

Embedded Coder[®] Release Notes



MATLAB[®]&SIMULINK[®]



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.
1 Apple Hill Drive
Natick, MA 01760-2098

Embedded Coder[®] Release Notes

© COPYRIGHT 2011–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

Code Generation from MATLAB Code	1-2
Removal of initialized but unused class properties in generated C/C++ code	1-2
Reduction of violations for MISRA C:2012 and AUTOSAR C++14 rules in generated code	1-2
Model Architecture and Design	1-4
Deploy models as components that include comprehensive service interface support	1-4
Control interface of generated code using data and service interface configurations in Embedded Coder Dictionary	1-6
Component service interface support for callable entry-point functions ...	1-7
Component service interface support for target platform data receiver and data sender services	1-7
Component service interface support for target platform data transfer service	1-7
Component service interface support for target platform timer service ...	1-8
Component service interface support for target platform parameter tuning and measurement services	1-8
Modeling guidelines and Model Advisor checks for component deployment using a service interface configuration	1-9
Code Interface Configuration and Integration	1-10
Map model elements to service interfaces	1-10
Dimension preservation of multidimensional arrays for GetSet and access function storage classes	1-10
Support for root level inports and outports as pointer members in C++ generated code	1-11
Functionality being removed or changed	1-11
Code Generation	1-13
Select code interface configuration using new configuration parameter .	1-13
Generate an example main program parameter not available for models configured with a service interface configuration	1-14
Generated C++11 example main program simplified	1-14
Include requirement comments in the generated code	1-15
Files and folders for target platform services	1-15
Code interface report for service interfaces	1-16
Generate code for Reusable custom storage classes with symbolic dimension inputs	1-16
New \$X naming rule token	1-16

Example models attached to examples and renamed	1-17
New Simulink Model Advisor check for numeric efficiency	1-18
Code replacement validation detects ambiguous overflow and rounding modes	1-18
Deployment	1-20
Retrieve metadata about service interface by using code descriptor programming interface	1-20
Target Language Compiler search functions for regular expressions	1-20
Introducing Embedded Coder Support Package for Linux Applications . .	1-21
Calibration File Customization	1-21
Performance	1-23
Data Store Memory block reuse in reusable subsystems inside While Iterator subsystems	1-23
Removed redundant multirate block output buffers	1-24
Buffer reuse optimization for referenced models	1-25
Improved cache efficiency of generated code containing loop distribution, interchange, and reversal	1-25
Generate SIMD code for Discrete FIR Filter block	1-28
Improved function argument generation eliminates extra global variable assignment	1-28
SIMD code for bitwise and shift operations	1-30
Code replacement for lookup tables that support differently sized table and breakpoint objects	1-30
Code execution profiling for models that use GRT system target files . . .	1-31
Task scheduling visualization with XCP external mode simulations	1-31
Optimized bandwidth usage during XCP external mode profiling	1-31
Verification	1-32
SIL or PIL block workflow	1-32
Reusable subsystems with input signals that map to const variables	1-32
Check bug reports for issues and fixes	1-33

R2022a

Code Generation from MATLAB Code	2-2
Removal of unused class properties in generated C/C++ code	2-2
Reduction of violations for MISRA C:2012, MISRA C++:2008, and AUTOSAR C++14 rules in generated code	2-2
Stack usage profiling for code generated from MATLAB code	2-3
Identification of performance bottlenecks in generated code	2-3
Model Architecture and Design	2-4
Symbolic dimension inputs for Squeeze block	2-4
Embedded Coder Dictionary interface improvements	2-4

Code Interface Configuration and Integration	2-5
Control code interface generated for models by specifying deployment types	2-5
Changes to class namespaces and default class name in C++ generated code	2-6
Calibration file customization	2-6
Memory section mapping for grouped entry-point functions	2-6
Code Generation	2-8
Regular expression token decorators to modify certain tokens	2-8
Improved comments for code that initializes instance-specific values for model arguments	2-8
New parentheses level for MISRA standard compliance and code readability	2-9
Improved code readability by adding "U" suffix to unsigned integer constants	2-10
Changes to initialization	2-10
AUTOSAR C++14 Rule A12-4-2 violation resolution	2-10
AUTOSAR C++14 Rule A12-0-1 violation resolution	2-11
Removed redundant S-function output buffer	2-11
C++ Code Generation for client-server interfaces	2-13
C++ code generation for new Message Triggered Subsystem and Message Polling Subsystem blocks to control event-triggered execution of messages	2-13
CustomSymbolStrUtil parameter available for C++ and AUTOSAR code generation	2-13
Functionality being removed or changed	2-13
Deployment	2-15
TLC function STRNREP for string replacement	2-15
Configuration Parameter dialog box no longer lists VxWorksExample as a setting for parameter Target operating system	2-15
Texas Instruments C2000: Support for Texas Instruments F28003x processor	2-15
Texas Instruments C2000: Support for F28M35x (C28x), F28M36x (C28x), and ARM Cortex-M3 Core	2-15
Embedded Coder Support Package for STMicroelectronics Discovery Boards renamed to Embedded Coder Support Package for STMicroelectronics STM32 Processors	2-16
Support for STMicroelectronics STM32F7xx, STM32G4xx, and STM32H7xx-based Boards	2-16
Performance	2-17
SIMD code for reduction operations	2-17
Code replacement for circular buffer index for Delay blocks	2-18
Code replacement for lookup tables by using index search algorithm parameter	2-18
Code generation by inlining redundant function calls	2-19
Stack usage profiling for code generated from Simulink models	2-20
Identification of performance bottlenecks in generated code	2-20
Code execution profiling for multiple Model blocks	2-20

Verification	2-21
Unit-testing atomic subsystem code in AUTOSAR software component . .	2-21
Functionality being removed or changed	2-21
Check bug reports for issues and fixes	2-22

