced model, the generated code algorithm has a different interface than it does when you generate code directly from the model.

For example, suppose that in the model `mySubModel` you set **Configuration Parameters** > **Code Generation** > **Interface** > **Code interface packaging** to `Reusable function`. In R2016a, if you generate code from `mySubModel` instead of `myTopModel`, the generated `step` function uses a different interface:

```
extern void mySubModel_step(RT_MODEL_mySubModel_T *const mySubModel_M);
```

The structure type `RT_MODEL_mySubModel_T` contains a substructure `ModelData`, which contains a substructure `inputs` of the type `ExtU_mySubModel_T`. The structure type `ExtU_mySubModel_T` contains a substructure `In1` of the type `myBusType`.

```
typedef struct {
  myBusType In1;
} ExtU_mySubModel_T;
```

To generate consistent interfaces that use structures whether you use the model as a referenced model or as a standalone model, use nonvirtual buses instead of virtual buses. The generated code represents the nonvirtual bus signals as structures. To use nonvirtual buses:

· In root-level Inport block dialog boxes, select **Output as nonvirtual bus**.

- In root-level Outport block dialog boxes, select **Output as nonvirtual bus in parent model**.

Data, Function, and File Definition

Tolerance of data type mismatch between bus elements and tunable structure fields

In Simulink, you can use a MATLAB structure to initialize the elements of a bus signal, or to drive a bus signal from a Constant block. Previously, if you configured the structure to appear in the generated code as a tunable global structure, you matched the numeric data types of the fields with those of the corresponding bus elements. If you did not match the data types, the code generator displayed an error.

In R2016a, the generated code algorithm uses explicit typecasts to reconcile the data type mismatches. As you create and experiment with a model, you can use default doubles to set the structure field values, and specify data types only for the bus elements.

To improve performance and readability of the generated code by avoiding typecasts, floating-point structure fields, and field-by-field assignment operations, match the data types of tunable structure fields with those of the corresponding bus elements. See Control Data Types of Initial Condition Structure Fields.

In R2016a, the Model Advisor check **Check for partial structure parameter usage with bus signals** has a new name, **Check structure parameter usage with bus signals**. Use this check to discover potential inefficient typecasts due to mismatched data types. For more information, see Check structure parameter usage with bus signals.

Model Advisor check for data type mismatches between bus elements and structure fields

In R2016a, you can generate code if the numeric data types of bus signal elements do not match those of the corresponding fields of an initial condition structure. Previously, the code generator displayed an error if the initial condition appeared in the code as a tunable global structure. For more information, see "Tolerance of data type mismatch between bus elements and tunable structure fields" on page 4-10.

The Model Advisor check **Check for partial structure parameter usage with bus signals** has a new name, **Check structure parameter usage with bus signals**. The check has a new programmatic ID, `mathworks.design.MismatchedBusParams`. Your scripts that use the old ID still work. Consider replacing the old ID with the new ID. Before you generate code from a model, use this check to discover potential inefficient

typecasts due to mismatched data types. For more information, see Check structure parameter usage with bus signals.

Simplified method to apply storage classes to signals and states

Previously, you applied storage classes and custom storage classes to signals and states by selecting a package and a storage class in the Signal Properties dialog box or on the **State Attributes** tab in a block dialog box. With the default package, `None`, you could select one of three built-in storage classes. You could select a package to enable custom storage classes. For example, in the Signal Properties dialog box, you selected a package and storage class by using the drop-down lists **Package** and **Storage class**.

If you set the package to `None`, you could select a storage type qualifier for the variable in the generated code.

In R2016a, you use a simplified method to apply storage classes and custom storage classes to signals and states. To apply storage classes, see Control Signals and States in Code by Applying Storage Classes. To apply custom storage classes, which require an Embedded Coder license, see Control Data Representation by Applying Custom Storage Classes.

Storage Classes

In R2016a, the drop-down list **Signal object class** replaces the drop-down list **Package**. The default value for this new list is `Simulink.Signal`, which allows you to select storage classes and custom storage classes from the built-in package `Simulink`. Use the new list to choose a different class of signal object, for example `mpt.Signal`. You can then select a custom storage class that the package `mpt` defines.

Storage Type Qualifiers

In R2016a, if a signal or state does not already use a code generation storage type qualifier, the option **Storage type qualifier** does not appear in the Signal Properties dialog box or on the **State Attributes** tab in the block dialog box.

To apply storage type qualifiers, use custom storage classes and memory sections.

Embedded Signal Objects

When you upgrade a model from a previous release to R2016a, signals and states for which you previously set **Package** to `None` and **Storage class** to a storage class other

than `Auto` acquire an embedded `Simulink.Signal` object. Use the functions `get_param` and `set_param` to interact with the embedded signal object through the programmatic parameters `SignalObject` (for block output ports) and `StateSignalObject` (for block states).

You can also continue to use the programmatic parameters `StorageClass` and `StateStorageClass` to apply storage classes. When you use these parameters, the new storage class applies to the embedded signal object. You can apply a basic storage class, such as `ExportedGlobal`, by writing fewer lines of code. To apply a custom storage class, interact with the embedded signal object instead.

Compatibility Considerations

- In the Simulink Preferences dialog box, the **Data Management Defaults** pane no longer appears.

  For the **Package** option that you previously set through the **Data Management Defaults** pane, the equivalent programmatic parameter `DefaultDataPackage` will be removed in a future release. In R2016a, setting the parameter generates a warning. If you wrote scripts that use this parameter, remove the parameter from the scripts. For example, if your script contains this line of code:

  ```
  set_param(0,'DefaultDataPackage','mpt')
  ```

- Unless you already set the option value in a previous release, the option **Storage type qualifier** is hidden in the Signal Properties dialog box and on the **State Attributes** tab in block dialog boxes.

Conflict between different storage classes applied to same signal

Previously, you could apply the storage class `SimulinkGlobal` to a signal line, and then apply a storage class other than `Auto` to a downstream or upstream line that represented the same signal data.

For example, suppose you applied the storage class `SimulinkGlobal` to a signal line that you connected to an Outport block inside a subsystem. Outside the subsystem, you could apply the storage class `ImportedExtern` to the signal line that the corresponding output port drives. When you generated code from the model, the signal data used the storage class `ImportedExtern`.

In R2016a, the model generates an error.

Compatibility Considerations

If you open a model that you created in a previous version, the model generates an error if you previously configured conflicting storage classes for a signal.

To resolve the error, set the storage class of the signal line from `SimulinkGlobal` to `Auto`. The signal data uses the other storage class.

Alternatively, set the storage class from `SimulinkGlobal` to the other storage class. If you later want to change the storage class for the signal data, you must remember to change the storage classes for both signal lines.

Visibility and functionality changes for programmatic properties of data objects

With objects of the class `Simulink.CoderInfo`, you can specify code generation settings for data objects, which include objects of the classes `Simulink.Parameter` and `Simulink.Signal`. The table summarizes changes to the programmatic properties of `Simulink.CoderInfo` objects.

Behavior Change	For These Properties
If you set the property `StorageClass` to `'Auto'`, these properties are hidden. Attempting to set them to a value other than the default value generates an error.	• `Alias` • `Alignment` • `TypeQualifier`
If you set any of these properties to a value other than the default value, setting the property `StorageClass` to `'Auto'` generates a warning. The object sets the property value to the default value.	
If you do not set the property `StorageClass` to `'Custom'`, setting the property `CustomStorageClass` to a value other than the default value generates a warning.	`CustomStorageClass`

Compatibility Considerations

If you have scripts that set the properties of `Simulink.CoderInfo` objects, make sure that the scripts do not generate unnecessary warnings or errors in R2016a. For example, before you set the value of the property `CustomStorageClass`, set the value of the property `StorageClass` to `'Custom'`.

Code Generation

Simplified Configuration Parameters: Configure model more easily via streamlined code generation panes

In the Configuration Parameters dialog box, streamlined category panes display only configuration parameters that you are most likely to use when configuring your model for code generation.

The category panes, previously referred to as the Category view, are now available on the **Commonly Used Parameters** tab. The **All Parameters** tab, previously referred to as the List view, provides the complete list of parameters in the model configuration set.
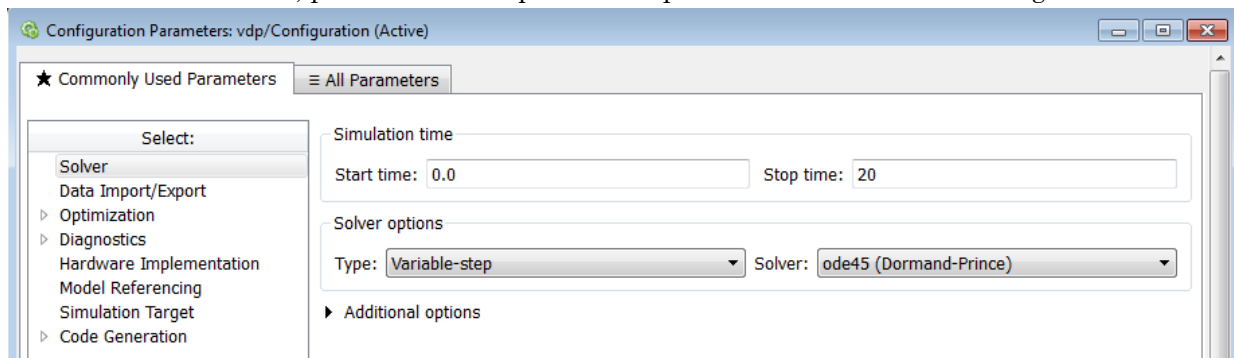
Compatibility Considerations

Following are the configuration parameters that have moved to the **All Parameters** tab or moved to a different pane.

Note Parameters that are removed from a pane are still available for configuration on the **All Parameters** tab. To locate a parameter on this tab, use either the search box or the **Category** filter.

Code Generation Pane

The following are moved to the **All Parameters** tab:

- **Ignore custom storage classes** parameter
- **Ignore test point signals** parameter
- **Validate** button for **Toolchain** parameter

Code Generation > Interface Pane

The following parameters are moved to the **All Parameters** tab:

- **Standard math library**
- **Support: non-inlined S-functions**
- **Multiword type definitions**
- **Maximum word length**
- **Use dynamic memory allocation for model initialization**
- **Classic call interface**
- **Single output/update function**
- **Terminate function required**
- **Combine signal/state structures**
- **Internal data visibility**
- **Internal data access**
- **Generate destructor**
- **Use dynamic memory allocation for model block instantiation**
- **MAT-file logging**
- **MAT-file variable name modifier**

Code Generation > Debug Pane

The pane is removed and its parameters are moved to the **All Parameters** tab:

- **Profile TLC**
- **Verbose build**
- **Retain .rtw file**
- **Enable TLC assertion**
- **Start TLC coverage when generating code**
- **Start TLC debugger when generating code**

Data Import/Export Pane

The **Enable live streaming of selected signal to Simulation Data Inspector** parameter is moved to the **All Parameters** tab.

The following parameters are available by clicking **Additional Parameters** at the bottom of the pane:

- **Limit data points to last**
- **Decimation**
- **Output options**
- **Refine factor**

Diagnostics Pane

The following parameter is moved to the **All Parameters** tab:

- **Solver data inconsistency**

Diagnostics > Data Validity Pane

The following parameters are moved to the **All Parameters** tab:

- **Array bounds exceeded**
- **Model verification block enabling**
- **Check preactivation output of execution context**
- **Check runtime output of execution context**
- **Check undefined subsystem initial output**
- **Detect multiple driving blocks executing at the same time step**
- **Underspecified initialization detection**

Diagnostics > Saving Pane

The pane is removed and its parameters are moved to the **All Parameters** tab:

- **Block diagram contains disabled library links**
- **Block diagram contains parameterized library links**

Diagnostics > Solver Pane

The following parameters are moved to the **Diagnostics > Sample Time** pane:

- **Sample hit time adjusting**
- **Unspecified inheritability of sample time**

The following parameter is moved to the **Diagnostics > Compatibility** pane:

- **SimState object from earlier release**

Optimization Pane

The following parameters are moved to the **All Parameters** tab:

- **Remove code from floating-point to integer conversions with saturation that maps NaN to zero**
- **Compiler optimization level**
- **Verbose accelerator builds**
- **Implement logic signals as Boolean data (vs. double)**
- **Block reduction**
- **Conditional input branch execution**
- **Use memset to initialize floats and doubles to 0.0**

Optimization > Signals and Parameters Pane

The following parameters are moved to the **All Parameters** tab:

- **Signal storage reuse**
- **Enable local block outputs**
- **Reuse local block outputs**
- **Optimize global data access**
- **Reuse global block outputs**
- **Eliminate superfluous local variables (Expression folding)**
- **Simplify array indexing**

Simulation Target Pane

The following parameters are moved to the **All Parameters** tab:

- **Echo expressions without semicolons**
- **Simulation target build mode**
- **Ensure responsiveness**
- **Generate typedefs for imported bus and enumeration types**
- **Ensure memory integrity**

Simulation Target > Custom Code Pane

The pane is removed and its parameters are moved to the **Simulation Target** pane:

- **Header file**
- **Initialize function**
- **Source file**
- **Terminate function**
- **Parse custom code symbols**
- **Include directories**
- **Libraries**
- **Source files**
- **Defines**

Simulation Target > Symbols Pane

The pane is removed and its parameter is moved to the **Simulation Target** pane:

- **Reserved names**

Add macro definitions to custom code

Previously, to add macro definitions—tokens with or without values submitted on the compiler command line—for toolchain approach builds, you directly modified the compiler command line in **Configuration Parameters** > **Code Generation** > **Build process**. In this section of the **Code Generation** pane, you set the **Build configuration** parameter value to `Specify` and added macro definitions to the compiler options. With the new **Configuration Parameters** > **Code Generation** > **Custom Code** > **Additional Build Information** > **Defines** parameter, you can add these definitions independent of the toolchain selection. This parameter applies for toolchain approach builds and template makefile approach builds.

The new **Defines** parameter lets you add a list of macro definitions to the compiler command line. Specify the parameters with a space-separated list of macro definitions. If a makefile is generated, these macro definitions are added to the compiler command line in the makefile. The list can include simple definitions (for example, -DDEF1), definitions with a value (for example, -DDEF2=1), and definitions with a space in the value (for example, -DDEF3="my value"). Definitions can omit the -D (for example, -DFOO=1 and FOO=1 are equivalent). If the toolchain uses a different flag for definitions, the code generator overrides the -D and uses the appropriate flag for the toolchain.

For more information, see Code Generation Pane: Custom Code: Additional Build Information: Defines.

Faster generated code for linear algebra in the MATLAB Function block

To improve the simulation speed of MATLAB Function block algorithms that call certain linear algebra functions, the simulation software can call LAPACK functions. In R2016a, if you use Simulink Coder to generate C/C++ code for these algorithms, you can specify that the code generator produce LAPACK function calls. If you specify that you want to generate LAPACK function calls, and the input arrays for the linear algebra functions meet certain criteria, the code generator produces calls to relevant LAPACK functions. The code generator uses the LAPACKE C interface.

LAPACK is a software library for numerical linear algebra. MATLAB uses this library in some linear algebra functions, such as `eig` and `svd`. Simulink uses the LAPACK library that is included with MATLAB. Simulink Coder uses the LAPACK library that you specify. If you do not specify a LAPACK library, the code generator produces code for the linear algebra function instead of generating a LAPACK call.

To specify that you want to generate LAPACK function calls and link to a specific LAPACK library, see Speed Up Linear Algebra in Code Generated from a MATLAB Function Block.

Build button removed from Configuration Parameters dialog box

The **Build** / **Generate Code** button is no longer available on the **Code Generation** pane in the Configuration Parameters dialog box.

Compatibility Considerations

To initiate code generation and the build process, press **Ctrl-B** or, on the Simulink Editor toolbar, click the **Build Model** icon.
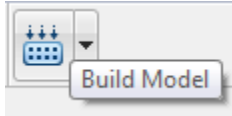
Deployment

Hardware implementation parameters enabled by default

In R2016a, the **Enable hardware specification** button is removed from the **Configuration Parameters** > **Hardware Implementation** pane. By default, the parameters on the pane are enabled.

Simulink Coder Support Package for ARM Cortex-Based VEX Microcontroller

From R2016a, you can use the Simulink Coder Support Package for ARM® Cortex®-Based VEX® Microcontroller to generate, build, and deploy code to the VEX microcontroller. This support package was earlier called Simulink Support Package for ARM Cortex-based VEX Microcontroller from its inception in R2014a until R2015b. However, you can use this support package on Embedded Coder to use some of the Embedded Coder features.

See Install Support for Simulink Coder Support Package for ARM Cortex-based VEX Microcontroller.

For more information, see ARM Cortex-Based VEX Microcontroller.

Performance

Removal of Minimize data copies between local and global variables parameter

In R2016a, there is no longer a **Minimize data copies between local and global variables** parameter. The code generator now generates code as if this parameter is set to `off`. To fine-tune this setting with an Embedded Coder license, use the **Optimize global data access** parameter. For more information, see Optimize Global Variable Usage.

Previously, in the Configuration Parameters dialog box, this parameter was on the **Optimization** > **Signals and Parameters** pane.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015aSP1

Version: 8.8.1

Bug Fixes

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015b

Version: 8.9

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Support for C++ code generation in protected models

In R2015b, you can create a protected model that supports C++ code generation. Protected models that support C++ are subject to the same requirements as previously supported protected models.

Reusable code for subsystems containing Stateflow charts

In R2015b, you can generate reusable code for a subsystem that contains a Stateflow chart or is a Stateflow chart. For the subsystem, the code generator creates a standalone function in the shared utilities folder. The generated code of multiple reference models can then call this function. You cannot create shared code for subsystems or Stateflow charts that use machine-parented data, import or export graphical functions, or contain atomic subcharts.

This enhancement reduces code size and ROM consumption. For more information, see Code Reuse For Subsystems Shared Across Models.

Header file change for model containing messages in Stateflow charts

In R2015b, if your model contains one or more Stateflow charts that use messages to communicate within or between Stateflow charts, the code generator creates a `builtin_typeid_types.h` file. For more information, see Header Dependencies When Interfacing Legacy/Custom Code with Generated Code. For more information on messages, see How Messages Work in Stateflow Charts.