R2021b

Code Generation from MATLAB Code	3-2
Communication I/O information display during SIL or PIL execution	3-2
Visualization of task scheduling	3-2
Reduction of violations for MISRA C++:2008 and AUTOSAR C++14 rules in generated code	3-2
Model Architecture and Design	3-3
Built-in storage class for multi-instance data	3-3
Symbolic dimension inputs for Bitwise Operator, Saturation, and Data Type Propagation blocks	3-3
Code Interface Configuration and Integration	3-4
Storage class with pointer data access in Embedded Coder Dictionary . . .	3-4
Unstructured Embedded Coder Dictionary storage class application to model reference root I/O	3-4
Embedded Coder Dictionary storage class application to signals and parameters with symbolic dimensions	3-4
Changes to model hierarchy requirements	3-4
Calibration file customization	3-5
TLC code storage classes in default mapping	3-5
Configure additional properties from the Code Mappings editor	3-5
View In Bus Element and Out Bus Element blocks in a hierarchy in the Code Mappings editor	3-6
Configuring C/C++ function prototypes for subsystems not recommended	3-6
Reusable storage class in Code Mappings editor	3-7
Generated C++ model class name can be the model name	3-7
Code Generation	3-8
Accessibility of step entry-point functions generated for models designed for multitasking and concurrency streamlined	3-8
Code view for MATLAB Function block	3-9
Enhanced code to reduce MISRA C:2012 Rule 10.3 and Directive 4.1 violations	3-10
Changes to generated C++ header files	3-10
const member functions for C++ class interface	3-10
Minimized variable visibility for C++ code	3-11
Image data by using OpenCV class cv::Mat	3-12
Shared types and parameters storage in same header file	3-13

Bidirectional traceability in Code view by default	3-14
Deployment	3-15
New TLC variable OverrideSampleERTMain for disabling generation of example main program	3-15
Texas Instruments C2000: Code generation support for Configurable Logic Block (CLB) and CLB X-Bar in Embedded Coder Support Package for Texas Instruments C2000 Processors	3-15
Texas Instruments C2000: External Mode Simulation Using XCP on CAN Interface	3-15
Support for STMicroelectronics STM32F4xx-based Boards	3-15
Performance	3-16
Generation of SIMD code by using new configuration parameter	3-16
Image Processing Toolbox functions enhanced with multithreading and algorithm improvements	3-16
Reduced data copies for models that have Bus Creator blocks	3-17
SIMD optimization for more integer data types	3-19
Root output initialization code performance improvements	3-20
Readability improvement for root output initialization code	3-21
Optimize code by unrolling parallel for-loops	3-22
Improved common subexpression elimination	3-22
Optimized SIMD code that performs fused multiply add operations	3-23
Redundant data copies elimination by reusing S-function block buffers ..	3-24
Optimized code for models containing referenced models	3-26
Nonstatic data class member initialization of instance-specific parameters	3-27
Code replacement for trigonometric functions that use lookup table approximation	3-28
Verification	3-29
Communication I/O information display during SIL or PIL simulation ...	3-29
Signal and state logging for SIL and PIL simulations	3-29
LDRA tool suite code coverage analysis	3-29
Check bug reports for issues and fixes	3-30

R2021a

Code Generation from MATLAB Code	4-2
Multiple signature for software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution	4-2
Reduction of violations for MISRA C++:2008 and AUTOSAR C++14 rules in generated code	4-2
Format generated code by using clang-format	4-3
Model Architecture and Design	4-4

Step entry-point functions generated for rate-based and concurrent execution models declared in model.h	4-4
Code Interface Configuration and Integration	4-5
C++ class interface configuration by using a code mappings workflow . . .	4-5
Instance specific parameter support for C++ class interfaces	4-5
Auto data initialization for new storage classes	4-5
Dimension preservation of multidimensional arrays for Simulink.Bus object	4-6
Calibration file generation	4-6
Code configuration for data dictionary defaults	4-6
ASAP2 system target file being removed	4-6
Functionality being removed or changed	4-6
Code Generation	4-8
Enhanced generated code to reduce MISRA C:2012 Rule 12.2 violations	4-8
Removal of typedef from C++ struct definitions	4-8
Braced variable initialization for C++ 11 library	4-8
Code generation and SIL or PIL simulations for protected models from R2018b and later releases	4-9
Performance	4-10
Code execution profiling information in Code view	4-10
Visualization of task scheduling	4-10
Removal of instrumentation overhead from execution-time profiling by using target package	4-10
Enhanced code for models containing mask blocks or Data Store Memory blocks	4-11
GCC ARM Cortex-A code replacement library contains other ARM libraries	4-11
Multithreading capabilities for more Image Processing Toolbox functions	4-12
Improved cache performance of generated code containing distributed loop nests	4-13
Improved expression folding in generated code	4-15
Improved root output buffer reuse to reduce data copies	4-16
Reduced data copies for blocks with bus inputs and outputs	4-17
Verification	4-19
PIL target connectivity with debugger	4-19
Unit-tests for generated code from subsystems within code from parent model	4-19
Code view in SIL/PIL Manager	4-19
Check bug reports for issues and fixes	4-21

Model Architecture and Design	5-2
Determine programmatically if model or data dictionary contains Embedded Coder Dictionary	5-2
Symbolic dimension inputs for Add, Subtract, Sum of Elements, and Sum blocks	5-2
Improved readability for preprocessor conditionals in generated code . . .	5-2
Memory section configurations for atomic subsystems	5-3
Code Interface Configuration and Integration	5-4
Streamlined model data configuration for code generation	5-4
Dimension preservation of multidimensional arrays for individual model elements	5-5
Custom data type configuration and modification	5-6
Functionality being removed or changed	5-6
Code Generation	5-12
Static code metrics for C99 and C++ libraries	5-12
Code generation using multiple code replacement libraries	5-12
Static reusable subsystem functions for C++ class interface	5-12
Name mangling of functions inside MATLAB Function block code	5-13
Generated code enhanced to reduce MISRA C:2012 Rule 13.5 violations	5-13
Generate static code metrics report programmatically	5-13
Code generation and SIL or PIL simulations for protected models from R2018b and later releases	5-14
Cross-release code integration for non-finite numbers in shared utility code	5-14
Enhanced traceability between variables and modeling elements in Code view	5-14
Same name error message for Simulink.Bus object and data in C++ code	5-15
Standardization of header guards in header files	5-15
Deployment	5-16
Texas Instruments C2000: Support of UDP and Hardware Interrupt Blocks for F2838x (ARM Cortex-M4) Processor in Embedded Coder Support Package for Texas Instruments C2000 Processors	5-16
Texas Instruments C2000: Support Code Generation for SDFM Module in F2807x, F2837x, F28004x and F2838x Processors for Embedded Coder Support Package for Texas Instruments C2000 Processors	5-16
Performance	5-17
SIMD code generated using Intel AVX-512 code replacement library	5-17
Improved cache performance of generated code that has loop interchange	5-18
SIMD vectorization of loops in Simulink models	5-20
Generated code optimization through SIMD for integer data type	5-21