Type definitions in rapid accelerator mode

Previously, if you ran a model containing Stateflow and MATLAB function blocks in rapid accelerator mode, you could include a header file in the **Custom Code** pane. The header file contained your own definitions of enumerated, bus, or alias data types.

In R2015b, to get compilable code in rapid accelerator mode, you must use the code generator definitions. You cannot include a header file with your own type definitions. To use the code generator definitions, open the Configuration Parameters dialog box. On the **Simulation Target** pane, select **Generate typedefs for imported bus and enumeration types**.

You can continue to include a header file with your own definitions of enumerated, bus, or alias data types for use in normal and accelerator simulation modes.

Data, Function, and File Definition

Configuration parameter Inline parameters name and functionality change

The configuration parameter **Optimization** > **Signals and Parameters** > **Inline parameters** has a new name, **Default parameter behavior**. Previously, **Inline parameters** was a check box. In R2015b, **Default parameter behavior** is a drop-down list.

At the command prompt, use the name `DefaultParameterBehavior` to access the configuration parameter **Default parameter behavior**. Your scripts that use the names `InlineParameters` and `InlineParams` still work.

These tables compare the settings for the original parameter name, **Inline parameters**, with the settings for the new parameter name, **Default parameter behavior**.

In the Configuration Parameters Dialog Box

Inline parameters	Default parameter behavior
Selected	`Inlined`
Cleared	`Tunable`

At the Command Prompt

`InlineParameters`	`DefaultParameterBehavior`
`'on'`	`'Inlined'`
`'off'`	`'Tunable'`

Previously, constant folding eliminated the code that represented blocks that used constant sample time. If the code generator could not fold the block code, or if you selected settings to disable constant folding, the block code appeared in the model initialization function. However, if the parameters of a block were tunable, the block in the model did not use constant sample time. The block code instead appeared in the model `step` or output functions. Therefore, constant sample time indicated the block code placement.

In R2015b, constant sample time does not directly indicate block code placement. Block parameters are tunable during simulation regardless of the setting of **Default**

parameter behavior. You can still control block parameter tunability in the generated code by adjusting the setting for **Default parameter behavior** and by applying storage classes to parameter data objects. The placement of code for blocks that have constant output values still depends on the tunability of the block parameters in the generated code. However, these blocks use constant sample time in the model regardless of parameter tunability.

When you use the configuration parameter **Code Generation** > **System target file** to switch to an ERT-based code generation target from a target that is not ERT-based, the setting for **Default parameter behavior** switches from `Tunable` to `Inlined`. If necessary, you can then specify the parameter as `Tunable`.

Compatibility Considerations

If you use scripts that change code generation targets, confirm that the scripts do not alter the setting for **Default parameter behavior**.

Code Generation

MinGW-w64 Compiler Support: Compile MEX files on 64-bit Windows with free compiler

You can now use the MinGW-w64 compiler from TDM-GCC to build model code on 64-bit Windows® hosts. To download and install the compiler, see Install MinGW-w64 Compiler.

If you select a code generation target that supports toolchain controls, such as `grt.tlc` or `ert.tlc`, your model builds can use a MinGW compiler toolchain. Select the toolchain on the **Code Generation** pane in the Configuration Parameters dialog box.

Build process

Toolchain settings

Toolchain:	MinGW64 v4.x | gmake (64-bit Windows) ▼ Validate

Build configuration:	Faster Builds ▼ Hide settings
	Minimize compilation and linking time

Tool	Options
C Compiler	-c $(ANSI_OPTS) -O0
Linker	-Wl,-rpath,"$(MATLAB_ARCH_BIN)",-L"$(MATLAB_ARCH_BIN)"
Shared Library Linker	-shared -Wl,-rpath,"$(MATLAB_ARCH_BIN)",-L"$(MATLAB_ARCH_BIN)" -Wl,--no-undefined
C++ Compiler	-c $(CPP_ANSI_OPTS) -O0
C++ Linker	-Wl,-rpath,"$(MATLAB_ARCH_BIN)",-L"$(MATLAB_ARCH_BIN)"
C++ Shared Library Linker	-shared -Wl,-rpath,"$(MATLAB_ARCH_BIN)",-L"$(MATLAB_ARCH_BIN)" -Wl,--no-undefined
Archiver	ruvs

Internationalization: Generate and review code containing mixed languages for different locales

In R2015b, the code generator introduces support for non-US-ASCII characters in compilable portions of generated source code. The code generator processes strings without loss of information or character corruption by replacing unrepresented characters of the user default encoding with an escape sequence of the form `&#xcode-`

unit;. *code-unit* is the hexadecimal value for the unrepresented character. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `ア`. Cases where escape sequence replacements occur include:

- Strings representing model parameters, block names, and signal names that appear in generated code comments.
- Output variables representing signal names and block names on block paths logged to MAT- files.
- Variables representing block names on block paths logged to C API files *model*_capi.c (or .cpp) and *model*_capi.h.

When generating HTML code reports, the code generator converts replacement character escape sequences with original strings.

An exception to the character escape sequence replacement scheme is variables and function names in Target Language Compiler (.tlc) files. These files support user default encoding only. To use the compiler to produce international custom generated code that is portable, use the 7-bit ASCII character set when naming variables and functions.

For more information, see Internationalization and Code Generation.

Hardware Implementation Selection: Quickly generate code for popular embedded processors

Specification of hardware configurations has been simplified. Top-level Configuration Parameters dialog box panes, **Run on Target Hardware** and **Coder Target**, have been removed. Parameters previously available on those panes now appear on the **Hardware Implementation** pane. A parameter has also moved from the **Code Generation** pane to the **Hardware Implementation** pane.

This list summarizes the R2015b changes and new behavior:

- By default, the **Hardware Implementation** pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only.
- If you use Simulink without a Simulink Coder license, initially parameters on the **Hardware Implementation** pane are disabled. To enable them, click **Enable hardware specification**. The parameters remain enabled for the current MATLAB session.

- By default, the **Hardware board** list includes: `None` or `Determine by Code Generation system target file`, and `Get Hardware Support Packages`. After installing a hardware support package, the list also includes corresponding hardware board names.

- If you select a hardware board name, parameters for that board appear in the dialog box display.

- Lists for the **Device vendor** and **Device type** parameters have been updated to reflect hardware that is available on the market. The default **Device vendor** and **Device type** are `Intel` and `x86-64 (Windows64)`, respectively.

- If Simulink Coder is installed, the revised **Hardware Implementation** pane identifies the system target file that you selected on the **Code Generation** pane.

- A **Device details** option provides a way to display parameters for setting details such as number of bits and byte ordering.

- To specify target hardware for a Simulink support package, select a value from **Configuration Parameters** > **Hardware Implementation** > **Hardware board**. Before R2015b, you selected **Tools** > **Run on Target Hardware** > **Prepare to run**. Then, you selected a value from **Configuration Parameters** > **Run on Target Hardware** > **Target hardware**.

- To specify target hardware for an Embedded Coder support package, select a value from **Configuration Parameters** > **Hardware Implementation** > **Hardware board**. Before R2015b, you selected a value from **Configuration Parameters** > **Code Generation** > **Target hardware**.

- The **Test hardware** section was removed. Configure test hardware from the Configuration Parameters list view. Set `ProdEqTarget` to `off`, which enables parameters for configuring test hardware details.

- If you set **Configuration Parameters** > **Code Generation** > **System target file** to `ert.tlc`, `realtime.tlc`, or `autosar.tlc`, the default setting for **Configuration Parameters** > **Hardware Implementation** > **Hardware board** is `None`. If you set **System target file** to value other than `ert.tlc`, `autosar.tlc`, or `realtime.tlc`, the default setting for **Hardware board** is `Determine by Code Generation system target file`.

For more information, see Hardware Implementation Pane.

Compatibility Considerations

Starting in R2015b:

- By default, the **Hardware Implementation** pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only. To view parameters for setting details, such as number of bits and byte ordering, click **Device details**.

- The following devices appear on the **Hardware Implementation** pane only for models that you create with a version of the software earlier than R2015b. These devices are considered legacy devices.

  Generic, 32-bit Embedded Processor
  Generic, 64-bit Embedded Processor (LP64)
  Generic, 64-bit Embedded Processor (LLP64)
  Generic, 16-bit Embedded Processor
  Generic, 8-bit Embedded Processor
  Generic, 32-bit Real-Time Simulator
  Generic, 32-bit x86 compatible
  Intel, 8051 Compatible
  Intel, x86–64
  SGI, UltraSPARC Iii

  In R2015b, if you open a model configured for a legacy device and change the **Device type** setting, you cannot select the legacy device again.

- Device parameter **Signed integer division rounds to** is set to `Zero` instead of `Undefined`. For some cases, numerical differences can occur in results produced with `Zero` versus `Undefined` for simulation and code generation.

  This change does not apply to legacy devices.

- To associate a new model with an existing configuration set that has the following characteristics, configure the model to use the same hardware device as the existing model.

  - The model consists of a model reference hierarchy. Models in the hierarchy use different configuration sets.

  - The existing configuration set was saved as a script and associated with a configuration set variable.

  If the code generator detects differences in device parameter settings, a consistency error occurs. To correct the condition, look for differences in the device parameter settings, and make the appropriate adjustments.

Smarter Code Regeneration: Regenerate code only when model settings that impact code are modified

Selecting the model option **Configuration Parameters** > **Code Generation** > **Generate code only** configures the model build process to generate code, without compiling and building the generated code. R2015b provides more intuitive and flexible behavior for the **Generate code only** option. In R2015b:

- Toggling the **Generate code only** option on or off between builds no longer forces regeneration of source code. For example, suppose that you clear **Generate code only** after generating code, and make no other model change that affects code generation. The next build detects that up-to-date source code is already available and compiles the code without regenerating it.

- In a model reference hierarchy, the **Generate code only** setting of the top model overrides the **Generate code only** setting of referenced models. This change relaxes the constraint that the **Generate code only** setting must be consistent within a model reference hierarchy. The change helps prevent unnecessary regeneration of referenced model code.

- If you have an Embedded Coder license, running a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation for a top model or Model block no longer requires that you clear **Generate code only**. See Embedded Coder release note Removal of Generate code only parameter restriction.

Toolchain approach with custom targets added

You can configure properties of a custom target such that the system target file is toolchain-compliant. When you select a toolchain-compliant STF from the **Code Generation** pane in the Configuration Parameters dialog box, the software recognizes toolchain compliance and provides the build process controls for the toolchain approach.

Previously, it was not possible to define toolchain compliance for custom targets. You had to use the template makefile approach to build using production targets. With toolchain approach support for custom targets, you can generate code using the toolchain approach throughout your development process from model architecture through verification.

Build configuration setting can affect setting for toolchain

When using the toolchain approach to build a model, you can configure the code generator to use a specific toolchain and build configuration. On the Configuration

Parameters dialog box, you can set values for **Code Generation** > **Toolchain** and **Code Generation** > **Build configuration**.

As of R2015b, a change to the **Build configuration** setting can affect the setting for **Toolchain**.

- Changing the **Build configuration** from any value to `Specify`, changes the default **Toolchain** value (`Automatically locate an installed toolchain`) to the value of the toolchain that was automatically located. For example, the value changes from `Automatically locate an installed toolchain` to `Microsoft Visual C++ 2012 v11.0 |(64-bit Windows)`.
- Changing the **Build configuration** from `Specify` to any other value has no effect on the **Toolchain** value.

This operation improvement synchronizes the **Toolchain** setting with the setting for **Build configuration**.

Deployment

External mode MEX-file build requires sl_services library

As of R2015b, the `mex` commands to rebuild MEX-file modules for external mode communication require linking the `sl_services` library. Examples of external mode communication modules include TCP/IP module `ext_comm` and serial module `ext_serial_win32_comm`.

Compatibility Considerations

You must update existing scripts for external mode MEX-file builds to link the `sl_services` library. For Windows, add `-lsl_services`. For Linux® or Mac, add `-lmwsl_services`. For example, here is an updated Windows command to build `ext_comm`, with the library addition in bold.

```
>> cd (matlabroot)
>> mex -setup
>> mex toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_comm.c ...
toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_convert.c ...
toolbox\coder\simulinkcoder_core\ext_mode\host\common\rtiostream_interface.c ...
toolbox\coder\simulinkcoder_core\ext_mode\host\common\ext_util.c ...
-Irtw\c\src -Irtw\c\src\rtiostream\utils ...
-Irtw\c\src\ext_mode\common ...
-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common ...
-Itoolbox\coder\simulinkcoder_core\ext_mode\host\common\include ...
-lmwrtiostreamutils -lsl_services ...
-DEXTMODE_TCPIP_TRANSPORT ...
-DSL_EXT_DLL -output toolbox\coder\simulinkcoder_core\ext_comm
```

For more information, see MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files.

Performance

Consolidation of redundant if-else and for statements in separate code regions

Previously, the code generator tried to combine adjacent `if-else` and `for` statements that shared the same condition. In R2015b, this optimization extends to `if-else` and `for` statements located in separate, noninterfering regions of the generated code. This enhancement results in:

- Reduced data copies, code size, and RAM consumption.
- Less complex code.
- Improved execution speed.

Consider the following Enabled Subsystem block named `S4`. This subsystem contains three Switch blocks named `Switch 1`, `Switch 2`, and `Switch 3`.
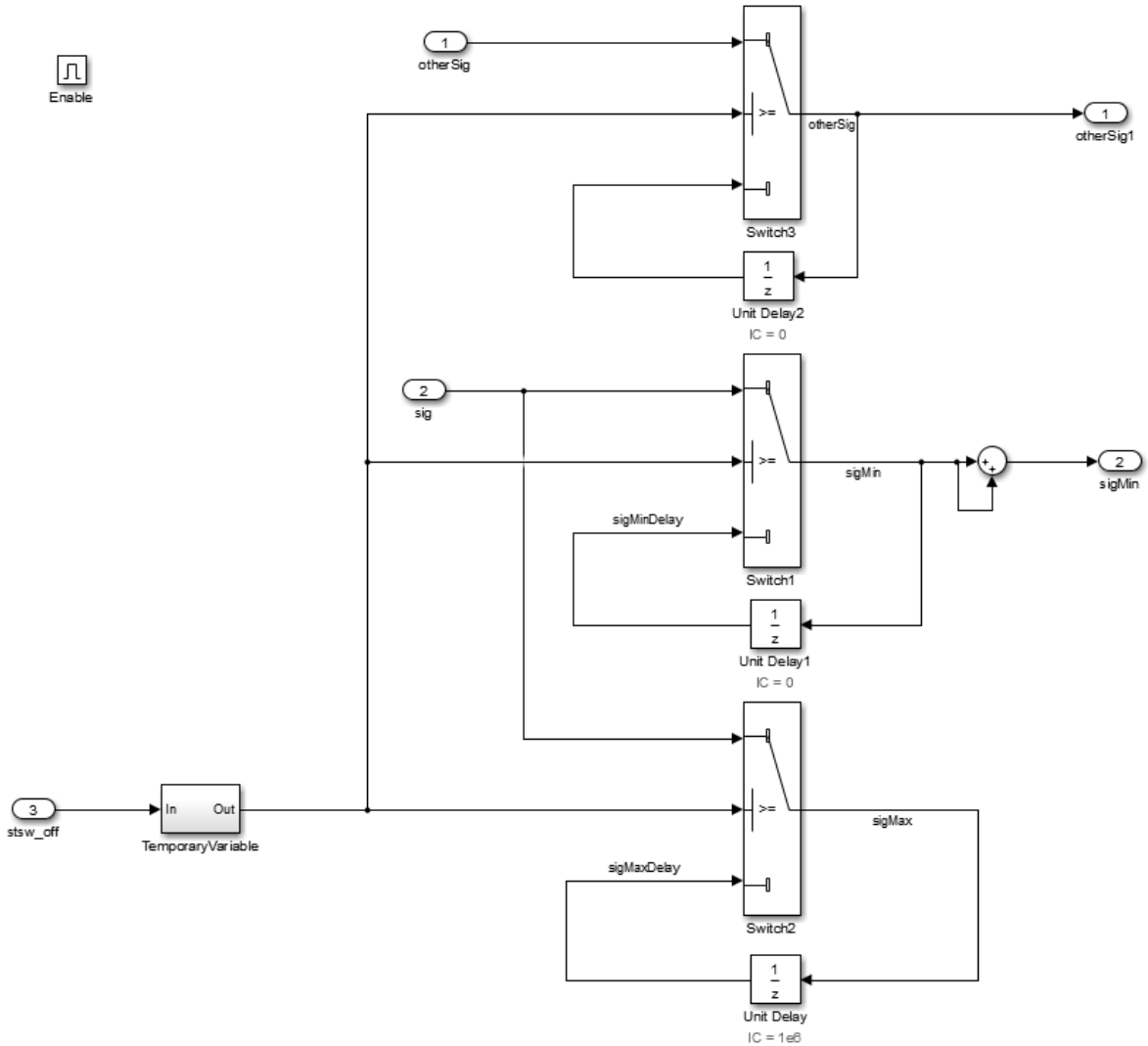
In R2015a, the code generator produced the following code:

```
void IfElseWithState_R2015a_step(void)
{
  real32_T rtb_sigMin;
```

```
  if (enable) {
    if (stsw_off) {
      rtb_sigMin = sig;
    } else {
      rtb_sigMin = IfElseWithState_R2015a_DW.UnitDelay1_DSTATE;
    }

    sigMin = rtb_sigMin + rtb_sigMin;
    if (stsw_off) {
      sigMax = sig;
      otherSigCapture = otherSig;
    } else {
      sigMax = IfElseWithState_R2015a_DW.UnitDelay_DSTATE;
      otherSigCapture = IfElseWithState_R2015a_DW.UnitDelay2_DSTATE;
    }

    IfElseWithState_R2015a_DW.UnitDelay1_DSTATE = rtb_sigMin;
    IfElseWithState_R2015a_DW.UnitDelay_DSTATE = sigMax;
    IfElseWithState_R2015a_DW.UnitDelay2_DSTATE = otherSigCapture;
  }
}
```

In R2015b, the code generator produces the following code:

```
    void IfElseWithState_step(void)
{
  real32_T rtb_sigMin;
  if (enable) {
    if (stsw_off) {
      rtb_sigMin = sig;
      sigMax = sig;
      otherSigCapture = otherSig;
    } else {
      rtb_sigMin = IfElseWithState_DW.UnitDelay1_DSTATE;
      sigMax = IfElseWithState_DW.UnitDelay_DSTATE;
      otherSigCapture = IfElseWithState_DW.UnitDelay2_DSTATE;
    }

    sigMin = rtb_sigMin + rtb_sigMin;
    IfElseWithState_DW.UnitDelay1_DSTATE = rtb_sigMin;
    IfElseWithState_DW.UnitDelay_DSTATE = sigMax;
    IfElseWithState_DW.UnitDelay2_DSTATE = otherSigCapture;
  }
}
```

In R2015a, the generated code contained two `if-else` statements because of the sum operation that followed `Switch 1`. In R2015b, there is one `if-else` statement for all three Switch blocks. The code generator combines the `if-else` statements because they share the same condition. The outcome of the addition operation has no effect on this condition.