Enhanced Image Processing Toolbox functions in generated code	5-22
Distribution of execution times for generated code internal functions . . .	5-24
Hardware timer for code execution profiling during PIL simulations	5-24
Caching of array elements to scalar variables reduces computations in generated code	5-24
Verification	5-26
Target connectivity for PIL simulations	5-26
SIL and PIL testing of reusable library subsystems	5-26
Signal and state logging for SIL and PIL simulations	5-26
Removal of top-model SIL and PIL limitations	5-27
SIL/PIL Manager settings	5-27
Functionality being removed or changed	5-28
Check bug reports for issues and fixes	5-29

R2020a

Code Generation from MATLAB Code	6-2
Model Architecture and Design	6-3
Function arguments to match graphical block interface for nonreusable subsystems	6-3
External I/O visibility for C++ class interface	6-3
C++ message-based communication provides length argument for service functions	6-3
Zero initialization code model configuration parameters disabled for C++ class interface	6-4
Code Interface Configuration and Integration	6-5
Alias property of Simulink.CoderInfo renamed Identifier	6-5
Model type definitions within class namespace	6-5
Dimension preservation of multidimensional arrays for Data Store Memory blocks, states, and signals	6-5
Storage class change for model workspace parameter converted to Simulink.Parameter	6-6
Functionality being removed or changed	6-7
Code Generation	6-8
std::array support in C++ code generation	6-8
Allow Arguments for non-reusable subsystems with C++	6-8
\$R token in Memory Sections of Embedded Coder Dictionary	6-8
Reduction in identifier collisions in model reference hierarchy	6-9
Static code metrics in Code view without code generation report	6-9
SIL or PIL simulations with protected model AUTOSAR code from R2018b or later	6-10
Storage classes on signal lines	6-10
Removal of preprocessor guards in C++ code	6-10

Removal of configuration parameter limitations for Simulink string code generation	6-10
Deployment	6-12
FFT code replacement library (CRL) support for ARM Cortex-A and Cortex-M processors	6-12
Performance	6-13
Data Store Memory block reuse to reduce data copies in subsystems . . .	6-13
Buffer reuse optimization for multidimensional arrays	6-15
Logical operators conversion to bitwise operators in generated code . . .	6-16
Enhanced Image Processing Toolbox functions in generated code	6-17
Capture main code execution profiling metrics on target hardware	6-18
Efficient code for model-reference builds in presence of function prototype control	6-18
Symbolic dimension support for Reshape blocks	6-19
Check bug reports for issues and fixes	6-20

R2019b

Code Generation from MATLAB Code	7-2
Customize C/C++ code file names generated from MATLAB code	7-2
Custom type definitions from external header files	7-2
Disable generation of initialize function	7-2
Function profiling for SIL and PIL execution	7-2
Model Architecture and Design	7-3
Symbolic dimension support for Stateflow Data	7-3
Generate C++ Code for Software Compositions with Message-Based Communication	7-3
Cut, copy, and paste code definitions in Embedded Coder Dictionary	7-3
Configure Embedded Coder Dictionary programmatically	7-3
Data, Function, and File Definition	7-5
Generated code calibration and monitoring through XCP and third-party tools	7-5
Argument specifications not required for Function Caller blocks that invoke scoped Simulink functions	7-5
Implicit validation occurs when configuring C function prototypes	7-5
Map storage classes defined in Embedded Coder Dictionary to nonreusable subsystems with separate data	7-5
Code Mappings Editor Changes	7-6
Function <code>rtw.asap2SetAddress</code> extracts DWARF debug symbols from binaries compiled using MinGW compiler	7-6
Code Generation	7-7

Optimized C++ generated code for reusable functions	7-7
Embedded Coder contextual tabs on the Simulink Toolstrip	7-7
Simulink strings through standard C++ string library	7-8
C++ static_cast in generated code	7-10
Inline traceability for variable and type definitions	7-10
Deployment	7-12
Performance	7-13
Improved Data Store Memory block reuse to reduce data copies	7-13
SIMD vectorization for loops	7-14
Optimized code execution speed for Ceiling, Floor, Minimum and Maximum SIMD intrinsic functions	7-14
SIMD vectorization for loops without compile-time bounds	7-16
SIMD for row-major operations	7-18
Specification of upper constraint limit for symbolic dimensions	7-19
Parameter expression saturation	7-21
Changes to zero initialization code model configuration parameter default settings	7-21
Enhanced code execution profiling report	7-22
Elimination of unused writes to global variables	7-23
Verification	7-26
SIL/PIL Manager	7-26
Code coverage information in Code view	7-26
Data logging and signal viewer block support for export function models	7-26
SIL/PIL for AUTOSAR Classic Software Components containing referenced models	7-26
Traceability for hidden blocks	7-27
Check bug reports for issues and fixes	7-28

R2019a

Code Generation from MATLAB Code	8-2
Custom Data Type Replacement: Specify custom data type names for MATLAB data types	8-2
Model Architecture and Design	8-3
Library-based code generation for reusable subsystem function interfaces	8-3
AUTOSAR Blockset product replaces Embedded Coder Support Package for AUTOSAR Standard	8-3
MISRA C:2012 and Secure Coding checks to improve compliance of generated code	8-3
Data, Function, and File Definition	8-5

Preserve array dimensions for root-level inports and outports in generated code	8-5
Custom storage class with different code generation settings for single-instance and multi-instance data	8-5
Code generation definitions in multiple packages from Embedded Coder Dictionary	8-6
Storage classes with get and/or set data access functions in Embedded Coder Dictionary	8-6
Code definitions from local and shared Embedded Coder Dictionaries	8-6
Code packaging support for model arguments	8-7
Model argument support for top models	8-7
C entry-point function prototype preview and customization in the Code Mapping Editor	8-7
Code Generation	8-9
Code metrics information in code view	8-9
Cross-release code import without opening previous release	8-9
Import of code from previous release for code generation-only workflow	8-10
Maximum line width for generated code	8-10
Symbolic dimension support for %roll directive	8-11
Embedded Coder contextual tabs on the Simulink Toolstrip Tech Preview	8-12
Deployment	8-13
Embedded Coder Support Package for PX4 Autopilots: Generate, build and deploy Simulink models on Pixhawk flight controllers	8-13
DSP System Toolbox Support Packages for ARM Cortex -A and ARM Cortex -M Processors will be removed	8-13
Performance	8-14
Reusable custom storage classes across referenced models	8-14
Parallelization of execution of for-loops	8-15
Subsystem output with internal signals for buffer reduction	8-16
Optimized code execution speed for Single Instruction, Multiple Data (SIMD) intrinsic division operation	8-17
Optimized code for Switch Case blocks	8-19
Removal of instrumentation overhead from execution-time profiling	8-22
Improvement in execution speed through common subexpression elimination	8-22
Data copy reduction in function calls	8-24
Code generation for lookup table optimization	8-25
Verification	8-26
Check bug reports for issues and fixes	8-27

Code Generation from MATLAB Code	9-2
Column Limit in Generated Code: Generate more readable code by controlling line wrapping	9-2
Static Code Metrics On Demand: Run static code metrics analysis when needed after code generation	9-2
Single Instruction, Multiple Data (SIMD) Support: Generate Intel SSE/AVX intrinsic in MATLAB Coder	9-3
Model Architecture and Design	9-4
Multi-Instance Code Generation: Generate multi-instance code for top and referenced models that are based on rates, exported functions, or rates and exported functions	9-4
Code Preview in Embedded Coder Dictionary: Verify pseudocode preview as you select data, function, and memory section properties	9-4
Embedded Coder Dictionary Mapping Control: Define storage classes that restrict mappings to parameters or signals	9-4
Embedded Coder Dictionary Version Handling: Use and export code definitions saved in previous releases with models created in later releases	9-4
AUTOSAR Run-Time Calibration: Map internal signals, states, and model workspace parameters to AUTOSAR component memory and internal parameters for calibration	9-5
AUTOSAR Memory Sections: Use SwAddrMethods to control memory placement of AUTOSAR runnable functions and internal data	9-6
AUTOSAR XML Import and Export: Round trip imported arxml file structure and control packaging of new elements	9-6
AUTOSAR XML Import: Changes to ArTypedPerInstanceMemory and StaticMemory import behavior	9-7
Obsolete AUTOSAR signal and state map functions removed	9-7
MISRA C:2012 and Secure Coding Standards: Improve compliance of generated code by using updated Model Advisor checks	9-8
Data, Function, and File Definition	9-9
Individual Function Mappings in Code Mapping Editor: Override default function mappings with individual function mappings	9-9
Function Interface Control: Access Configure C Step Function Interface dialog box from Code Mapping Editor in code perspective	9-9
Function Interface Control: Configure step functions for multi-instance, rate-grouped, single-tasking models	9-10
Shared Default Code Configurations for Data and Functions: Share default code configuration settings between models	9-10
Storage Class on Root-Level I/O: Access global data and functions in multi- instance models	9-10
Code Generation	9-11
Code View in Code Perspective: View generated code directly in Code Perspective	9-11

Data Coherency: Generate one variable for each Data Store read and write operation	9-11
AUTOSAR Code Generation: Automatically generate AUTOSAR platform data types in C code	9-11
Data Type Replacement: Specify replacement types for 64-bit integers . .	9-12
Multi-Dimensional Arrays: Preserve array dimensions for parameters and lookup tables in generated code	9-12
Hardware Implementation Parameters: ProdHWDeviceType and TargetHWDeviceType are case-insensitive	9-14
Enumerated Types: Optimizations in generated code	9-14
Deployment	9-16
Texas Instruments C2000: Use DMA and CAN blocks for all supported C28x devices with the addition of DMA for F28x7x/F28004x and CAN for F28004x	9-16
Code Generation Assumptions: Use standalone workflow to run checks .	9-16
Build Process: Library and header files for model reference hierarchy are not copied	9-16
Build Process: MATLAB_INCLUDES is not required in custom template makefiles	9-17
STM32F7 Tuning and Monitoring: Perform external mode simulation on STM32F7 for parameter tuning and signal monitoring by using XCP over TCP/IP or UART (Serial)	9-17
Performance	9-18
Execution-Time Profiling: Specify profiling granularity through model-wide and block-specific controls	9-18
Global Variable Caching: Reduce access for global variable arrays with custom storage classes	9-18
Data Copy Reduction: Eliminate unnecessary data copies for Mux blocks	9-20
Enhanced Buffer Reuse: Buffer reuse across the boundary of an Iterator subsystem	9-22
Code Replacement: Optimize generated code with SIMD and row-major order support and code replacement enhancements	9-24
Inplace Optimization for Assignment Blocks: Reduce data copies for Assignment blocks	9-25
Execution Speed: Eliminate redundant subexpressions	9-26
Single Instruction, Multiple Data (SIMD) Intrinsics: Generate code with optimized load and store operations for multidimensional signals and square root operations	9-28
Code Generation Report: Generate static code metrics reports faster . . .	9-30
Functionality Being Removed or Changed	9-30
Cache Efficiency: Store global block signal and state data operating at the same rate in one data structure	9-30
Verification	9-33
SIL and PIL Simulations: Advanced custom storage classes support	9-33
SIL and PIL Simulations: Support for imported grouped custom storage classes	9-33
Model Block SIL and PIL: Accelerator mode SIM target is not built	9-33
Check bug reports for issues and fixes	9-34