More efficient code for multirate models

Previously, if a referenced model or atomic subsystem contained blocks that executed at different sample times, the code generator produced separate output and update functions for each sample time. In R2015b, for each sample time, the code generator produces one output and update function. This optimization increases execution speed and conserves RAM and ROM consumption.

Consider the following model named `mTopMultiRateMultiTasking`. This model contains a referenced model, `mSubMultiRateMultiTasking`, that executes at two different sample times.
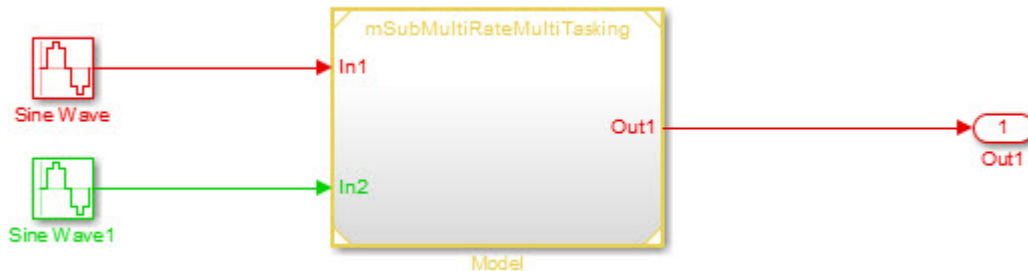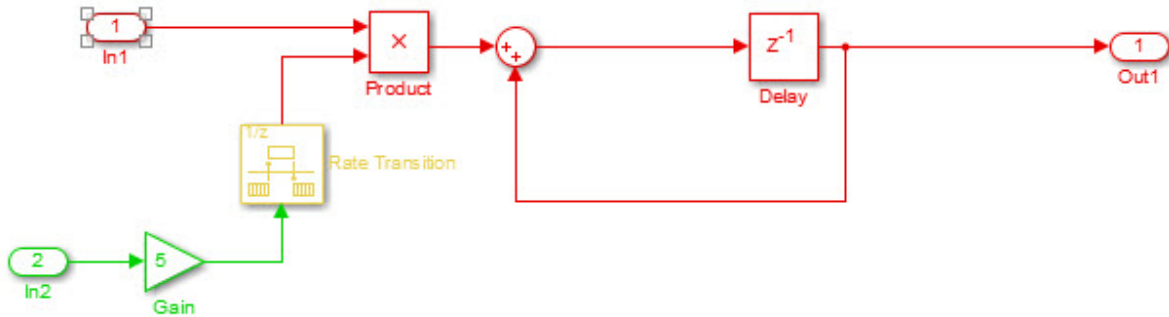
In R2015a, for `mSubMultiRateMultiTasking`, the code generator produced this code:

```
void mSubMultiRateMultiTaskingTID0(const real_T *rtu_In1, real_T *rty_Out1,
  B_mSubMultiRateMultiTasking_c_T *localB, DW_mSubMultiRateMultiTaskin_f_T
  *localDW)
{
  int_T tid = 0;
  *rty_Out1 = localDW->Delay_DSTATE;
  if (rtmIsSpecialSampleHit(1, 0, tid)) {
    localB->RateTransition = localDW->RateTransition_Buffer0;
  }

  localB->Sum = *rtu_In1 * localB->RateTransition + *rty_Out1;
  (void) (tid);
}

void mSubMultiRateMultiTaskingTID1(const real_T *rtu_In2,
  B_mSubMultiRateMultiTasking_c_T *localB)
{
  int_T tid = 1;
  localB->Gain = 5.0 * *rtu_In2;
  (void) (tid);
}

void mSubMultiRateMultiTa_UpdateTID0(B_mSubMultiRateMultiTasking_c_T *localB,
  DW_mSubMultiRateMultiTaskin_f_T *localDW)
{
  localDW->Delay_DSTATE = localB->Sum;
```

```
}

void mSubMultiRateMultiTa_UpdateTID1(B_mSubMultiRateMultiTasking_c_T *localB,
  DW_mSubMultiRateMultiTaskin_f_T *localDW)
{
  localDW->RateTransition_Buffer0 = localB->Gain;
}
```

In R2015a, the generated code contains four function calls. The first two functions
produce the output at each sample time (`tid=0` and `tid=1`). The second two functions
update the tasks at each sample time.

In R2015b, for `mSubMultiRateMultiTasking`, the code generator produces this code:

```
void mSubMultiRateMultiTaskingTID0(const real_T *rtu_In1, real_T *rty_Out1,
  B_mSubMultiRateMultiTasking_c_T *localB, DW_mSubMultiRateMultiTaskin_f_T
  *localDW)
{
  int_T tid = 0;
  *rty_Out1 = localDW->Delay_DSTATE;
  if (rtmIsSpecialSampleHit(1, 0, tid)) {
    localB->RateTransition = localDW->RateTransition_Buffer0;
  }

  localDW->Delay_DSTATE = *rtu_In1 * localB->RateTransition + *rty_Out1;
  (void) (tid);
}

void mSubMultiRateMultiTaskingTID1(const real_T *rtu_In2,
  DW_mSubMultiRateMultiTaskin_f_T *localDW)
{
  int_T tid = 1;
  real_T rtb_Gain;
  rtb_Gain = 5.0 * *rtu_In2;
  localDW->RateTransition_Buffer0 = rtb_Gain;
  (void) (tid);
}
```

In R2015b, the generated code contains two function calls. For each sample time, there is
one function producing output and updating tasks. If `mTopMultiRateMultiTasking` is
an atomic subsystem instead of a referenced model, a similar enhancement to the
generated code from R2015a to R2015b occurs.

If you have a Simulink Code Inspector™ license, this optimization enables code inspection for a subset of multirate models. For more information on how Simulink Code Inspector supports multirate models, see Code inspection for multiple rate modeling including top models and Rate Transition blocks.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2015a

Version: 8.8

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Command-line APIs for protected models

Previously, protected models supported only a single system target. In R2015a, using new command-line APIs, you can create a protected model that supports code generation for multiple system targets.

If you create a protected model that supports multiple targets, create the protected model with the `Modifiable` option using the `Simulink.ModelReference.protect` function. Once you have this modifiable protected model, you can manage the targets it supports using the following new functions:

- `Simulink.ProtectedModel.addTarget`
- `Simulink.ProtectedModel.removeTarget`
- `Simulink.ProtectedModel.getSupportedTargets`
- `Simulink.ProtectedModel.getCurrentTarget`
- `Simulink.ProtectedModel.setCurrentTarget`
- `Simulink.ProtectedModel.getConfigSet`

For more information on creating a multi-target protected model, see Create a Protected Model with Multiple Targets.

If you are using a protected model that supports multiple targets, the new APIs allow you to:

- Get a list of supported targets using `Simulink.ProtectedModel.getSupportedTargets`. This information is also available in the protected model report.
- Get the configuration set for your target using `Simulink.ProtectedModel.getConfigSet`. With this information, you can verify that your interface is compatible with the protected model.

When generating code for your protected model, the build process selects the appropriate target.

For more information on using a multi-target protected model, see Use a Protected Model with Multiple Targets.

Improved use of workers for faster parallel builds

In R2015a, parallel builds of model reference hierarchies use an enhanced scheduling algorithm that potentially improves worker allocation and processor core use. For models containing large model reference hierarchies, the new algorithm might speed up Simulink diagram updates and Simulink Coder builds. For more information, see Reduce Update Time for Referenced Models and Reduce Build Time for Referenced Models.

Usability enhancements for protected models

Open support for protected models

Previously, to inspect a protected model, you referenced it in a model block, and then right-clicked in the model block and selected `Display report` or `Display Web view` from the context menu.

In R2015a, it is easier to inspect these models. You can access the Web view or the report for your protected model by using one of the following methods:

* Use the `Simulink.ProtectedModel.open` function. Calling this function with only the protected model name opens the model according to the following rules. Or you can choose how to view the model by specifying `'webview'` or `'report'` as the second argument. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  ```
  Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');
  ```
* Double-click the `.slxp` protected model file in the Current Folder browser.
* In the Block Parameter dialog box for the protected model, click **Open Model**.

Unless you have selected a specific option using the `Simulink.ProtectedModel.open` function, each of these methods first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

Protected model support for Rapid Accelerator mode

In R2015a, top models that reference protected models can be simulated in Rapid Accelerator mode.

Platform independence for protected model Web view

In R2015a, the Web view for a protected model is independent of the platform. You can view it on platforms other than the platform for which you created the protected model.

No code reuse for function-call subsystems with mask parameters

When you use multiple identical instances of a function-call subsystem in a model, the code generator does not create a reusable function if both of these conditions are true:

1   You clear the model configuration parameter **Inline parameters**.

2   You use a mask parameter of any kind in any of the subsystem instances.

Under these conditions, the code generator creates a unique function to represent each instance of the subsystem instead of a single reusable function.

Compatibility Considerations

Previously, if you generated code using a model that satisfied the preceding conditions, the code generator created a single reusable function to represent the subsystems.

In R2015a, the code generator creates a unique function for each subsystem instance, increasing code size and reducing readability. However, the code produces the same numerical results.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2014b

Version: 8.7

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Code generation for Simulink Function and Function Caller blocks

Simulink Coder supports code generation for the Simulink Function and Function Caller blocks. These blocks allow you to:

- Define a function implementation, which can be invoked by a function caller or a Stateflow chart
- Call a function implementation to compute outputs

For more information, see the Simulink Function and Function Caller block reference pages.

You can use the blocks to model client-server communication. For example, using Embedded Coder, you can model AUTOSAR clients and servers for simulation and code generation. For more information, see Client-Server Interface in the Embedded Coder AUTOSAR documentation.

Option to suppress generation of shared constants

You can choose whether or not the code generator produces shared constants and shared functions. You can change this parameter programmatically using the parameter `GenerateSharedConstants` with `set_param` and `get_param`.

For more information, see Shared Constant Parameters for Code Reuse.

Usability enhancements for protected models

In R2014b, the following features enhance the usability of protected models:

- Previously, you created a protected model only from a command-line API or from the context menu of a model reference block. In R2014b, from the Simulink Editor menu bar, you can select **File > Export Model To > Protected Model** to create a protected model from the current model. For more information on protecting a model, see Protect a Referenced Model.
- Previously, to update the configuration options for a protected model, you deleted the current protected model and recreated it with the new options. In R2014b, using the `Modifiable` option for `Simulink.ModelReference.protect`, you can create a

protected model that is modifiable by an authorized user. An authorized user can then use the `Simulink.ModelReference.modifyProtectedModel` and the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` functions to update the options for the protected model and provide a password for authorization.

- The protected model report provides more information. For each possible functionality that the protected model can support, the **Supported functionality** section reports `On`, `Off`, or `On with password protection`. The new **Licenses required** section lists all licenses required to run the protected model. For more information on the protected model report, see Protected Model Report.

- The read-only view functionality for protected models now uses the model Web view features introduced in R2014a. For more information on the Web view, see the Simulink Report Generator documentation.

Data, Function, and File Definition

Enumerated data type size control

You can reduce ROM/RAM usage, improve the portability of generated code, and improve integration with legacy code by specifying the size for enumerated data types. The code generator uses the super class that you define for the enumeration to specify the data type size in the generated code.

For example, the code generator uses this class definition:

```
classdef Colors
int8
   enumeration
     Red(0)
     Green(1)
     Blue(2)
   end
end
```

to generate this code:

```
typedef int8_T Colors;