Code Generation from MATLAB Code	10-2
Interactive Traceability: Visualize mapping between MATLAB code and C code	10-2
Polyspace Integration: Verify C/C++ code generated with MATLAB Coder by using simplified workflow	10-2
Changes to Setup for MISRA C Compliance: Disable dynamic memory allocation and set C standard math library to C99 (ISO)	10-2
Model Architecture and Design	10-4
Embedded Coder Dictionary: Create custom code generation definitions for data and functions	10-4
Multi-Instance Code Generation: Apply more control when generating reusable, reentrant code	10-4
Variant Blocks Usability Enhancement: Generate Preprocessor Conditionals by using MATLAB variables as variant controls	10-5
MISRA C:2012 Compliance and Deviation Considerations: Guidance for evaluating your generated code for compliance with MISRA C:2012 directives and rules	10-5
Modeling Checks: Improve compliance of generated code by using Model Advisor check for MISRA C:2012	10-6
AUTOSAR Release 4.3: Import and export AUTOSAR XML schema version 4.3	10-6
AUTOSAR Perspective: Map and configure software components by using Code Mapping Editor and AUTOSAR Dictionary	10-6
AUTOSAR XML Import and Export: Round-trip ComSpecs, import bitfield CompuMethods, export interface variation points, and automate more element creation	10-8
AUTOSAR Signal Invalidation Block: Specify invalidation policy and initial value directly as block parameters	10-11
AUTOSAR Basic Software: Use array and bus data types with NvMServiceCaller operations	10-11
Obsolete AUTOSAR functions removed	10-12
Data, Function, and File Definition	10-14
Function-Prototype Control: Configure step function name with void void interface	10-14
Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model	10-14
GetSet Custom Storage Class Enhancement: Improved readability for an array of buses	10-15
Local Storage Class: Preserve local variables with Localizable storage class	10-16
Accurate Header File Extension: Generate correct #include statements for imported data types	10-16
Macro Access: Get data through a macro that your code defines	10-16
Tokens for Memory Sections: Use \$N token instead of identifier	10-17
Parameter Initialization: Statically initialize tunable parameters from system constants and other macros	10-17

Model-Scoped Parameter Objects: Use FileScope to prevent name clashes between parameters in different models	10-17
File Packaging of Generated Code for Global Simulink Function Blocks: Code for function body placed in model.c	10-18
Identifiers: Represent name of storage class in identifier naming rules by using new token \$G	10-18
Functionality Being Removed or Changed	10-19
Code Generation	10-20
Code Perspective: Customize Simulink desktop for code generation workflows	10-20
Rate Transition Block Code Customization: Separate Rate Transition block code and data from algorithm code and data	10-20
Generated Files: Customize generated file names with new token \$E . .	10-21
Hardware Implementation Settings: Inaccurate values corrected	10-22
Cross-Release Code Integration: Reuse referenced model code with instance-specific parameters	10-22
Cross-Release Code Integration: Import and simulate AUTOSAR code . .	10-22
Traceability Comments: Specify Simulink identifier in comments for Simulink blocks, Stateflow objects, and MATLAB Function blocks . . .	10-22
Newline Style: Customize linefeed character irrespective of the operating system	10-23
Export Functions: Generate ScratchModel file containing a Model block	10-23
Deployment	10-24
Build Process: Specify toolchain for template makefile	10-24
Build Process Status for Parallel Builds: View and interact with build process status for parallel builds of referenced model hierarchies . .	10-24
TI C2000 IPC Block: Support for Inter-Processor Communications for F2837xD in TI C2000 Support Package	10-24
C2000 F28004x: Support for peripherals in Texas Instruments C2000 Support Package	10-24
STM32F7 Audio: Multiple channel Mic-In, Line-In, and Speaker out for STM32F769I-Discovery in STM32 Support Package	10-24
STM32F7 External Mode: Support for TCP/IP and Serial Communication for STM32F769I-Discovery board in STM32 Support Package	10-24
External Mode Simulation: Upload execution-time metrics through XCP transport layer	10-25
Performance	10-26
Single Instruction, Multiple Data (SIMD) Intrinsics: Generate code with optimized load and store operations for use with Intel processors with SSE/AVX support	10-26
Preprocessor Conditionals: Obtain better readability of generated code for variant systems	10-27
Buffer Reuse: Prioritize buffer reuse based on signal labels in model diagram	10-28
Configuration Set: New location and layout for optimization model configuration parameters	10-29
Data Copy Reduction: Generate code with fewer data copies for writes to structure fields and matrix elements and for control flow patterns . .	10-30

Code Size Reduction: Eliminate identical functions in the generated code	10-34
Code Replacement: Optimize generated code with SIMD and row-major order support and improved library header file packaging	10-36
Execution Speed: Move invariant code containing global variables out of for loops	10-36
Verification	10-39
PIL Simulation: Verify initial values of global variables	10-39
Check bug reports for issues and fixes	10-40

R2017b

Code Generation from MATLAB Code	11-2
Setup for MISRA C Compliance: Configure code generation parameters to increase compliance with MISRA C:2012 guidelines	11-2
SIL/PIL Execution Performance: Speed up SIL or PIL execution by disabling constant input checking and global data synchronization	11-2
Execution-Time Profiling: Display time units in code execution profiling report	11-2
Default Case for Switch Statements: Increase generated code compliance with coding standards	11-2
Model Architecture and Design	11-3
Function Interfaces: Generate multi-instance functions from export-function models and control scope of Simulink functions	11-3
AUTOSAR Compositions and Basic Software: Import AUTOSAR compositions and simulate diagnostic and memory services	11-4
AUTOSAR Sender-Receiver Communication: Model AUTOSAR queued send and receive using Simulink messages	11-4
MISRA C: 2012 Modeling Checks: Improve compliance of generated code by using new MISRA C: 2012 standards checks	11-4
Modeling Support for Secure Coding Standards: Check model for compliance with secure coding requirements in CERT C, CWE, ISO/IEC TS 17961 standards to improve security of generated code	11-6
Code Reuse: Generate reusable code for subsystems that contain data objects with imported custom storage classes	11-7
Data, Function, and File Definition	11-8
Storage Class for Model Workspace Parameters: Apply custom storage classes to parameter objects in a model workspace	11-8
Custom Storage Class Simplification: Default removed from drop-down lists	11-8
Code Generation	11-9

Cross-Release Code Integration: Reuse code from models containing model references, global I/O, data stores, and parameters	11-9
Cross-Release Code Integration: Run all workflow tasks from current release	11-9
AUTOSAR Run-Time Calibration: Measure and calibrate signal and discrete state data using arTypedPerInstanceMemory	11-9
Stateflow Element Traceability: Obtain enhanced inline traceability . . .	11-10
Stateflow Objects and MATLAB User Comments: Configure comments flexibly	11-10
Enhanced Shared Utilities Naming: Customize the names of shared utility functions that are inside MATLAB Function blocks	11-11
Checksum Length: Specify the character length of the \$C token	11-11
Code Style: Generate static keyword for locally scoped functions	11-11
Configuration Parameters Dialog Box: View your model and code generation configuration parameters in unified dialog box with search capability	11-11
Improved Readability of the Generated Code: Include parentheses around compound expressions containing right-shift operators	11-13
Deployment	11-14
AUTOSAR Support Package: Run live-script examples for AUTOSAR compositions and Basic Software	11-14
Support Package renamed to Embedded Coder Support Package for Intel SoC Devices	11-14
Support Package renamed to Embedded Coder Support Package for Xilinx Zynq Platform	11-14
Removed Support for Wind River VxWorks Hardware	11-14
Performance	11-15
RAM Reduction: Reduce data copies in For Each subsystems and reuse buffers of different sizes	11-15
Reusable Storage Class: Specify reusable custom storage classes anywhere on a path	11-18
Execution Speed: Eliminate redundant subexpressions	11-18
Execution Speed: Convert data copies to pointer assignments for more modeling patterns	11-19
Execution Speed: Move invariant code out of for loops	11-22
Block Reordering for Improved Execution Efficiency: Change block execution order to enable buffer reuse and loop fusion	11-26
MATLAB Function Block Buffer Reuse: Perform inplace assignment with root I/O	11-26
Execution-Time Profiling: Display time units in code execution profiling report and Simulation Data Inspector	11-27
memcpy and memset Optimization: Generate more efficient code for variable-size arrays	11-27
Data Copy Reduction: Generate fewer data copies at function call sites	11-29
Code Replacement: Apply MustHaveZeroNetBias and SlopesMustBeTheSame properties for fixed-point operator code replacement	11-30
Enumerated Data Types Optimization: Improve the efficiency of the generated code for enumerated data types	11-30
Verification	11-33

Multiple Processor SIL/PIL Testing: Perform SIL or PIL component tests on different processors simultaneously	11-33
SIL Simulation: Simplified configuration of hardware implementation settings	11-33
SIL/PIL Configuration: Parent model code coverage, execution-time profiling, and SIL debugging settings apply to Model blocks with Top-model code interface	11-33
Hardware Implementation Settings: SIL checks relaxed for data type sizes and byte ordering	11-34
Check bug reports for issues and fixes	11-35

R2017a

Code Generation from MATLAB Code	12-2
SIL and PIL execution improvements for MATLAB Coder	12-2
Verification of PIL target connectivity configuration	12-2
Code Replacement for MATLAB Coder: Create code replacement library entries for target implementations that require data alignment	12-2
Model Architecture and Design	12-3
AUTOSAR arxml File Import: Flexibly model imported periodic, asynchronous, and initialization runnables	12-3
AUTOSAR DESC elements populate Simulink Description fields	12-3
External mode code generation for a model containing inline variant blocks	12-4
Code generation support for Variant Subsystems containing global signals	12-4
Preprocessor conditionals guard content inside and outside of function-call site	12-5
Data, Function, and File Definition	12-7
Function Interface: Return nonvoid type for scalar output of reusable functions	12-7
Utility to generate Simulink representations of struct and enum types defined by external C code	12-9
Code Generation	12-10
Cross-Release Code Integration: Reuse model reference code generated from previous releases	12-10
Code Replacement for Cast and Multiply Operations: Detect overflow and rounding mode equivalence for increased matches and code efficiency	12-10
More information in code generation report summary	12-10
Code Interface Report: Includes entry-point function for code generated from Reset Function block	12-11
Shared utility memory section associated with subfunctions	12-11
Inline traceability for generated code	12-11

Clear file section content from TLC file	12-12
Identifier case control with token decorators and custom text token \$U	12-12
Name change for AUTOSAR local temporary variables	12-13
Additional checks against MISRA C:2012 guidelines in Code Generation Advisor	12-13
Deployment	12-14
TI Code Composer Studio (CCS): Generate projects for CCS versions 5 and 6 with Embedded Coder Target for TI C2000	12-14
Customize generated makefiles for S-Functions	12-14
Release notes and workflow overview documentation added to AUTOSAR support package	12-14
SPI and I2C blocks added to TI C2000 support package	12-14
CCS v3.3 IDE automation support for TI C2000 has been removed	12-15
Real-time multitasking profiling for TI C2000	12-15
TCP and UDP blocks added to STMicroelectronics STM32F746G-Discovery board	12-15
MATLAB Coder PIL with STMicroelectronics STM32F4-Discovery Board	12-15
External Mode and PIL supported over TCP/IP by STMicroelectronics STM32F746G-Discovery board	12-17
Linux Support: Connect to ARM Cortex-M processor on Linux platform	12-17
ARM Cortex-R optimized code	12-17
Develop a Target for ARM Cortex-R processors	12-17
Support for Wind River VxWorks RTOS will be removed	12-18
Performance	12-19
Data Copy Reduction: Generate fewer data copies and use less RAM for buses, data stores, and model blocks	12-19
Code Efficiency: Improve loop fusion for Sum of Elements blocks and generate less code for temporal logic in Stateflow	12-26
Data copy reduction for Merge blocks	12-28
More instances of buffer reuse for blocks and subsystems in a chain ...	12-31
Improved buffer reuse due to changes in block execution order	12-34
More efficient code for Bus Creator blocks	12-35
Buffer reuse for Variant Source blocks	12-36
Verification	12-38
SIL and PIL Testing: Log signals inside exported functions and stream signals to Simulation Data Inspector during simulation	12-38
Verification of PIL target connectivity configuration	12-38
Check bug reports for issues and fixes	12-40