#define Red     ((Colors)0)
#define Green ((Colors)1)
#define Blue    ((Colors)2)
```

For more information, see Enumerations.

Vector and matrix expressions as model argument values

You can now provide a vector or matrix expression as a model argument for a Model block. You can generate code for a model that contains a referenced model where the **Model argument values (for this instance)** parameter takes a vector or matrix expression.

Code Generation

Option to separate output and update functions for GRT targets

The software supports the **Single output/update function** (`CombineOutputUpdateFcns`) configuration parameter for Generic Real-Time (`grt.tlc`) targets.

If you clear the **Configuration Parameters** > **Code Generation** > **Interface** > **Single output/update function** check box, the software generates output and update function code for your model blocks as separate *model_output* and *model_update* functions. If you select the check box, the software generates the output and update function code as a single *model_step* function. For more information, see Single output/update function.

Previously, for Generic Real-Time targets, the software generated code as a single function. Support for the **Single output/update function** parameter was available only for Embedded Coder (`ert.tlc`) targets.

Highlighted configuration parameters from Code Generation Advisor reports

When you click a link to a configuration parameter from a Code Generation Advisor report, the parameter is highlighted in the Configuration Parameters dialog box.

License requirement for viewing code generation report

In R2014b, a Simulink Coder license is required to view a code generation report.

Compatibility Considerations

Previously, you did not need a license to view the code generation report.

Improved report generation performance

When you use `codegen.rpt` to create code generation reports with Simulink Report Generator™, in the Report Options dialog box on the **Properties** pane, the **Compile model to report on compiled information** check box is selected by default. With this

option, the software updates a model only once when creating the report. You get much faster report generation, especially for models with many atomic subsystems. For more information, see Document Generated Code with Simulink Report Generator.

Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation

In R2014b, you can select an Intel® Performance Primitive (IPP) code replacement library for a specific platform. You can generate code for a platform that is different from the host platform that you use for code generation. The new code replacement libraries are:

- Intel IPP for x86-64 (Windows)
- Intel IPP/SSE with GNU99 extensions for x86-64 (Windows)
- Intel IPP for x86/Pentium (Windows)
- Intel IPP/SSE with GNU99 extensions for x86/Pentium (Windows)
- Intel IPP for x86-64 (Linux)
- Intel IPP/SSE with GNU99 extensions for x86-64 (Linux)

For a model that you create in R2014b, you can no longer select these libraries:

- Intel IPP
- Intel IPP/SSE with GNU99 extensions

If, however, you open a model from a previous release that specifies Intel IPP or Intel IPP/SSE with GNU99 extensions, the library selection is preserved and that library appears in the selection list.

See Choose a Code Replacement Library.

Deployment

Support for Eclipse IDE and Desktop Targets has been removed

Simulink Coder support for Eclipse™ IDE has been removed.

You can no longer use Simulink Coder with Eclipse IDE to build and run generated code on your host desktop computer that has Linux or Windows.

Compatibility Considerations

There are no recommended alternatives for using Simulink Coder with Eclipse IDE and Desktop Targets.

Performance

Block reduction optimization improvement

The block reduction optimization includes reduction of code for blocks that generate dead code. In the Configuration Parameters dialog box, on the **Optimization** pane, select the Block reduction check box to enable this optimization. In R2014b, the code generator searches for source blocks connected to a block's unused input port.

To use this optimization for an S-function block, designate an input port as `NEVER_NEEDED` using `ssSetInputPortSignalWhenNeeded(S,0,NEVER_NEEDED)`. `S` is a `SimStruct` representing an S-Function block. You can call this function in the `mdlInitializeSizes` function or the `mdlSetWorkWidths` function.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2014a

Version: 8.6

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Custom post-processing function for protected models

You can specify a post-processing function for code generated from a protected model using the `'CustomPostProcessingHook'` option of the `Simulink.ModelReference.protect` function. You can use this option to run a third-party custom obfuscator on the generated code.

For more information, see Specify Custom Obfuscator for Protected Model.

Context-sensitive help for the Create Protected Model dialog box

In R2014a, context-sensitive help (CSH) is available for parameters in the Create Protected Model dialog box.

Data, Function, and File Definition

C++ class generation

Beginning in R2014a, you can generate encapsulated C++ class code from GRT-based models, in addition to the previously supported ERT-based models. The following new parameters on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box support C++ class code generation:

- **Code interface packaging**
- **Multi-instance code error diagnostic**

The general procedure for generating C++ class interfaces to model code is as follows:

1   Select the `C++` language for your model.

2   Select `C++ class` code interface packaging for your model.

3   Generate model code.

4   Examine the C++ class interfaces in the generated files and the HTML code generation report.

For more information, see Generate C++ Class Interface to Model or Subsystem Code.

Simpler behavior for tuning all parameters and support for referenced models

This release simplifies the way Simulink considers the `InlineParameters` option when it is set to `Off`. You can perform the following operations:

- Tune all block parameters in your model during simulation, either through the parameters themselves or through the tunable variables that they reference.
- Preserve the mapping between a block parameter and a variable in generated code even when the block parameter does not reference any tunable variables.
- Retain the mapping between tunable workspace variables and variables in generated code, irrespective of the `InlineParameters` setting.
- Set the value of `InlineParameters` to `Off` for model references.

These behaviors are consistent across models containing reusable subsystems and reference models.

Compatibility Considerations

The simplified behavior enhances the generated code and provides improved mapping between a block parameter and a variable in generated code.

Block parameter expression	Code generated previously	Code generated in R2014a
Expressions referencing global variables (e.g., `K+1`)	Variable name is not preserved. Block parameter name is preserved. ```struct Parameters_model_ {
real_T Gain_Gain; // Expression: K+1
}
y = model_P.Gain_Gain*u;```	Expression is considered tunable. Variable name is preserved in code and is tunable. ```real_T K = 2.0;
y = (K+1)*u;```
Expressions referencing mask parameters for nonreusable subsystems (e.g., `MP*3`), the value of `MP` being a nontunable expression.	Variable name is not preserved. Block parameter name is preserved. ```struct Parameters_model_ {
real_T Gain_Gain; // Expression: MP*3
}
y = model_P.Gain_Gain*u;```	Expression is considered tunable. Variable name is substituted by parameter value. ```struct Parameters_model_ {
real_T Subsystem_MP;
}
y = (model_P.Subsystem_MP * 3) * u;```
Expressions referencing model arguments (resp. mask parameters) for referenced models (resp. reusable subsystems) (e.g., `Arg+1`)	Variable name is not preserved. Block parameter name is preserved. ```struct Subsystem {
Gain_Gain; // Expression: Arg+1
}
y = model_P.Subsystem1.Gain_Gain*u;```	Variable name is preserved as an argument name. ```subsystem(y, u, rtp_Arg) {
y = (rtp_Arg+1)*u;
}```

To revert the behavior of InlineParameters Off to what it was in R2013b, run `revertInlineParametersOffToR2013b` at the MATLAB Command Line. Alternately, add `revertInlineParametersOffToR2013b` to your MATLAB startup function.

After running `revertInlineParametersOffToR2013b`, you cannot undo the change in the same MATLAB session. To return to the InlineParameters Off behavior in R2014a, restart MATLAB.

The command `inlineParametersOffRevertedToR2013b` returns a logical true or false to indicate whether the InlineParameters Off behavior has been reverted to that in R2013b.

Code generated using the R2013b `InlineParameters` Off behavior is not compatible with the code generated using the R2014a `InlineParameters` Off behavior. Therefore, run `revertInlineParametersOffToR2013b` before code generation.

Improved control of C and C++ code interface packaging

R2014a provides improved control of code interface packaging for generated model code, including nonreusable code, reusable code, and encapsulated C++ class code. To support more robust control of code interface packaging, the following changes were made to the **Code Generation > Interface** pane of the Configuration Parameters dialog box and corresponding command-line model parameters:

- The new model parameter **Code interface packaging** (`CodeInterfacePackaging`) selects the packaging for the C or C++ code interface generated for your model. The possible values are:

  - `Nonreusable function`
  - `Reusable function`
  - `C++ class` (available if **Language** is set to `C++`)

  Note As described in "C++ class generation" on page 9-3, C++ class code generation is now available to GRT-based models as well as ERT-based models.

- The model option **Generate reusable code** (`MultiInstanceERTCode`) has been removed. Setting the value of **Code interface packaging** to `Reusable function` or `Nonreusable function` is equivalent to selecting or clearing **Generate reusable code** in releases before R2014a.

- The model parameter **Reusable code error diagnostic** has been renamed to **Multi-instance code error diagnostic**. The parameter now supports GRT models and C++ class code generation. For more information, see "Multi-instance code error diagnostic for reusable function code and C++ class code" on page 9-7.

- Parameters within the **Code interface** subpane have been regrouped and relocated to improve pane navigation and code interface configuration workflows. The following figure shows the **Code interface** subpane when **Code interface packaging** is set to `C++ class` for an ERT model.

Compatibility Considerations

- For GRT and ERT-based models, selecting the **Language** (`TargetLang`) value `C++` now provides two possible forms of C++ code interface packaging:

  - If you set **Code interface packaging** (`CodeInterfacePackaging`) to `C++ class`, the build generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.

  - If you set **Code interface packaging** to a value other than `C++ class`, the build generates C++ compatible `.cpp` files containing model interfaces enclosed within an `extern "C"` link directive. This behavior is equivalent to code generation with **Language** set to `C++` before R2014a.

- A script created before R2014a might select C++ `extern "C"` code packaging for a model by using `set_param` to set `TargetLang` to `C++`. Beginning in R2014a, setting `TargetLang` to `C++` selects only the C++ language, and does not alter the C++ code

interface packaging. To produce the same result as before R2014a, update the script to set both `TargetLang` and `CodeInterfacePackaging`.

- For ERT-based models, the **Language** value `C++ (Encapsulated)` is no longer available. To configure encapsulated C++ class code generation, set **Language** to `C++` and set **Code interface packaging** to `C++ class`. This pair of settings is equivalent to setting **Language** to `C++ (Encapsulated)` in releases before R2014a.

- For GRT and ERT-based models, the model option **Generate reusable code** (`MultiInstanceERTCode`) has been removed. Setting the value of **Code interface packaging** to `Reusable function` or `Nonreusable function` is equivalent to selecting or clearing **Generate reusable code** in releases before R2014a.

Multi-instance code error diagnostic for reusable function code and C++ class code

The model parameter **Reusable code error diagnostic** has been renamed to **Multi-instance code error diagnostic**. Before R2014a, **Reusable code error diagnostic** applied only to ERT-based models and to reusable function code. Beginning in R2014a, **Multi-instance code error diagnostic** also applies to GRT-based models and to C++ class code.

The **Multi-instance code error diagnostic** parameter specifies the diagnostic action that the build process takes when a model violates strict requirements for generating multi-instance code:

- `None` — Proceed with build without displaying a diagnostic message.
- `Warning` — Proceed with build after displaying a warning message.
- `Error` (default) — Abort build after displaying an error message.

The name of the equivalent command-line parameter, `MultiInstanceErrorCode`, is unchanged.

Compatibility Considerations

- In releases before R2014a, **Reusable code error diagnostic** did not apply to GRT models. Now, renamed to **Multi-instance code error diagnostic**, the parameter applies to GRT models. Its default value is `Error`.

    If you load a GRT model created before R2014a, for which reusable code generation is selected, by default, code generation now applies strict multi-instance code

requirements to the model during code generation, and the build might fail. If the build fails, examine the condition that caused the error message. Decide whether to reset **Multi-instance code error diagnostic** to `Warning` or `None`, or leave **Multi-instance code error diagnostic** set to `Error` and modify the model to remove the condition.

- Before R2014a, the setting of the ERT model parameter **Reusable code error diagnostic** was ignored if a model was configured to do all of the following:

  - Generate reusable code.
  - Generate a function to allocate model data for each model instance.
  - Simulate in External mode.

  Beginning in R2014a, if a model is configured as described, the setting of the GRT and ERT model parameter **Multi-instance code error diagnostic** is honored. If the diagnostic parameter is set to `Error` (the default value), a model that built successfully before R2014a might fail to build. If the build fails, examine the condition that caused the error message. Decide whether to reset **Multi-instance code error diagnostic** to `Warning` or `None`, or leave **Multi-instance code error diagnostic** set to `Error` and modify the model to remove the condition.

Removal of TRUE and FALSE from rtwtypes.h

When the target language is C, `rtwtypes.h` compiles the definitions for `true` and `false` into the code. It no longer defines `TRUE` and `FALSE`.

If you integrate code generated in R2014a with custom code that references `TRUE` and `FALSE`, modify your custom code in one of these ways:

- Define `TRUE` and `FALSE`.
- Change `TRUE` to `true` and `FALSE` to `false`.
- Change `TRUE` to `1U` and `FALSE` to `0U`.

Code Generation

Independent configuration selections for standard math and code replacement libraries

In R2014a, you can independently select and configure standard math and code replacement libraries for code generation with the following changes in the Configuration Parameters dialog box.

- On the top-level **Code Generation** pane, the **Language** (`TargetLang`) parameter setting determines options that are available for a new **Standard math library** parameter on the **Code Generation** > **Interface** pane.

- Depending on your **Language** selection, the new **Standard math library** (`TargetLangStandard`) parameter on the **Code Generation** > **Interface** pane lists these options.

Language	Standard Math Libraries
C	C89/C90 (ANSI) – default C99 (ISO)
C++	C89/C90 (ANSI) – default C99 (ISO) C++03 (ISO)

- On the **Code Generation** > **Interface** pane, the **Code replacement library** (`CodeReplacementLibrary`) parameter lists available code replacement libraries. The Simulink Coder software filters the list based on compatibility with the **Language** and **Standard math library** settings and on product licensing (for example, Embedded Coder offers more libraries and the ability to create and use custom code replacement libraries).

For more information, see:

- Language
- Standard math library
- Code replacement library

Compatibility Considerations

In R2014a, code replacement libraries provided by MathWorks® no longer include standard math libraries.

- When you load a model created with an earlier version:

  - The `CodeReplacementLibrary` parameter setting remains the same unless previously set to `C89/C90 (ANSI)`, `C99 (ISO)`, `C++ (ISO)`, `Intel IPP (ANSI)`, or `Intel IPP (ISO)`. In these cases, Simulink Coder software sets `CodeReplacementLibrary` to `None` or `Intel IPP`.

  - Simulink Coder software sets the new `TargetLangStandard` parameter to `C89/C90 (ANSI)`, `C99 (ISO)`, or `C++03 (ISO)`, depending on the previous `CodeReplacementLibrary` setting.

If `CodeReplacementLibrary` was set to	`TargetLangStandard` is set to
`C89/C90 (ANSI)`, `C99 (ISO)`, or `C++ (ISO)`	`C89/C90 (ANSI)`, `C99 (ISO)`, or `C++03 (ISO)`, respectively
`GNU99 (GNU)`, `Intel IPP (ISO)`, `Intel IPP (GNU)`, `ADI TigerSHARC` (Embedded Coder only), or `MULTI BF53x` (Embedded Coder only)	`C99 (ISO)`
A custom library (Embedded Coder only), and the corresponding registration file has been loaded in memory	A value based on the `BaseTfl` property setting
Any other value	The default standard math library, `C89/C90 (ANSI)`

- When you select a code replacement library provided by MathWorks after you load a model, the code generator can produce different code than in previous versions depending on the `TargetLangStandard` setting. Verify generated code.

- When you export a model created in R2014a, the Simulink Coder software:

  - Uses the `TargetLangStandard` setting to map to the closest available code replacement library or the default library in the previous version, if `CodeReplacementLibrary` is set to `None` or `Intel IPP`.

  - Otherwise, ignores the `TargetLangStandard` parameter.

Generated code compilation using LCC-64 bit on Windows hosts

You can now use the `LCC-win64` compiler, included on Windows 64-bit platforms, for GRT model builds. If you have an Embedded Coder license, you also can use the compiler for ERT model builds and SIL and PIL mode simulation on Windows hosts.

Improved code integration of shared utility files

Previously, the code generator created the file `rtw_shared_utils.h` which included header files associated with CRL replacements and Simulink and MATLAB utilities.

In R2014a, code generation no longer creates `rtw_shared_utils.h`. Code generation for a model produces code which directly includes only those header files required for the code. For subsystems, code generation includes only those header files required for the subsystem code. The generated code is more deterministic. You can easily integrate code generated from separate software components.

Optimized inline constant expansion

In R2014a, the code generator expands inline references to buses, bus arrays, and complex arrays where the elements are all constant and equal. This enhancement improves execution speed and reduces RAM consumption.

rtwtypes.h included before tmwtypes.h

If code generation produces a program file which includes the header files `rtwtypes.h` and `tmwtypes.h`, then code generation includes `rtwtypes.h` first. This file order occurs whether the program file includes the header files directly or indirectly.

When code generation creates `rtwtypes.h`, it includes typedef definitions tailored for the target, using model parameter settings. `rtwtypes.h` is included first so that its target-specific typedef definitions are the definitions used when compiling the code.

Constant block output value used when in nonreusable subsystem

Previously, when you defined a Constant block within a nonreusable subsystem, and then connected a block to the Constant block outside that subsystem, the connected block used the value from the `Value` parameter directly. Now, the connected block uses the output of the Constant block.

Using the Constant block output simplifies mapping from the Subsystem block to its representation in the generated C code.

Deployment

Support for Eclipse IDE and Desktop Targets will be removed

Simulink Coder support for Eclipse IDE will be removed in a future release.

Currently, you can use Simulink Coder support for Eclipse IDE to:

- Build and run generated code on your host desktop computer running Linux or Windows.
- Generate multitasking code that uses POSIX threads (Pthreads) for concurrent execution.
- Tune parameters on, and monitor data from, an executable running on the target hardware (External mode).
- Perform numeric verification using processor-in-the-loop (PIL) simulation.
- Generate IDE projects and use the Automation Interface API.
- Generate makefile projects using the mingw_host configuration in XMakefile.
- Use Linux Task and Windows Task blocks

Compatibility Considerations

There are no recommended alternatives for using Simulink Coder with Eclipse IDE and Desktop Targets.

Additional build folder information and protected model support for RTW.getBuildDir function

In 2014a, when you use the `RTW.getBuildDir` function to get build folder information, these new fields are available:

- `ModelRefRelativeRootSimDir` — String specifying the relative root folder for the model reference target simulation folder.
- `ModelRefRelativeRootTgtDir` — String specifying the relative root folder for the model reference target build folder.
- `SharedUtilsSimDir` — String specifying the shared utility folder for simulation.

- `SharedUtilsTgtDir` – String specifying the shared utility folder for code generation.

In addition, the `RTW.getBuildDir` function can return build folder information for protected models.

Wind River Tornado (VxWorks 5.x) target to be removed in future release

The Wind River® Tornado® (VxWorks® 5.x) target will be removed from Simulink Coder software in a future release. If you generate code using the system target file `tornado.tlc`, the software displays a warning about future removal of the target.

Beginning in R2014a, you can no longer select the system target file `tornado.tlc` for a model using the list of targets in the System Target File Browser. However, you can still specify the Tornado target. Either enter the text `tornado.tlc` in the **System target file** parameter field or, from the MATLAB command line, use the `set_param` command to set the `SystemTargetFile` parameter to `'tornado.tlc'`.

Compatibility Considerations

If you have an Embedded Coder license, you can use the Wind River VxWorks support package. The support package allows you to use the XMakefiles feature to automatically generate and integrate code with VxWorks 6.7, VxWorks 6.8, and VxWorks 6.9. For more information, see `www.mathworks.com/hardware-support/vxworks.html`.

Performance

To Workspace, Display, and Scope blocks removed by block reduction

When performing block reduction, the code generator can eliminate the Display block when all of the following are true:

- External mode is off. On the **Code Generation** > **Interface** pane, set **Interface** to `None`.
- The target is not a simulation target, such as `grt.tlc`, `ert.tlc`, and `autosar.tlc`.

When doing block reduction, the code generator can eliminate the To Workspace and Scope blocks when all of the following are true:

- External mode is off. On the **Code Generation** > **Interface** pane, set **Interface** to `None`.
- The system target file is `grt.tlc` or `ert.tlc`.
- MAT-file logging is off. On the **Code Generation** > **Interface** pane, the **MAT-file logging** check box is cleared.

Optimized reusable subsystem inputs

When processing the inputs to reusable subsystems, code generation optimizes code reuse and efficiency when you select the input signal. To select the input signal, use a virtual Bus Selector block or a Selector block. This enhancement also improves traceability.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2013b

Version: 8.5

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Multilevel access control when creating password-protected models for IP protection

If you create a protected model, you have the option to specify different passwords to control protected model functionality. The supported types of functionality for password protection are:

- Model viewing
- Simulation
- Code generation

For password-protected models, before using each type of supported functionality, you must enter a password.

To protect your models with passwords, right-click a model reference block, and then select **Subsystem & Model Reference** > **Create Protected Model for Selected Model Block**. Select the functionality that you want supported inside your protected model. Enter a password for each type of supported functionality. Passwords must be a minimum of four characters. When you are finished creating passwords, click **Create**.

To use the supported functionality of a protected model, you must enter the password. Right-click the protected model shield icon and select **Authorize**. Enter the password, and click **OK**.

Simulink Coder checks in Model Advisor

You can use the Model Advisor Simulink Coder Checks to verify that your model is configured for code generation. Previously, the checks were available in the Model Advisor Embedded Coder folder.

Data, Function, and File Definition

Imported data can be shared

You can now import header files and use them in the shared utilities folder. Previously, the only imported data types available were those data types exported by a previous model.

Some shared items using imported data types include:

- Reusable library subsystems
- Constants shared across models
- Shared data and `Simulink` or `mpt` custom storage classes such as `ExportToFile` and `Volatile`

For more information, see Incremental Shared Utility Code Generation and Compilation.

Compatibility Considerations

- Previously, you changed imported data types without affecting the shared utilities folder. Now, if you change an imported data type, code generation notifies you that you must clear `slprj` and regenerate code.
- Previously, you created an imported data type with no header file specified. Now, if there is no header file for an imported data type, code generation generates an error.

Readability improved for constant names

To make constant parameter names more easily readable, code generation uses a macro (#define) to create the name when:

- The constant parameter is defined in the shared utilities folder.
- The constant parameter is not in a shared function.

When the constant parameter is used in a shared function, code generation always generates a checksum-based name.

Previously, if a constant parameter definition was generated to the shared location, code generation sometimes used the checksum-based name in nonshared function files.

Removal of two's complement guard and RTWTYPES_ID from rtwtypes.h

In 2013b, the following changes have been made to `rtwtypes.h`:

- This code has been removed from `rtwtypes.h`:

```
/*
 * Simulink Coder assumes the code is compiled on a target using a 2's complement
 * representation for signed integer values.
 */
#if ((SCHAR_MIN + 1) != -SCHAR_MAX)
#error "This code must be compiled using a 2's complement representation for signed integer values"
#endif
```

  Simulink Coder still assumes code is compiled on a target using a two's complement representation for signed integer values.

- The definition of the macro `RTWTYPES_ID` has been removed from `rtwtypes.h`. The definition is no longer referenced from `model_private.h`.

For information about the `rtwtypes.h` file, see Files and Folders Created by Build Process.

MODEL_M macro renamed in static main for multi-instance GRT target

In R2013a, the static main program module *matlabroot*`/rtw/c/src/common/rt_malloc_main.c` defined a *MODEL_M* macro for getting the `rtModel` type for the model. R2013b renames the macro to *MODEL_M_TYPE* to resolve a potential naming conflict.

Compatibility Considerations

If you used R2013a materials to develop a custom target based on GRT with model option **Generate reusable code** selected, update your custom static main to use *MODEL_M_TYPE* instead of *MODEL_M* to get the `rtModel` type. You can use the installed static main module *matlabroot*`/rtw/c/src/common/rt_malloc_main.c` as a reference point.

Code Generation

Optimized code for long long data type

If your target hardware and your compiler support the C99 long long integer data type, you can select to use this data type for code generation. Using long long results in more efficient generated code that contains fewer cumbersome operations and multiword helper functions. This data type also provides more accurate simulation results for fixed-point and integer simulations. If you are using Microsoft® Windows (64-bit), using long long improves performance for many workflows including using Accelerator mode and working with Stateflow software.

For more information, see the **Enable long long** and **Number of bits: long long** configuration parameters on the Hardware Implementation Pane.

At the command line, you can use the following new model parameters:

- `ProdLongLongMode`: Specify that your C compiler supports the long long data type. You must set this parameter to `'on'` to enable `ProdBitPerLongLong`.
- `ProdBitPerLongLong`: Describes the length in bits of the C long long data type supported by the production hardware.
- `TargetLongLongMode`: Specifies whether your C compiler supports the long long data type. You must set this parameter to `'on'` to enable `TargetBitPerLongLong`.
- `TargetBitPerLongLong`: Describes the length in bits of the C long long data type supported by the hardware used to test generated code.

For more information, see Model Parameters.

<LEGAL> tokens removed from comments in generated code

Copyright notice comments in the generated code no longer include a `<LEGAL>` token. Copyright notices are now bound by `COPYRIGHT NOTICE` at the top and `END` at the bottom.

Deployment

Compiler toolchain interface for automating code generation builds

You can configure your model to generate code using **Toolchain settings** instead of **Template makefile** parameters. The software and documentation refer to using these settings as the toolchain approach.

The toolchain approach enables you to:

- Select the toolchain a Simulink model uses to build generated code.
- Use custom toolchains that are registered using MATLAB Coder software.
- Select a build configuration such as `Faster Builds`, `Faster Runs`, `Debug`.
- Customize a build configuration, such as setting compiler optimization flags, using `Specify`.
- Use support packages that include custom toolchains.

To use the toolchain approach:

1 Open the Configuration Parameters dialog box for Simulink model by pressing **Ctrl +E**.
2 In Configuration Parameters, select the **Code Generation** pane.
3 Click the **Browse** button for the **System target file** parameter, and select one of the following:

- `grt.tlc — Generic Real-Time Target` (default)
- `ert.tlc — Embedded Coder` (Requires the Embedded Coder product)
- `ert_shrlib.tlc — Embedded Coder (host-based shared library target)` (Requires the Embedded Coder product)

When you use toolchain approach, the following **Toolchain settings** are available:

- **Target hardware** (only with `ert.tlc — Embedded Coder`)
- **Toolchain**
- **Build Configuration**

When you use the toolchain approach, the following **Makefile configuration** are unavailable:

- **Generate makefile**
- **Make command**
- **Template makefile**

For more information, see:

- Configure the Build Process
- Custom Toolchain Registration
- Code Generation Pane: General

Compatibility Considerations

When you open a Simulink model that has **System target file** set to `grt.tlc`, `ert.tlc`, or `ert_shrlib.tlc`, Simulink Coder software automatically updates the model to use the toolchain approach. If the model does not use a default template makefile or configuration settings, Simulink Coder software might not upgrade the model. For more information, see Upgrade Model to Use Toolchain Approach.

Log data on Linux-based target hardware

With Linux-based target hardware for the Simulink "Run on Target Hardware" feature, you can log data from a model to a MAT-file, and then pull that data into MATLAB for analysis.

This capability:

- Works with Raspberry Pi™, BeagleBoard, Gumstix® Overo®, and PandaBoard hardware.
- Enables you to log real-time data from signals attached to scopes, root-level I/O ports, or To Workspace blocks.
- Saves the logged data to MAT-files on the target hardware.
- Enables you to use SSH to transfer MAT-files to your host computer.

For more information, see Log Data on Linux-based Target Hardware and Target Hardware

---

Note This feature requires a Simulink Coder license.

Modified file locations and commands for rebuilding external mode MEX files

In R2013b, files used in MATLAB commands to rebuild the standard external mode MEX files `ext_comm` and `ext_serial_win32` moved to new locations in the installed MATLAB tree. For example:

- Files located in `matlabroot`/toolbox/rtw/rtw moved to `matlabroot`/toolbox/coder/simulinkcoder_core.

- Files located in `matlabroot`/rtw/ext_mode moved to `matlabroot`/toolbox/coder/simulinkcoder_core/ext_mode/host.

Compatibility Considerations

Commands for rebuilding the standard `ext_comm` and `ext_serial_win32` modules on Windows and UNIX® platforms must be updated to reference the new file locations. See the table MATLAB Commands to Rebuild ext_comm and ext_serial_win32 MEX-Files.

Performance

Reduced data copies for bus signals with global storage

Data copies are reduced when subsystem outputs are global and packed into a bus through a bus creator block. This enhancement improves execution speed and reduces RAM consumption.

For this optimization your model requires all of the following conditions:

- Set subsystem **Function packaging** parameter to `Inline` or `Nonreusable`.
- The signal property for output signal cannot be `Testpoint`.
- The subsystem must have a single destination.
- For a conditionally executed subsystem's properties, set the output, when disabled, to `held`.

Previously, code generation might produce extra data copies for bus signals with global storage.

Customization

Support for user-authored MATLAB system objects

Simulink Coder supports code generation for the MATLAB System block, which allows you to include a System object in your Simulink model. This capability is useful for including algorithms. For more information, see System Object Integration.

TLC Options removed from Configuration Parameters dialog box

The model parameter **TLC options** has been removed from the **Code Generation** pane of the Configuration Parameters dialog box. However, at the MATLAB command line, you can still use the `set_param` command to set the equivalent command-line parameter `TLCOptions`. For more information, see Specify TLC Options and Configure TLC.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2013a

Version: 8.4

New Features

Bug Fixes

Compatibility Considerations

Data, Function, and File Definition

Optimized interfaces for Simulink functions called in Stateflow

Previously, when subsystem input and output signals were used inside a Stateflow chart, the generated code for the input and output signals was copied into global variables. In R2013a, when the Subsystem block parameter Function packaging is set to `Inline`, the subsystem inputs and outputs called within a Stateflow chart are now local variables. This optimization improves execution speed and memory usage.

Shortened system-generated identifier names

For GRT targets, the length of the system-generated identifier names are shortened to allow for more space for the user-specified components of the generated identifier names. The name changes provide a more consistent and predictable naming system that uses camel case, no underscores or plurals, and consistent abbreviations for both a type and a variable.

The default for the system-generated identifiers per model are changed.

Before R2013a	In R2013a	Type
`BlockIO, B`	`B`	Type
`ExternalInputs`	`ExtU`	Type
`ExternalInputSizes`	`ExtUSize`	Type
`ExternalOutputs`	`ExtY`	Type
`ExternaOutputSizes`	`ExtYSize`	Type
`Parameters`	`P`	Type
`ConstBlockIO`	`ConstB`	`Const` Type
`MachineLocalData`	`MachLocal`	`Const` Type
`ConstParam, ConstP`	`ConstP`	`Const` Type, Global Variable
`ConstParamWithInit,` `ConstWithInitP`	`ConstInitP`	`Const` Type, Global Variable
`D_Work, DWork`	`DW`	Type, Global Variable

Before R2013a	In R2013a	Type
MassMatrixGlobal	MassMatrix	Type, Global Variable
PrevZCSigStates, PrevZCSigState	PrevZCX	Type, Global Variable
ContinuousStates, X	X	Type, Global Variable
StateDisabled, Xdis	XDis	Type, Global Variable
StateDerivatives, Xdot	XDot	Type, Global Variable
ZCSignalValues, ZCSignalValues	ZCV	Type, Global Variable
DefaultParameters	DefaultP	Global Variable
GlobalTID	GlobalTID	Global Variable
InvariantSignals	Invariant	Global Variable
Machine	MachLocal	Global Variable
NSTAGES	NSTAGES	Global Variable
Object	Obj	Global Variable
TimingBridge	TimingBrdg	Global Variable
U	U	Global Variable
USize	USize	Global Variable
Y	Y	Global Variable
YSize	YSize	Global Variable

The default for the system-generated identifiers names per referenced model or reusable subsystem are changed.

Before R2013a	In R2013a	Type
rtB, B	B	Type, Global Variable
rtC, C	ConstB	Type, Global Variable
rtDW, DW	DW	Type, Global Variable
rtMdlrefDWork , MdlrefDWork	MdlRefDW	Type, Global Variable
rtP, P	P	Type, Global Variable
rtRTM, RTM	RTM	Type, Global Variable

Before R2013a	In R2013a	Type
`rtX, X`	`X`	Type, Global Variable
`rtXdis, Xdis`	`XDis`	Type, Global Variable
`rtXdot, Xdot`	`XDot`	Type, Global Variable
`rtZCE, ZCE`	`ZCE`	Type, Global Variable
`rtZCSV, ZCSV`	`ZCV`	Type, Global Variable

For more information, see Construction of Generated Identifiers.

Code Generation

Shared utility name consistency across builds with maximum identifier length control

In R2013a, shared utility names remain consistent in the generated code across multiple builds of your model. In addition, shared utility names now comply with the **Maximum identifier length** parameter specified on the **Code Generation > Symbols** pane in the Configuration Parameters dialog box. The **Maximum identifier length** parameter does not apply to fixed-point and DSP utilities.

Code Generation Advisor available on menu bar

To launch the Code Generation Advisor, on the model menu bar, select **Code > C/C++ Code > Code Generation Advisor**. Alternatively, the Code Generation Advisor remains available in the Configuration Parameters dialog box, on the **Code Generation** pane.

For information about using the Code Generation Advisor to configure your model to meet specific code generation objectives, see:

- Application Objectives in Simulink Coder
- Application Objectives in Embedded Coder

Code generation build when reusable library subsystem link status changes

Shared functions for a reusable library subsystem are generated only for resolved library links. If you enable or disable a library link for a reusable subsystem, and then build your model, new code is generated.

Protected models usable in model reference hierarchies

Previously, you could not protect a model and use it in a model reference hierarchy.

In R2013a, you can use protected models in a model reference hierarchy. In addition, R2013a includes enhancements to the programmatic interface as well as the dialog for model protection.

To learn more about changes to the programmatic interface, see
`Simulink.ModelReference.protect` and to view the changes to the model protection
dialog, see Create a Protected Model.

Deployment

Simplified multi-instance code with support for referenced models

R2013a provides simplified multi-instance code deployment for GRT targets with support for referenced models.

In previous releases, to generate reentrant, reusable code with dynamic allocation of per-instance model data, you had to select a specialized target, `grt_malloc.tlc`, for the model. If you selected the GRT malloc target for a model, you could not include referenced models in your model design.

Beginning in R2013a, you can generate reentrant, reusable code for a GRT model by selecting the model configuration option **Generate reusable code**, which is located on the **Code Generation** > **Interface** pane of the Configuration Parameters dialog box.

Code interface

☐ Classic call interface   ☑ Generate reusable code
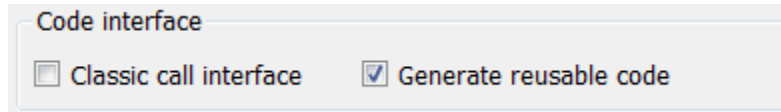
When you select **Generate reusable code** for a GRT model, the build process generates reusable, multi-instance code that is reentrant, as follows:

- The generated *model*.c source file contains an allocation function that dynamically allocates model data for each instance of the model.
- The generated code passes the real-time model data structure in, by reference, as an argument to *model*_step and the other model entry point functions.
- The real-time model data structure is exported with the *model*.h header file.

With the new GRT model option **Generate reusable code**, you can generate and deploy multi-instance code for your model without selecting a specialized target, and you can include referenced models in your model design.

Note Use of the `grt_malloc.tlc` target is no longer recommended. For more information, see "GRT malloc target to be removed in future release" on page 11-10.

External mode control panel improvements and C API access

Improved External mode graphical controls

External mode dialog boxes are now consistent with other Simulink dialog boxes, with improved layout, ability to resize, and consistent sets of buttons. The improved dialog boxes include the **External Mode Control Panel** and the subsidiary dialog boxes that you can open from it, **External Signal & Triggering** and **Enable Data Archiving**. Here is the improved **Enable Data Archiving** dialog box.
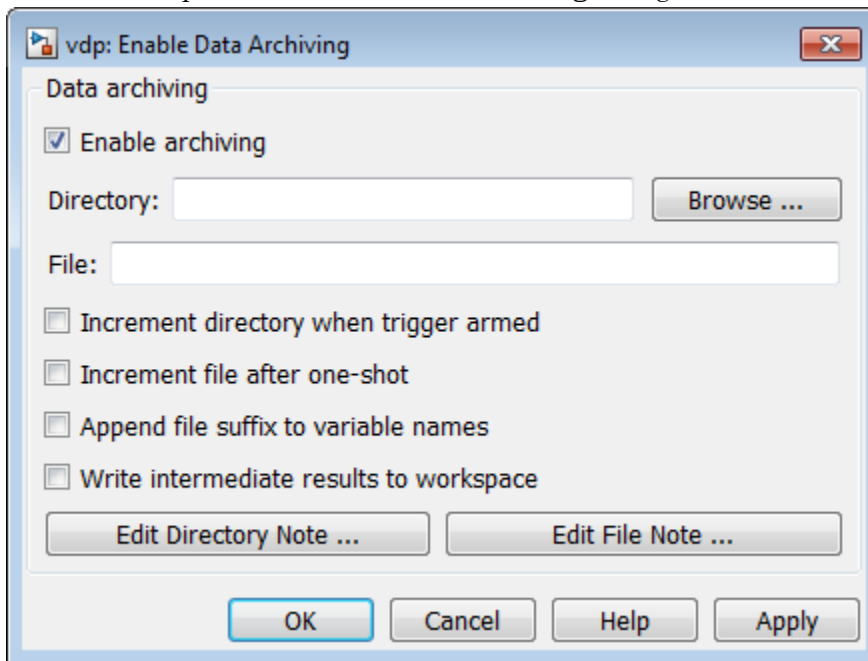
To view the improved External mode dialog boxes, open a model and select **Code** > **External Mode Control Panel**.

C API access from External mode simulations

In previous releases, the External mode and C API data interfaces for model code were mutually exclusive. Beginning in R2013a, you can generate code for your model with both the External mode and C API interfaces enabled. Custom code now can access C API data structures during an External mode simulation.

For more information, see Generate External Mode and C API Data Interfaces.

Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog

The contents of the Target Preferences block have been relocated to the new **Target Hardware Resources** tab on the Coder Target pane in the Configuration Parameters dialog box.
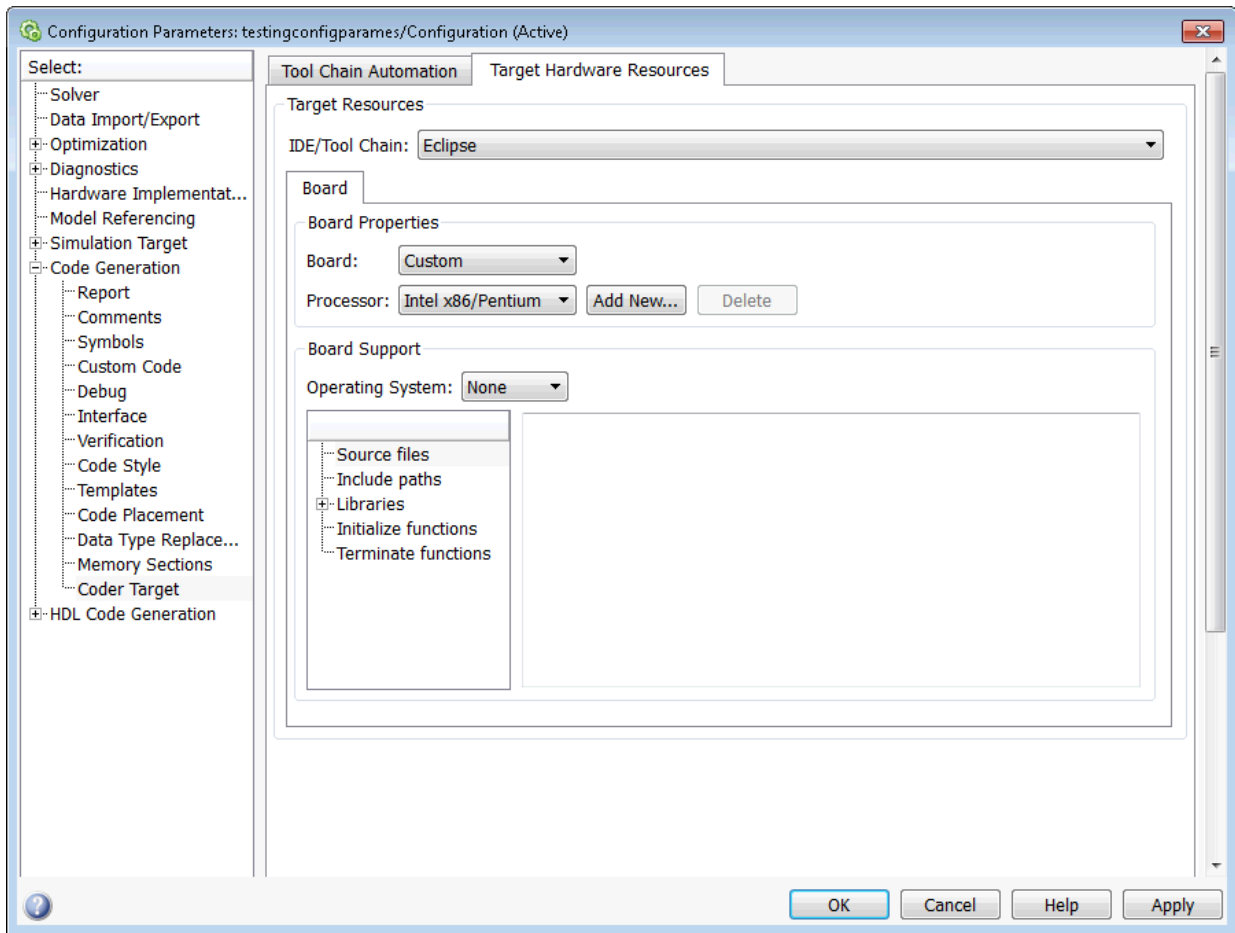
The Target Preferences block has been removed from the Desktop Targets block library.

If you open a model that contains a Target Preferences block, a warning instructs you that the block is optional and can be removed from your model.

Opening the Target Preferences block automatically displays the **Target Hardware Resources** tab.

For instructions on how to use **Target Hardware Resources** to build and run a model on desktop system, see Model Setup.

For information about specific parameters and settings, see Code Generation: Coder Target Pane.

Support ending for Eclipse IDE in a future release

Support for the Eclipse IDE will end in a future release of the Embedded Coder and Simulink Coder products.

GRT malloc target to be removed in future release

The GRT malloc target will be removed from Simulink Coder software in a future release.

Beginning in R2013a, you can no longer select the system target file `grt_malloc.tlc` for a model using the list of targets in the System Target File Browser. However, you can still specify the GRT malloc target. Either enter the text `grt_malloc.tlc` in the **System target file** parameter field or use the `set_param` command to set the `SystemTargetFile` parameter from the MATLAB command line.

Compatibility Considerations

If you are using the system target file `grt_malloc.tlc` to generate reentrant code with dynamic memory allocation, switch to using `grt.tlc` with the model configuration option **Generate reusable code**. As described in "Simplified multi-instance code with support for referenced models" on page 11-7, the **Generate reusable code** option offers several advantages over the GRT malloc target, including a simple multi-instance call interface and support for model reference hierarchies. For more information, see the help for **Generate reusable code**.

Customization

MakeRTWSettingsObject model parameter removed

In R2013a, the model parameter `MakeRTWSettingsObject` has been removed from the software. Before R2013a, custom target authors used `MakeRTWSettingsObject` in build hook functions to get the value of the current build folder path during the model build process.

Compatibility Considerations

If your *STF_make_rtw_hook* function uses the model parameter `MakeRTWSettingsObject` in a `get_param` function call, you must update the MATLAB code to use a different function call. For example, your hook function might contain code similar to the following.

```
makertwObj = get_param(gcs,'MakeRTWSettingsObject');
buildDirPath = getfield(makertwObj,'BuildDirectory');
```

In R2013a, you can replace the above code with the following code, which returns the current build folder path.

```
buildDirPath = rtwprivate('get_makertwsettings',gcs,'BuildDirectory');
```

For more information about build hook functions, see Customize Build Process with STF_make_rtw_hook File.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2012b

Version: 8.3

New Features

Bug Fixes

Compatibility Considerations

Unified and simplified code interface for ERT and GRT targets

Previously, Simulink Coder software provided a static main program for GRT-based targets, *matlabroot*/rtw/c/grt/grt_main.c, that was distinct from the static main program that Embedded Coder software provided for ERT-based targets, *matlabroot*/rtw/c/ert/ert_main.c.

Beginning in R2012b, Simulink Coder software provides a unified static main program for both GRT- and ERT-based targets:

matlabroot/rtw/c/src/common/rt_main.c

Generated code for GRT-based models is simplified and more consistent with generated code for ERT-based models. As a result, GRT- and ERT-based models can now use a common static main program. The benefits for GRT-based models include:

- The generated rtModel structure has a minimal number of fields.
- Unused macros no longer appear in the generated code.
- Multitasking behavior is consistent between GRT and ERT generated code.

Note

- If you are using the pre-R2012a GRT call interface (by selecting the model option **Classic call interface**) with a static main program, use the static main program *matlabroot*/rtw/c/grt/classic_main.c as a reference point.
- The previous GRT and ERT static main program files, *matlabroot*/rtw/c/grt/grt_main.c and *matlabroot*/rtw/c/ert/ert_main.c, have been removed from the software and are replaced by the new simplified and classic static main program files, *matlabroot*/rtw/c/src/common/rt_main.c and *matlabroot*/rtw/c/grt/classic_main.c.
- The generated main program file for ERT targets is still named ert_main.c/cpp.
- If you have an Embedded Coder license, see also External mode support for ERT targets with static main in the Embedded Coder release notes.

Compatibility Considerations

If you use a GRT-based target with a static main program, and if you configure your models with the simplified call interface that was made available to GRT targets in

R2012a (that is, you do not use the model option **Classic call interface**), you must update your static main program to be compatible with the R2012b static main changes. Use the code in *matlabroot*/rtw/c/src/common/rt_main.c as an example. The following sections outline some of the key changes to look for.

Error status handling

In R2012a, GRT targets using the simplified call interface handled stop simulation requests (during MAT-file logging or External mode simulation) differently from ERT targets using the simplified call interface:

- For ERT targets, a stop simulation request caused the error status to be set to Simulation finished. The main program (ert_main.c) treated this error status as a non-error, while treating all other non-NULL status values as errors.
- For GRT targets, a stop simulation request did not cause the error status to be set (it remained NULL). The main program treated all non-NULL status values as errors.

Beginning in R2012b, the error status handling for GRT targets using the simplified call interface has been changed to match ERT targets using the simplified interface.

Unused macros

In R2012a, GRT targets using the simplified call interface generated macros differently from ERT targets using the simplified call interface:

- For ERT targets, the build process did not generate macros if they were not used in the generated code.
- For GRT targets, the build process unconditionally generated several macros that were not used in generated code.

Beginning in R2012b, the build process no longer unconditionally generates unused macros for GRT targets using the simplified call interface. The macros affected include:

- rtm* macros for accessing unused fields of the rtModel structure, such as ModelPtrs, StepSize, ChildSfunction, TPtr, and TaskTime
- IsSampleHit

Multitasking functions

In R2012a, GRT targets using the simplified call interface generated functions for multitasking differently from ERT targets using the simplified call interface:

- For ERT targets, the build process never generated the
  `rt_SimUpdateDiscreteEvents` function and, by default, never generated the
  `rate_monotonic_scheduler` function. (The `rate_monotonic_scheduler`
  function is for MathWorks internal use only.)

- For GRT targets, the build process generated the functions
  `rt_SimUpdateDiscreteEvents` and `rate_monotonic_scheduler` for
  multitasking.

Beginning in R2012b, the build process no longer generates the multitasking functions
`rt_SimUpdateDiscreteEvents` and `rate_monotonic_scheduler` for GRT targets
using the simplified call interface.

Convenient packNGo dialog for packaging generated code and artifacts

R2012b adds model configuration parameters for packaging generated code and artifacts
as part of a model build. The following new parameters are located on the **Code
Generation** pane of the Configuration Parameters dialog box:

- **Package code and artifacts** (`PackageGeneratedCodeAndArtifacts`) — Specify
  whether to automatically package generated code and artifacts for relocation.

- **Zip file name** (`PackageName`) — Specify the name of the `.zip` file in which to
  package generated code and artifacts for relocation.

If you select **Package code and artifacts**, the build process runs the `packNGo` function
after code generation to package generated code and artifacts for relocation. Selecting
Package code and artifacts also enables the **Zip file name** parameter for specifying
a `.zip` file name. The default file name is *model*`.zip`. (*model* represents the name of
the top model for which code is being generated.)

For more information, see Relocate Code to Another Development Environment.

Reusable code for subsystems shared by referenced models

In R2012b, you can configure a subsystem that is shared across referenced models to
generate code to the shared utilities folder. Code generation creates a standalone
function in the shared utilities folder that can be called by the generated code of multiple
referenced models.

To generate a single function for a reusable subsystem, the subsystem must be an active link to a subsystem in a library. For more information, see Code Reuse For Subsystems Shared Across Referenced Models.

Code generation for protected models for accelerated simulations and host targets

A protected model can include the generated code of the model. To create a protected model, right-click the referenced model and select **Subsystem & Model Reference > Create Protected Model for Selected Block** to open the Create Protected Model dialog box. You can select options that:

- Include the generated C code of the referenced model.
- Obfuscate the generated code.
- Create a protected model report.

You can then package the protected model, generated code, and protected model report for a third party to use for accelerated simulations and code generation. In R2012b, the file extension for protected models is `.slxp` (instead of the `.mdlp` extension in previous releases).

For more information, see Protect a Referenced Model and Package a Protected Model.

Reduction of data copies with buses and more efficient for-loops in generated code

Reduction of cyclomatic complexity with virtual bus expansion

In R2012b, code generation reduces cyclomatic complexity introduced by virtual bus expansion. This enhancement improves execution speed, reduces code size, and enables additional optimizations that reduce data copies and RAM consumption.

Simplifying for loop control statements

Improvements to `for` loops in the generated code include lifting invariance out of the `for` loop header and simplifying complex control statements in the `for` loop header. This enhancement improves execution speed and the readability of the generated code.

Unified rtiostream serial and TCP/IP target connectivity for all host platforms

Beginning in R2012b, Simulink Coder software provides unified `rtiostream` serial and TCP/IP target connectivity for all host platforms. Specifically, R2012b extends `rtiostream` serial connectivity to Linux and Macintosh host platforms; previously, only Windows host platforms were supported.

If you have implemented `rtiostream` serial connectivity for your embedded target environment, you can use `rtiostream` serial communication on any valid host to connect a Simulink model to your embedded target, using External mode or processor-in-the-loop (PIL) simulation.

Note Simulink Coder software provides `rtiostream` serial and TCP/IP target connectivity for all host platforms. If required, you can implement custom `rtiostream` connectivity—for example, to support a communication protocol other than serial or TCP/IP—for both the host platform and the embedded target environment.

Constant parameters generated as individual constants to shared location

Previously, constant parameters were generated to a model-specific structure, `rtConstP`, in the *model*`_data.c` file. If constant parameters are part of a model reference hierarchy or the model configuration parameter **Shared code placement** is set to `Shared location`, they are generated to a shared location. In R2012b, shared constant parameters are generated as individual constants to the `const_params.c` file in the `_sharedutils` folder. This code generation improvement generates less code and allows for subsystem code reuse across models. For more information, see Shared Constant Parameters for Code Reuse.

Code efficiency enhancements

The following code generation enhancements improve the efficiency of the generated code by:

- Removing a root-level outport data copy in the generated code when data is from a Stateflow chart. This enhancement reduces RAM and ROM consumption and improves execution speed.

- Removing a data copy for masked subsystems when a parameter is a matrix data type. This enhancement reduces RAM and ROM consumption and improves execution speed.

- Removing a limitation where the joint presence of initial value and function prototype control prevent removal of the root-level outport data copy in the generated code. The outport data copy is removed when the initial value is zero. This enhancement reduces RAM and ROM consumption and improves execution speed.

- Removing an unnecessary global variable generated by a For Each Subsystem or as a result from the selected configuration parameter **Pack Boolean data into bitfields**. In R2012b, the variable is removed from the global block structure which reduces global RAM.

Optimized code generation of Delay block

In R2011b, a new Delay block replaced the Integer Delay block. The Delay block now supports optimized code generation.

Search improvements in code generation report

Searching text in the code generation report highlights results and then scrolls to the first result. Press **Enter** to scroll through the subsequent search results. If the search returns no results, the background of the search box is highlighted red.

GRT template makefile change for MAT-file logging support

In R2012b, the template makefiles (TMFs) for GRT-based targets have been updated to better support the **MAT-file logging** (`MatFileLogging`) model option, which was added to the **Interface** pane of the Configuration Parameters dialog box for GRT targets in R2010b.

Compatibility Considerations

If you authored a TMF for a GRT-based target, you should update your TMF to better support the **MAT-file logging** option. If **MAT-file logging** is selected for a GRT model, your existing TMF will continue to work. But if **MAT-file logging** is cleared, compilation of the model code will fail unless your TMF is updated.

To update your TMF, do the following:

1 Add a makefile variable token for MAT-file logging to the TMF:

```
MAT_FILE        = |>MAT_FILE<|
```

2 Use this variable to create a `-D` define that is part of the compiler invocation. For example

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DRT -DNUMST=$(NUMST) \
        -DTID01EQ=$(TID01EQ) -DNCSTATES=$(NCSTATES) -DUNIX \
        -DMT=$(MULTITASKING) -DHAVESTDIO -DMAT_FILE=$(MAT_FILE)
```

For examples of this update, see the GRT-based TMFs provided with Simulink Coder, located at *matlabroot*/rtw/c/grt/grt_*.tmf.

Change for blocks that use TLC custom code functions in multirate subsystems

In earlier releases, blocks could use the TLC functions `LibSystem*CustomCode` to register custom code to be placed inside the `gcd` rate of a multirate subsystem. Beginning in R2012b, blocks that register custom code for this purpose must additionally register use of custom code with the Simulink software, using the `SimStruct` macro `ssSetUsingTLCCustomCodeFunctions`. Registering allows the Simulink engine to perform necessary adjustments to handle multiple rates for subsystems with custom code. Code generation will generate an error if all of the following conditions are true:

- An S-function uses `LibSystem*CustomCode` functions without registering their use to Simulink.
- The S-function is placed in a multirate subsystem.
- No nonvirtual block in the subsystem has a sample time equal to the `gcd` of the sample times in the system.

Compatibility Considerations

If you authored a block that uses any of the TLC `LibSystem*CustomCode` functions to register custom code to be placed inside multirate subsystem functions, the block now must register custom code use with the Simulink software. Modify the `mdlInitializeSizes` code in the block to call the `ssSetUsingTLCCustomCodeFunctions` macro, as shown below:

```
ssSetUsingTLCCustomCodeFunctions (S, 1);
```

Model rtwdemo_f14 removed from software

In R2012b, the example model `rtwdemo_f14` has been removed from the Simulink Coder software.

Compatibility Considerations

If you need an example model with similar content, open the Simulink example model `sldemo_f14` and configure it with a fixed-step solver. If you need an example GRT model that is configured for code generation, see the Simulink Coder models in the `rtwdemos` list.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2012a

Version: 8.2

New Features

Bug Fixes

Compatibility Considerations

Simplified Call Interface for Generated Code

In previous releases, GRT and GRT-based targets generated code with a GRT-specific call interface, using the model entry functions `model`, `MdlInitializeSizes`, `MdlInitializeSampleTimes`, `MdlStart`, `MdlOutputs`, `MdlUpdate`, and `MdlTerminate`. ERT and ERT derived targets, by default, generated code with a simplified call interface, using the model entry functions `model_initialize`, `model_step`, and `model_terminate`. (Additionally, model options could be applied to customize the simplified call interface, such as clearing **Single output/update function** or **Terminate function required**.)

In R2012a, GRT targets can now generate code with the same simplified call interface as ERT targets. This simplifies the task of interacting with the generated code. Target authors can author simpler `main.c` or `.cpp` programs for GRT targets. Also, it is no longer required to author different main programs for GRT and ERT targets.

To preserve compatibility with models, GRT-based custom targets, and GRT main modules created in earlier releases, R2012a provides the model option **Classic call interface** (`GRTInterface`), which is located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. If you select **Classic call interface**, code generation generates model function calls compatible with the main program module of the GRT target in models created before R2012a. If you clear the **Classic call interface** option, code generation generates the simplified call interface.

Note

- The **Classic call interface** (`GRTInterface`) option is available for both GRT-based and ERT-based models. For Embedded Coder users, it replaces the ERT model option **GRT-compatible call interface** (`GRTInterface`).

- For new GRT and ERT models, the **Classic call interface** option is cleared by default. New models use the simplified call interface.

- For GRT models created before R2012a, **Classic call interface** is selected by default. Existing GRT models can continue to use the pre-R2012a GRT-specific call interface.

Incremental Code Generation for Top-Level Models

R2012a provides the ability to omit unnecessary code regeneration from top model builds, allowing top models to be built incrementally. This can significantly reduce model build times.

Previously, each model build fully regenerated and compiled the top model code. Beginning in R2012a, the build process checks the structural checksum of the model to determine whether changes to the model require code regeneration. If code regeneration is required, the build process fully regenerates and compiles the model code, in the manner of earlier releases. However, if the generated code is found to be current with respect to the model, the build process does the following:

1 Skips model code regeneration.

2 Still calls build process hooks, including *STF_make_rtw_hook* functions and the post code generation command.

3 Reruns the makefile to make sure external dependencies are recompiled and relinked.

Additionally, command-line options exist for controlling or overriding the new build behavior. For more information, see Control Regeneration of Top Model Code.

Minimal Header File Dependencies with packNGo Function

The `packNGo` function, which packages model code files in a zip file for relocation, now by default includes only the minimal header files required in the zip file. The `packNGo` function now runs a preprocessor to determine the minimal header files required to build the code. Previously, `packNGo` included all header files found on the include path.

To revert to the behavior of previous releases, you can use the following form of the function:

```
>> packNGo(buildInfo,{'minimalHeaders',false})
```

ASAP2 Enhancements for Model Referencing and Structured Data

Ability to Merge ASAP2 Files Generated for Top and Referenced Models

R2012a provides the ability to merge ASAP2 files generated for top and referenced models into a single ASAP2 file. To merge ASAP2 files for a given model, use the function `rtw.asap2MergeMdlRefs`, which has the following syntax:

```
[status,info]=rtw.asap2MergeMdlRefs(topModelName,asap2FileName)
```

For more information, see Merge ASAP2 Files Generated for Top and Referenced Models

ASAP2 File Generation for Test Pointed Signals and States

ASAP2 file generation has been enhanced to generate ASAP2 MEASUREMENT records for the following data, without the need to resolve them to Simulink data objects:

- Test-pointed Simulink signals, usable inside reusable subsystems
- Test pointed Stateflow states, allowing you to monitor which state is active during real-time testing
- Test-pointed Stateflow local data
- Root-level inports and outports

Options to control ASAP2 record generation for structured data are defined in *matlabroot*/toolbox/rtw/targets/asap2/asap2/user/asap2setup.tlc:

- `ASAP2EnableTestPoints` enables or disables record generation for test pointed Simulink signals, test pointed Stateflow states, and test-pointed Stateflow local data (enabled by default)
- `ASAP2EnableRootLevelIO` enables or disables record generation for root-level inports and outports (disabled by default)

For more information, see Customize an ASAP2 File.

ASAP2 File Generation for Tunable Structure Parameters

ASAP2 file generation has been enhanced to generate ASAP2 CHARACTERISTIC records for tunable structure parameters. This allows you to tune structure parameters with ASAP2 tools and potentially manage large parameter sets.

For more information, see Customize an ASAP2 File.

Serial External Mode Communication Using rtiostream API

In R2012a, you can create a serial transport layer for Simulink external mode communication using the `rtiostream` API. For more information, see Create a Transport Layer for External Communication.

Improved Data Transfer in External Mode Communication

In Simulink External mode communication, the `rt_OneStep` function runs in the foreground and the `while` loop of the `main` function runs in the background. See Real-Time Single-Tasking Systems. Previously, with code generated for GRT and Embedded Coder bareboard ERT targets, data transfer between host and server was performed by functions within the model step function in `rt_OneStep`. The data transfer between host and server (in the foreground) would slow down model execution, potentially impairing real-time performance.

Now, the function that is responsible for data transfer between host and server (`rtExtModeOneStep`) is inserted in the `while` loop of the `main` function. As the execution of the `while` loop in the `main` function is a background task, real-time performance potentially is enhanced.