R2016b

Code Generation from MATLAB Code	13-2
---	-------------

Static code metrics report for C++ code	13-2
Verification of size_t and ptrdiff_t hardware settings	13-2
Verification of PIL target connectivity configuration	13-2
Optimization for array indexing in loops	13-2
Reduction of the Intel Performance Primitives (IPP) code replacement libraries (CRL)	13-3
Model Architecture and Design	13-4
AUTOSAR Basic Software (BSW) Services: Simulate BSW including Diagnostic Event Manager (DEM) and NVRAM Manager (NvM)	13-4
AUTOSAR Parameters: Model STD_AXIS and COM_AXIS lookup table parameters, export SwRecordLayouts, and apply SwAddrMethods ...	13-4
AUTOSAR startup, reset, and shutdown modeling	13-6
AUTOSAR external trigger event communication	13-6
AUTOSAR support for JMAAB model architecture	13-7
AUTOSAR ExplicitReceiveByVal data access mode for receiver ports ...	13-8
AUTOSAR ModeSenderPorts and ModeSwitchPoints for application mode management	13-8
AUTOSAR reference element definitions for sharing among components and services	13-9
ERT Target Code Generation: Remove unreachable reset and disable functions to reduce dead code	13-9
Conditional compile time check for imported macros with ImportedDefine custom storage class	13-10
Additional guarding of global data for variant systems	13-11
Data, Function, and File Definition	13-14
Simulink Function Code Interface: Configure generated C/C++ function interfaces for Simulink Function and Function Caller blocks	13-14
ERT default value for configuration parameter ParameterTunabilityLossMsg	13-14
Code Generation	13-16
Cross-Release Code Integration: Reuse code generated from earlier releases	13-16
Compound Operation Code Replacement: Replace "Multiply Shift Right Arithmetic" and "Multiply Divide" in generated code with a single custom operation	13-17
ARXML import/export and C code generation for latest AUTOSAR 4.2 and 3.2 standard revisions	13-17
AUTOSAR code replacement library enhancements	13-17
Static code metrics report for C++ code	13-17
Static code metrics data produced by Polyspace	13-18
Streamlined report pane for easier model configuration	13-18
Improved traceability between model and code	13-18
Code replacement enhancements	13-19
\$I macro changed for argument names used as input and output	13-19
Improved compliance with MISRA C:2012 Rules 10.1, 10.5, and 10.8 ..	13-19
Improved compliance with MISRA AC AGC Rule 12.6	13-21
Use default installation folder on Windows system with ReFS file system	13-22
Deployment	13-24

Cortex-M7 Target Support Package: Generate code for STM32F746G-Discovery Board	13-24
Added Embedded Coder Support Package for ARM Cortex-R Processors	13-24
Improved External mode over serial communication	13-25
New blocks added to TI's C2000 support package	13-25
Change in name and the base product for the FRDM-K64F and the FRDM-KL25Z support packages	13-25
Support for TI's C5000 DSPs has been removed	13-25
Support for TI's C6000 has been removed	13-25
Support for Wind River VxWorks RTOS will be removed	13-25
Support for idelink_ert.tlc will be removed	13-26
Performance	13-27
Data Reuse and Memory Reduction: Reuse global data for nonreusable subsystems and reduce data copies with user-specified buffers	13-27
Code Optimizations: Generate more efficient code with select-assign-iterator pattern and matrix padding operations	13-29
Display of code execution times for model component	13-33
More efficient code for array element assignments	13-33
Loop fusion for nested for loops	13-35
More efficient initialization code for root-level inports	13-36
More efficient code for Boolean expressions	13-39
Verification	13-41
Verification of size_t and ptrdiff_t hardware settings	13-41
Verification of PIL target connectivity configuration	13-41
Signal range checking in SIL and PIL simulations	13-41
SIL and PIL block support for Simulink Function and Function Caller blocks	13-41
Check bug reports for issues and fixes	13-42

R2016a

Code Generation from MATLAB Code	14-2
Export data by using ExportedDefine storage class	14-2
SIL execution returns standard output and standard error streams	14-2
Model Architecture and Design	14-3
Compile-Time Dimensions: Generate compiler directives (#define) for implementing signal dimensions	14-3
Compile-Time Variants: Generate compiler directives (#if) for variant choices specified with Variant Source and Variant Sink blocks	14-3
C++ Code Generation: Use referenced models with multitasking, export-functions, and virtual buses	14-4
MISRA C:2012 Compliance: Check block names and Assignment blocks by using the Model Advisor	14-4

AUTOSAR Round Trip: Automate model additions for update and merge of ARXML files	14-4
Comment change in generated code	14-5
Variants in AUTOSAR component modeling	14-5
AUTOSAR DataReceivedEvents for receiver ports in ImplicitReceive data access mode	14-7
AUTOSAR LiteralPrefix for enumerations in IncludedDataTypeSets	14-7
Programmatic validation and synchronization of AUTOSAR model configurations	14-7
Data, Function, and File Definition	14-8
In/Out Arguments: Specify same variable name for in/out arguments of MATLAB Function and Model blocks	14-8
Custom Storage Class Type AccessFunction	14-11
Creation of custom storage classes for macros defined by compiler options	14-11
Generation of ERT S-functions that represent variant controls as preprocessor conditionals	14-11
Code Generation	14-13
Default style C++ interface replaces the void-void style C++ interface	14-13
Compiler warning limitation removed for portable word sizes in SIL simulations	14-13
AUTOSAR arxml round trip	14-14
Improved AUTOSAR library support for Mfx functions	14-15
AUTOSAR target no longer supports building wrapper subsystem as AUTOSAR SW-Component	14-15
Root model name in generated identifier for shared utility files	14-16
Improved configuration parameter defaults for Embedded Coder targets	14-16
Streamlined code generation panes for easier model configuration	14-17
Build button removed from Configuration Parameters dialog box	14-20
Improved web view for code generation report	14-21
Dependent parameters not added to custom code generation objective	14-21
Removal of leading underscore character in macro type definitions	14-21
Deployment	14-23
Hardware implementation parameters enabled by default	14-23
MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with Altera SoC and Xilinx Zynq hardware	14-23
Updates to support package for Texas Instruments C2000 processors	14-23
Support package for Freescale FRDM-K64F board	14-23
Support for TI's C5000 DSPs will be removed	14-24
Support for TI's C6000 DSPs will be removed	14-24
Change in base product for ARM Cortex-Based VEX Microcontroller support package	14-24
Performance	14-25
Data Buffer Reuse: Use same variable for multiple signals in a path by using the same Reusable storage class specification	14-25