Changes for Desktop IDEs and Desktop Targets

- "Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE" on page 13-5
- "Limitation: Parallel Builds Not Supported for Desktop Targets" on page 13-5

Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE

Simulink Coder software now supports GCC 4.4 on host computers running Linux with Eclipse IDE. This support is on both 32-bit and 64-bit host Linux platforms.

If you were using an earlier version of GCC on Linux with Eclipse, upgrade to GCC 4.4.

Limitation: Parallel Builds Not Supported for Desktop Targets

The Simulink Coder product provides an API for MATLAB Distributed Computing Server™ and Parallel Computing Toolbox™ products. The API allows these products to

perform parallel builds that reduce build time for referenced models. However, the API does not support parallel builds for models whose **System target file** parameter is set to `idelink_ert.tlc` or `idelink_grt.tlc`. Thus, you cannot perform parallel builds for Desktop Targets.

Code Generation Report Enhancements

Post-build Report Generation

In previous releases, if you did not configure your model to create a code generation report, you had to build your model again to open the code generation report. You can now generate a code generation report after the code generation process completes without building your model again. This option is available on the model diagram **Tools** menu. After building your model, select **Tools** > **Code Generation** > **Code Generation Report** > **Open Model Report**. You can also open a code generation report after building a subsystem. For more information on creating and opening the code generation report, see Generate an HTML Code Generation Report.

Generate Code Generation Report Programmatically

At the MATLAB command line, you can generate, open, and close an HTML Code Generation Report with the following functions:

- `coder.report.generate` generates the code generation report for the specified model.
- `coder.report.open` opens an existing code generation report.
- `coder.report.close` closes the code generation report.

Searching in the Code Generation Report

You can now search within the code generation report using a search box in the navigation section. After entering text in the search box, the current page scrolls to the first match and highlights all of the matches on the page. To access the **Search** text box, press **Ctrl-F**.

New Reserved Keywords for Code Generation

The Simulink Coder software includes the following reserved keywords to the Simulink Coder Code Generation keywords list. For more information, see Reserved Keywords.

ERT	LINK_DATA_STREAM	NUMST	RT_MALLOC
HAVESTDIO	MODEL	PROFILING_ENABLED	TID01EQ
INTEGER_CODE	MT	PROFILING_NUM_SAMPLES	USE_RTMODEL
LINK_DATA_BUFFER_SIZE	NCSTATES	RT	VCAST_FLUSH_DATA

Improved MAT-File Logging

R2012a enhances Simulink Coder MAT-file logging to allow logging of multiple data points per time step, by reallocating buffer memory during target execution. Generated code logging results now match simulation results for blocks executing multiple times per step, such as blocks in an iterator subsystem. Previously, code generation issued a warning that the logged results for blocks executing in an iterator subsystem might not match the results from simulation.

rtwdemo_f14 Being Removed in a Future Release

The demo model `rtwdemo_f14` will be removed in a future release of Simulink Coder software.

Compatibility Considerations

In R2012a, you can still open `rtwdemo_f14` by entering `rtwdemo_f14` in the MATLAB Command Window. Going forward, transition to using `f14`, `sldemo_f14`, or a Simulink Coder model in the `rtwdemos` list.

New and Enhanced Demos

The following demos have been enhanced in R2012a:

Demo...	Now...
rtwdemo_asap2	• Illustrates ASAP2 file generation for test pointed signals and states. • Shows how to generate a single ASAP2 file from files for top and referenced models. • Generates STD_AXIS and FIX_AXIS descriptions for lookup table breakpoints.
rtwdemo_configuration_set	Shows how to use the Code Generation Advisor and the `Simulink.ConfigSet` saveAs method.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

14

R2011b

Version: 8.1

New Features

Bug Fixes

Compatibility Considerations

n-D Lookup Table Block Supports Tunable Table Size

The n-D Lookup Table block provides new parameters for specifying a tunable table size in the generated code.
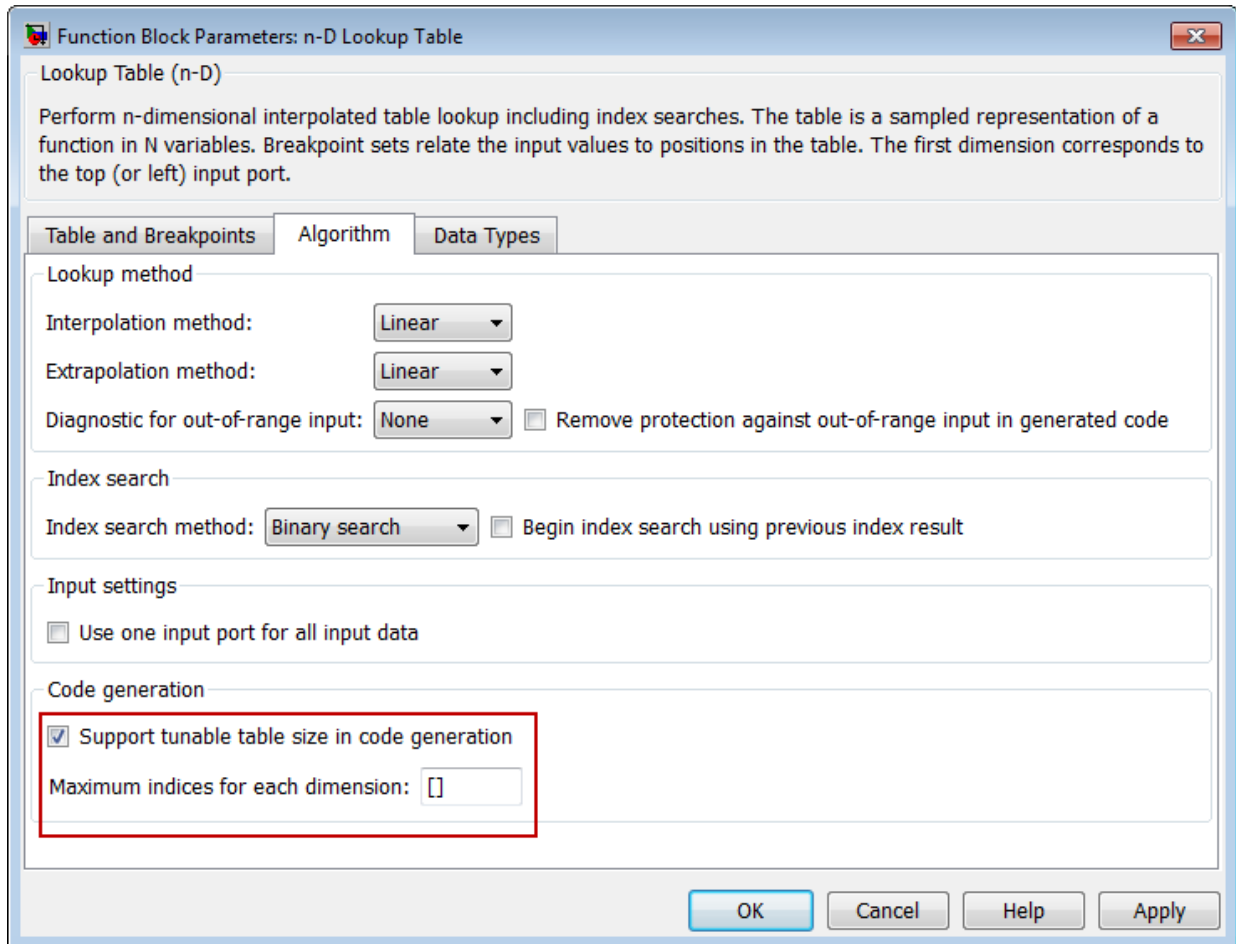
This enhancement enables you to change the size and values of your lookup table and breakpoint data without regenerating or recompiling the code.

Complex Output Support in Generated Code for the Trigonometric Function Block

In previous releases, the imaginary part of a complex output signal was always zero in the generated code for the Trigonometric Function block. In R2011b, this limitation no longer exists. Code that you generate for a function in this block now supports complex outputs.

Code Optimizations for the Combinatorial Logic Block

The Simulink Coder build process uses a new technique to provide more efficient code for the Combinatorial Logic block.

Benefits include:

· Reuse of variables

· Dead code elimination

· Constant folding

· Expression folding

For example, in previous releases, temporary buffers were created to carry concatenated signals for this block. In R2011b, the build process eliminates unnecessary temporary buffers and writes the concatenated signal to the downstream global buffer directly. This enhancement reduces the stack size and improves code execution speed.

Code Optimizations for the Product Block

The Simulink Coder build process provides more efficient code for matrix inverse and division operations in the Product block. The following summary describes the benefits and when each benefit is available:

Benefit	Small matrices (2-by-2 to 5-by-5)	Medium matrices (6-by-6 to 20-by-20)	Large matrices (larger than 20-by-20)
Faster code execution time	Yes, much faster	No, slightly slower	Yes, faster
Reduced ROM and RAM usage	Yes, for real values	Yes, for real values	Yes, for real values

Benefit	Small matrices (2-by-2 to 5-by-5)	Medium matrices (6-by-6 to 20-by-20)	Large matrices (larger than 20-by-20)
Reuse of variables	Yes	Yes	Yes
Dead code elimination	Yes	Yes	Yes
Constant folding	Yes	Yes	Yes
Expression folding	Yes	Yes	Yes
Consistency with MATLAB Coder	Yes	Yes	Yes

Compatibility Considerations

In the following cases, the generated code might regress from previous releases:

- The ROM and RAM usage increase for complex input data types.
- For blocks configured with 3 or more inputs of different dimensions, the code might include an extra buffer to store temporary variables for intermediate results.

Enhanced MISRA C Code Generation Support for Stateflow Charts

In previous releases, the code generated to check whether or not a state in a Stateflow chart was active included a line that looked something like this:

```
if (mdl_state_check_er_DWork.is_active_c1_mdl_state_c == 0)
```

In R2011b, that line has been modified to:

```
if (mdl_state_check_er_DWork.is_active_c1_mdl_state_c == 0U)
```

This enhancement supports MISRA C® 2004, rule 10.1.

Change for Constant Sample Time Signals in Generated Code

In previous releases, constant sample time signals were initialized even if the **Data Initialization** field of their custom storage class was set to `None`.

In R2011b, constant sample time signals using a custom storage class for which the **Data Initialization** field is set to `None` will not be initialized for non-conditionally executed systems in generated code.

Compatibility Considerations

If you use such constant time signals, you will notice that they are not initialized in the generated code in R2011b. To enable their initialization, change the setting of the **Data Initialization** field of their custom storage class from `None` to another value.

New Code Generation Advisor Objective for GRT Targets

In R2011b, `Execution efficiency` is now available as a Code Generation Advisor objective for models with generic real-time (GRT) targets. You can use this objective to achieve faster code execution times for your models. For more information, see Application Objectives.

Improved Integer and Fixed-Point Saturating Cast

Simulink Coder software now eliminates more dead branches in both integer and fixed-point saturation code.

Generate Multitasking Code for Concurrent Execution on Multicore Processors

The Simulink Coder product extends the concurrent execution modeling capability of the Simulink product. With Simulink Coder, you can generate multitasking code that uses POSIX threads (Pthreads) or Windows threads for concurrent execution on multicore processors running Linux, Mac OS X, or Windows.

See Configuring Models for Targets with Multicore Processors.

Changes for Desktop IDEs and Desktop Targets

- "New Target Function Library for Intel IPP/SSE (GNU)" on page 14-6
- "Support Added for Single Instruction Multiple Data (SIMD) with Intel Processors" on page 14-6

New Target Function Library for Intel IPP/SSE (GNU)

This release adds a new Target Function Library (TFL), `Intel IPP/SSE (GNU)`, for the GCC compiler. This library includes the Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE) code replacements.

Support Added for Single Instruction Multiple Data (SIMD) with Intel Processors

This release adds support for the SIMD capabilities of the Intel processors. The use of SIMD instructions increases throughput compared to traditional Single Instruction Single Data (SISD) processing.

The `Intel IPP/SSE (GNU)` TFL (code replacement library) optimizes generated code for SIMD.

The performance of the SIMD-enabled executable depends on several factors, including:

- Processor architecture of the target
- Optimized library support for the target
- The type and number of TFL replacements in the generated algorithmic code

Evaluate the performance of your application before and after using the TFL.

To use the SIMD capabilities with GCC and Intel processors, enable the `Intel IPP/SSE (GNU)` TFL. See Code Replacement Library (CRL).

Reserved Keyword UNUSED_PARAMETER

The Simulink Coder software adds the `UNUSED_PARAMETER` macro to the reserved keywords list for code generation. To view the complete list, see Reserved Keywords. In R2011b, code generation now defines `UNUSED_PARAMETER` in `rt_defines.h`. Previously, it was defined in `model_private.h`.

Target API for Verifying MATLAB® Distributed Computing Server™ Worker Configuration for Parallel Builds

R2010b added the ability to use remote workers in MATLAB® Distributed Computing Server™ configurations for parallel builds of model reference hierarchies. This introduced the possibility that parallel workers might have different configurations,

some of which might not be compatible with a specific Simulink Coder target build. For example, the required compiler might not be installed on a worker system.

R2011b provides a programming interface that target authors can use to automatically check the configuration of parallel workers and, if the parallel workers are not set up as required, take action, such as throwing an error or reverting to sequential builds. For more information, see Support Model Referencing in the Simulink Coder documentation.

For more information about parallel builds, see Reduce Build Time for Referenced Models in the Simulink Coder documentation.

License Names Not Yet Updated for Coder Product Restructuring

The Simulink Coder and Embedded Coder license name strings stored in `license.dat` and returned by the `license ('inuse')` function have not yet been updated for the R2011a coder product restructuring. Specifically, the `license ('inuse')` function continues to return `'real-time_workshop'` for Simulink Coder and `'rtw_embedded_coder'` for Embedded Coder, as shown below:

```
>> license('inuse')
matlab
matlab_coder
real-time_workshop
rtw_embedded_coder
simulink
>>
```

The license name strings intentionally were not changed, in order to avoid license management complications in situations where Release 2011a or higher is used alongside a preR2011a release in a common operating environment. MathWorks plans to address this issue in a future release.

For more information about using the function, see the `license` documentation.

New and Enhanced Demos

The following demos have been enhanced in R2011b:

Demo...	Now...
`rtwdemo_pmsmfoc_script`	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

R2011a

Version: 8.0

New Features

Bug Fixes

Compatibility Considerations

Coder Product Restructuring

- "Product Restructuring Overview" on page 15-2
- "Resources for Upgrading from Real-Time Workshop or Stateflow Coder" on page 15-3
- "Migration of Embedded MATLAB Coder Features to MATLAB Coder" on page 15-4
- "Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder" on page 15-4
- "User Interface Changes Related to Product Restructuring" on page 15-5
- "Simulink Graphical User Interface Changes" on page 15-5

Product Restructuring Overview

In R2011a, the Simulink Coder product combines and replaces the Real-Time Workshop® and Stateflow Coder products. Additionally,

- The Real-Time Workshop facility for converting MATLAB code to C/C++ code, formerly referred to as Embedded MATLAB® Coder, has migrated to the new MATLAB Coder product.
- The previously existing products Embedded IDE Link™ and Target Support Package™ have been integrated into the new Simulink Coder and Embedded Coder products.

The following figure shows the R2011a transitions for C/C++ code generation related products, from the R2010b products to the new MATLAB Coder, Simulink Coder, and Embedded Coder products.
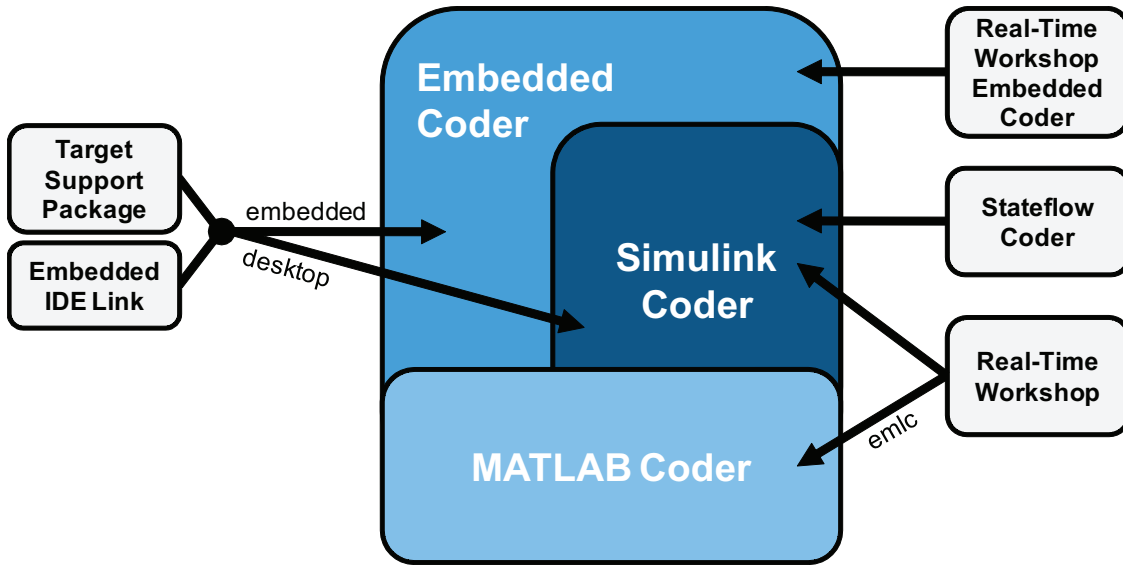
The following sections address topics related to the product restructuring.

Resources for Upgrading from Real-Time Workshop or Stateflow Coder

If you are upgrading to Simulink Coder from Real-Time Workshop or Stateflow Coder, review information about compatibility and upgrade issues at the following locations:

- *Release Notes for Simulink Coder* (latest release), "Compatibility Summary" section
- In the Archived documentation on the MathWorks web site, select R2010b, and view the following tables, which are provided in the release notes for Real-Time Workshop and Stateflow Coder:

  - *Compatibility Summary for Real-Time Workshop Software*
  - *Compatibility Summary for Stateflow and Stateflow Coder Software*

  These tables provide compatibility information for releases up through R2010b.

- If you use the Embedded IDE Link or Target Support Package capabilities that now are integrated into Simulink Coder and Embedded Coder, go to the Archived documentation, select R2010b, and view the corresponding tables for each product:

  - *Compatibility Summary for Embedded IDE Link*

- *Compatibility Summary for Target Support Package*

You can also refer to the rest of the archived documentation, including release notes, for the Real-Time Workshop, Stateflow Coder, Embedded IDE Link, and Target Support Package products.

Migration of Embedded MATLAB Coder Features to MATLAB Coder

In R2011a, the MATLAB Coder function `codegen` replaces the Real-Time Workshop function `emlc`. The `emlc` function still works in R2011a but generates a warning, and will be removed in a future release. For more information, see Migrating from Real-Time Workshop emlc Function in the MATLAB Coder release notes.

Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder

In R2011a, the capabilities formerly provided by the Embedded IDE Link and Target Support Package products have been integrated into Simulink Coder and Embedded Coder. The follow table summarizes the transition of the Embedded IDE Link and Target Support Package hardware and software support into coder products.

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
Embedded IDE Link	Altium® TASKING		x
	Analog Devices® VisualDSP ++®		x
	Eclipse IDE	x	x
	Green Hills® MULTI®		x
	Texas Instruments™ Code Composer Studio™		x
Target Support Package	Analog Devices Blackfin®		x
	ARM		x
	Freescale™ MPC5xx		x
	Infineon® C166®		x
	Texas Instruments C2000™		x
	Texas Instruments C5000™		x

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
	Texas Instruments C6000™		x
	Linux OS	x	x
	Windows OS	x	
	VxWorks RTOS		x

User Interface Changes Related to Product Restructuring

Some user interface changes were made as part of merging the Real-Time Workshop and Stateflow Coder products into Simulink Coder. They include:

- Changes to code generation related elements in the Simulink Configuration Parameters dialog box
- Changes to code generation related elements in Simulink menus
- Changes to code generation related elements in Simulink blocks, including block parameters and dialog boxes, and block libraries
- References to Real-Time Workshop and Stateflow Coder and related terms in error messages, tool tips, demos, and product documentation replaced with references to the latest software

Simulink Graphical User Interface Changes

Where...	Previously...	Now...
Configuration Parameters dialog box	**Real-Time Workshop** pane	**Code Generation** pane
Model diagram window	**Tools > Real-Time Workshop**	**Tools > Code Generation**
Subsystem context menu	**Real-Time Workshop**	**Code Generation**

Where...	Previously...	Now...
Subsystem Parameters dialog box	Following parameters on main pane: • **Real-Time Workshop system code** • **Real-Time Workshop function name options** • **Real-Time Workshop function name** • **Real-Time Workshop file name options** • **Real-Time Workshop file name (no extension)**	On new **Code Generation** pane and renamed: • **Function packaging** • **Function name options** • **Function name** • **File name options** • **File name (no extension)**

Changes for Desktop IDEs and Desktop Targets

Feature Support for Desktop IDEs and Desktop Targets

The Simulink Coder software provides the following features as implemented in the former Target Support Package and Embedded IDE Link products:

- Automation Interface
- External Mode
- Multicore Deployment with Rate Based Multithreading
- Makefile Generation (XMakefile)

Note You can only use these features in the 32-bit version of your MathWorks products. To use these features on 64-bit hardware, install and run the 32-bit versions of your MathWorks products.

Location of Blocks for Desktop Targets

Blocks from the former Target Support Package product and Embedded IDE Link product are now located in Simulink Coder under Desktop Targets.

```
Simulink Coder
    Asynchronous
    Custom Code
    Desktop Targets
        Host Communication
        Operating Systems
            Linux
            Windows
```
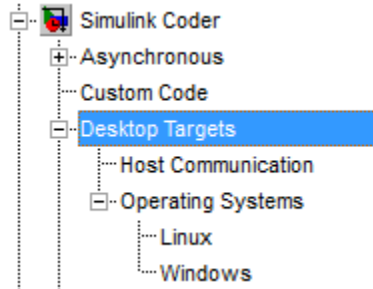
Desktop Targets includes the following types of blocks:

- Host Communication
- Operating Systems

    - Linux
    - Windows

Location of Demos for Desktop IDEs and Desktop Targets

Demos from the former Target Support Package product and Embedded IDE Link product now reside under Simulink Coder product help. Click the expandable links, as shown.

Simulink Coder
- Getting Started
- User's Guide
- Reference
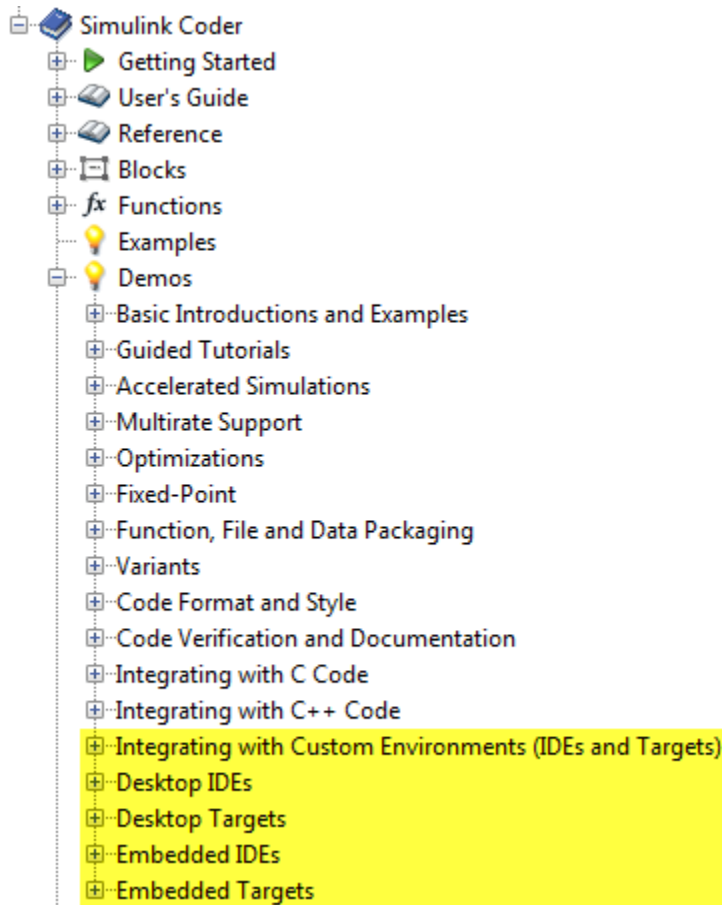- Blocks
- Functions
- Examples
- Demos
  - Basic Introductions and Examples
  - Guided Tutorials
  - Accelerated Simulations
  - Multirate Support
  - Optimizations
  - Fixed-Point
  - Function, File and Data Packaging
  - Variants
  - Code Format and Style
  - Code Verification and Documentation
  - Integrating with C Code
  - Integrating with C++ Code
  - Integrating with Custom Environments (IDEs and Targets)
  - Desktop IDEs
  - Desktop Targets
  - Embedded IDEs
  - Embedded Targets

Multicore Deployment with Rate Based Multithreading

You can deploy rate-based multithreading applications to multicore processors running Windows and Linux. This feature potentially improves performance by taking advantage of multicore hardware resources.

Also see the Running Target Applications on Multicore Processors user's guide topic.

Code Optimizations for Discrete State-Space Block, Product Block, and MinMax Block

The Simulink Coder build process uses a new technique to provide more efficient code for the following blocks:

- Discrete State-Space
- Product (element-wise matrix operations)

Benefits include:

- Reuse of variables
- Dead code elimination
- Constant folding
- Expression folding

For example, in previous releases, temporary buffers were created to carry concatenated signals for these blocks. In R2011a, the build process eliminates unnecessary temporary buffers and writes the concatenated signal to the downstream global buffer directly. This enhancement reduces the stack size and improves code execution speed.

The build process also provides more efficient code for the MinMax block. In R2011a, expression folding is enhanced with several local optimizations that enable more aggressive folding. This enhancement improves code efficiency for foldable matrix operations.

Ability to Share User-Defined Data Types Across Models

In previous releases, user-defined data types that were shared among multiple models generated duplicate type definitions in the `model_types.h` file for each model. R2011a provides the ability to generate user-defined data type definitions into a header file that can be shared across multiple models, removing the need for duplicate copies of the data type definitions. User-defined data types that you can set up in a shared header file include:

- Simulink data type objects that you instantiate from the classes `Simulink.AliasType`, `Simulink.Bus`, `Simulink.NumericType`, and `Simulink.StructType`

• Enumeration types that you define in MATLAB code

For more information, see Share User-Defined Data Types Across Models in the Simulink Coder documentation.

C API Provides Access to Root-Level Inputs and Outputs

The C API now provides programmatic access to root-level inputs and outputs. This allows you to log and monitor the root-level inputs and outputs of a model while you run the code generated for the model. To generate C API code for accessing root-level inputs and outputs at run time, select the model option **Generate C API for: root-level I/O**.

Macros for accessing C API generated structures are located in *matlabroot*/rtw/c/src/rtw_capi.h and *matlabroot*/rtw/c/src/ rtw_modelmap.h, where *matlabroot* represents your MATLAB installation root.

For more information, see Generate C API for: root-level I/O and Data Interchange Using the C API in the Simulink Coder documentation.

ASAP2 File Generation Supports Standard Axis Format for Lookup Tables

In previous releases, ASAP2 file generation for lookup table blocks supported the Fix Axis and Common Axis formats, but not the Standard Axis format, a format in which axis points are global in code but not shared among tables. R2011a adds support for Standard Axis format.

For more information, see Define ASAP2 Information for Lookup Tables in the Simulink Coder documentation.

ASAP2 File Generation Enhancements for Computation Methods

Custom Names for Computation Methods

In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model oriented representation. R2011a adds the MATLAB function getCompuMethodName, which you can use to customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file

readability, or names that meet organizational requirements. For more information, see Customize Computation Method Names in the Simulink Coder documentation.

Ability to Suppress Computation Methods for FIX_AXIS When Not Required

Versions 1.51 and later of the ASAP2 specification state that for certain cases of lookup table axis descriptions (integer data type and no doc units), a computation method is not required and the Conversion Method parameter must be set to the value `NO_COMPU_METHOD`. Beginning in R2011a, you can control whether or not computation methods are suppressed when not required, using the Target Language Compiler (TLC) option `ASAP2GenNoCompuMethod`. For more information, see Suppress Computation Methods for FIX_AXIS in the Simulink Coder documentation.

Lookup Table Block Option to Remove Input Range Checks in Generated Code

When the breakpoint input to a Prelookup, 1-D Lookup Table, 2-D Lookup Table, or n-D Lookup Table block always falls within the range of valid breakpoint values, you can disable range checking in the generated code. By selecting **Remove protection against out-of-range input in generated code** on the block dialog box, your code can be more efficient.
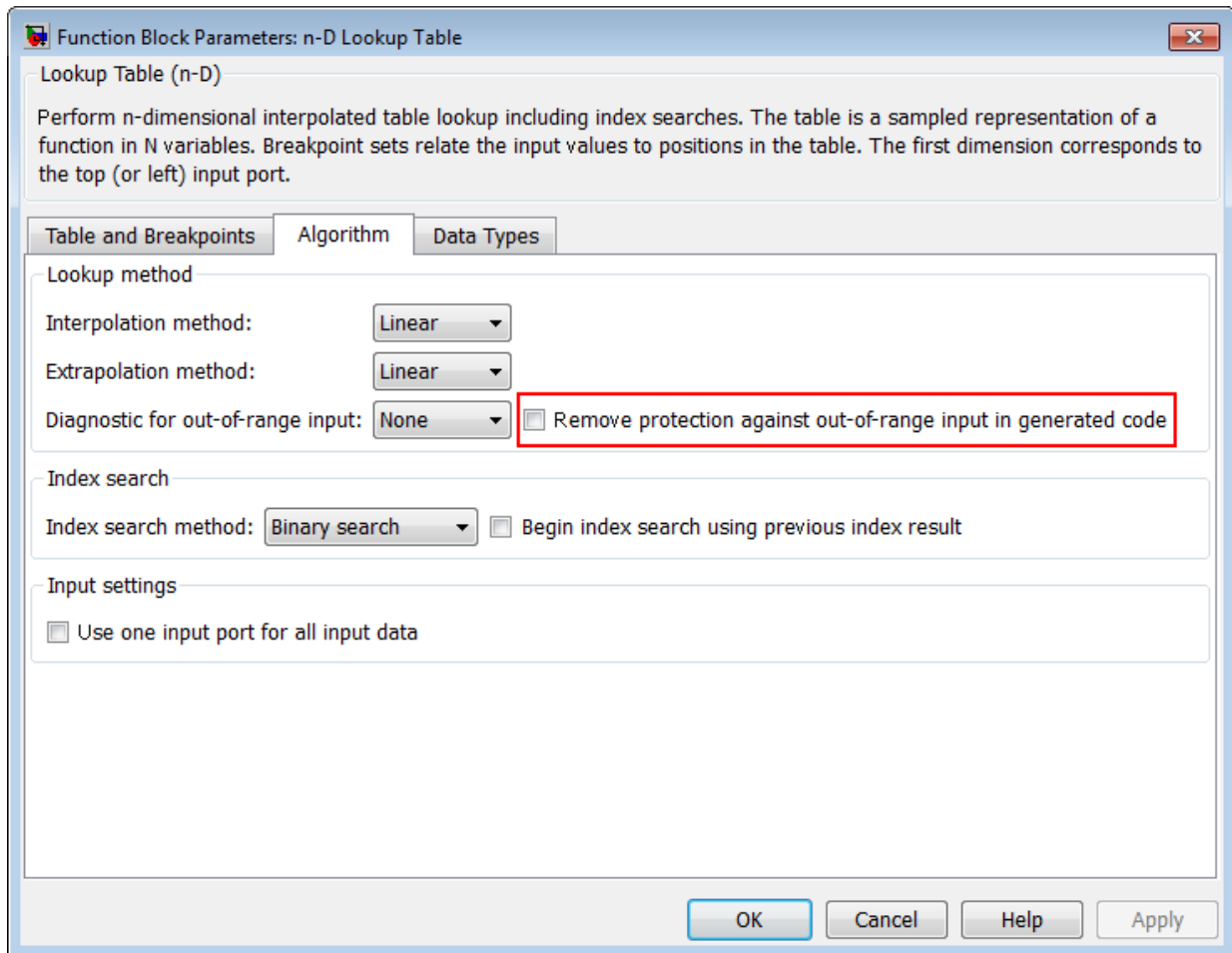
Reentrant Code Generation for Stateflow Charts That Use Events

When you generate code for Stateflow charts that use events, the code does not use a global variable to keep track of the currently active event. Elimination of this global variable enables the code to be reentrant, which allows you to:

- Deploy your code in multithreading environments
- Share the same algorithm with different persistent data

• Compile code that uses function variables that are too large to fit on the stack

In previous releases, reentrant code generation was not possible for charts that used events.

Redundant Check Code Removed for Stateflow Charts That Use Temporal Operators

When you generate code for Stateflow charts that use temporal operators, the code excludes redundant checks for `tick` events and input events that are always true. This enhancement enables the code to be more efficient and applies to temporal operators `after`, `before`, `at`, `every`, and `temporalCount`.

In previous releases, the code generated for a temporal logic expression such as `after(x,tick)` would check for two conditions:

```
(event == tick) && (counter > x)
```

In R2011a, the code generated for `after(x,tick)` checks only for when the temporal counter exceeds x:

```
(counter > x)
```

This enhancement does not apply when a chart with multiple input events has super-step semantics enabled.

Support for Multiple Asynchronous Function Calls Into a Model Block

Simulink and Simulink Coder software now support multiple asynchronous function calls into a Model block. This capability relies in part on the new Asynchronous Task Specification block.

The Asynchronous Task Specification block, in combination with a root-level Inport block, allows you to specify an asynchronous function-call input to a Model block. After placing this block at the output port of each root-level Inport block that outputs a function-call signal, select **Output function call** on the **Signal Attributes** pane of the Inport block. The Inport block then accepts function-call signals. You can use Asynchronous Task Specification blocks to specify parameters for the asynchronous task associated with the respective Inport blocks.

Note Use the new function call API, `LibBlockExecuteFcnCall`, to make function calls from an asynchronous source block to reference model destination blocks.

Note The demo model `rtwdemo_async_mdlreftop` shows how you can simulate and generate code for asynchronous events on a real-time multitasking system, using asynchronous function calls as Model block inputs.

Changes to ver Function Product Arguments

The following changes have been made to `ver` function arguments related to code generation products:

- The new argument `'simulinkcoder'` returns information about the installed version of the Simulink Coder product.

- The argument `'rtw'` works but now returns information about Simulink Coder instead of Real-Time Workshop. The software also displays the following message:

  ```
  Warning: Support for ver('rtw') will be removed in a future release.
  Use ver('simulinkcoder') instead.
  ```

- The argument `'coder'`, which previously returned information about the installed version of the Stateflow Coder product, no longer works. The software displays a "not found" warning.

For more information about using the function, see the `ver` documentation.

Compatibility Considerations

If a script calls the `ver` function with the `'rtw'` argument or the `'coder'` argument, update the script appropriately. For example, you can update the `ver` call to use the `'simulinkcoder'` argument, or remove the `ver` call.

Updates to Target Language Compiler (TLC) Semantics and Diagnostic Information

Updates to TLC simplifies semantics and produces diagnostic information when using the scope resolution operator (`::`) and built-in function `EXISTS(::)`.

- If *var* can not be resolved in global scope, `::var` errors out
- If *var* can only be resolved in local scope, `EXISTS(::var)` returns false
- Diagnostic information highlights problematic TLC coding

For more information, see Introducing the Target Language Compiler.

Change to Terminate Function for a Target Language Compiler (TLC) Block Implementation

Previously, the code generator attempted to execute the `Terminate` function from the TLC implementation of a block, even if the function did not exist. Now, the code generator only attempts to execute a `Terminate` function if it is defined in the TLC implementation of a block. In the case where the TLC implementation of a block includes a secondary TLC file, which includes a `Terminate` function, that `Terminate` function no longer executes.

New and Enhanced Demos

The following demos have been added in R2011a:

Demo...	Shows How You Can...
`rtwdemo_async_mdlreftop`	Simulate and generate code for asynchronous events on a real-time multitasking system, using asynchronous function calls as Model block inputs.

The following demos have been enhanced in R2011a:

Demo...	Now...
`vipstabilize_fixpt_beagleboard` `videostabilization_host_templ`	Use the new Video Capture block to simulate or capture a video input signal in the Video Stabilization demo.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.