Reuse input, output, and state of Delay block	14-25
Initialization code occurs once after start code in model_initialize function	14-25
Reset function improves initialization code optimization	14-28
Removal of unnecessary rtmIsFirstInitCond flag	14-30
Optimized code for models containing logical operator blocks	14-32
Improved code for conditional expressions involving Boolean expressions	14-33
memset Optimization for more scenarios	14-34
Changes to meaning of createCRLEntry wildcard syntax for fixed-point data	14-39
Code replacements involving root-level I/O variables and data alignment	14-40
Verification	14-41
SIL/PIL Data Access: Use vector Get/Set custom storage class and C++ parameter access methods	14-41
SIL/PIL support for variant condition propagation	14-41
SIL simulation returns standard output and standard error streams ...	14-41
Linux SIL/PIL support for LDRA Testbed	14-41
Check bug reports for issues and fixes	14-42

R2015aSP1

Bug Fixes

Check bug reports for issues and fixes	15-2
---	------

R2015b

Code Generation from MATLAB Code	16-2
MATLAB Coder Storage Classes: Easily import and export data by using storage classes	16-2
MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with BeagleBone Black hardware	16-3
Code generation assumptions verified during PIL execution	16-3
Control of signed right shifts in generated code	16-3
Detection of multiword operations	16-4
Model Architecture and Design	16-5
MISRA-C 2012: Comply with mandatory and required rules	16-5
AUTOSAR 4.1.3 and 4.2: Import and export ARXML and generate code for latest AUTOSAR standard	16-5
AUTOSAR sender-receiver modeling	16-6

AUTOSAR client-server modeling	16-8
AUTOSAR nonvolatile data communication modeling	16-9
AUTOSAR component behavior modeling	16-11
AUTOSAR COM_AXIS lookup table modeling	16-12
Embedded Coder model templates	16-12
Removal of uncalled Disable functions from generated code	16-13
Enhancement to option for generating preprocessor conditionals	16-13
Data, Function, and File Definition	16-15
Tokenized function names for custom storage class GetSet	16-15
Code Generation	16-16
Embedded Coder Quick Start: Quickly configure model to generate reusable and efficient code	16-16
Internationalization: Generate and review code containing mixed languages for different locales	16-16
MISRA C:2012 code generation objective	16-17
AUTOSAR arxml round-trip	16-17
Toolchain controls for AUTOSAR code generation	16-19
AUTOSAR RTE file generation enhanced for SIL and PIL	16-19
Lookup table blocks with new even spacing specification generate AUTOSAR compatible IFX library routines	16-20
Control use of signed shifts in generated code	16-20
Code generation report with operator traceability	16-21
Deployment	16-22
Hardware Implementation Selection: Quickly generate code for popular embedded processors	16-22
Code Replacement Tool uses simplified specification	16-23
Code replacement support for new lookup table breakpoint specification	16-24
Support for Analog Devices VisualDSP++ will be removed	16-24
Performance	16-25
RAM/ROM Optimization Improvements: Generate more efficient code using reusable storage class and converting data copies to pointer assignments	16-25
Live Execution Profiling: View PIL profile results during run-time	16-26
Enhanced support for buffer reuse at the root-level input and output ports	16-26
More efficient code for small subsystems	16-29
More efficient code for Simulink.Bus objects	16-30
Enhanced local variable reuse	16-32
Enhanced consolidation of for loops	16-33
Verification	16-35
Faster SIL and PIL Verification Workflow	16-35
Code generation assumptions verified during PIL simulation	16-35
SIL and PIL support for C++ class root-level I/O access methods	16-35
Removal of Generate code only parameter restriction	16-36
Removal of scheduling limitations that caused algebraic loops	16-36

Check bug reports for issues and fixes	16-2
---	-------------

R2015a

Code Generation from MATLAB Code	17-2
Indent style and size control for generated C/C++ code	17-2
Improved MISRA-C compliance for bitwise operations on signed integers	17-2
Improved MISRA-C type cast compliance	17-3
Model Architecture and Design	17-5
AUTOSAR improvements including multi-runnable modeling and code efficiency	17-5
Combined input/output arguments with function prototype control	17-5
Improved MISRA-C compliance for bitwise operations on signed integers	17-5
AUTOSAR multi-runnable modeling using Simulink rate-based multitasking	17-6
Enhanced modeling with AUTOSAR system constants	17-6
AUTOSAR CompuMethod enhancements	17-7
Preprocessor conditionals for single variant choice	17-7
Data, Function, and File Definition	17-8
Control of Boolean and data type limit identifiers in generated code	17-8
Names of built-in storage classes reserved	17-8
Code Generation	17-10
Simplified Code Replacement Library specification plus more replacements involving integer operations	17-10
Improved readability for shared header file 'rtwtypes.h'	17-11
New and enhanced Model Advisor checks for MISRA-C compliance	17-12
Improved traceability for AUTOSAR RTE implicit read	17-12
Configurable aliveTimeout value for AUTOSAR ports	17-13
AUTOSAR calibration parameter export for COM_AXIS lookup tables ..	17-13
Fixed-point scaling information in Code Interface Report	17-13
Unsigned integer minimum data limit identifiers	17-14
Default iteration variable data type	17-14
Deployment	17-16
Code Replacement Viewer enhanced	17-16
Model configuration parameter considered for division operator code replacements	17-16
Lookup table algorithm parameter specification enhancements	17-16
Header file for Basic Linear Algebra Subroutine (BLAS) multiplication function code replacement example changed	17-16
Code replacement detection of overflow and rounding mode equivalence	17-17

Feature being removed in a future release	17-17
Performance	17-18
More efficient code involving model references, unit delays, and global data references	17-18
Conditional compilation of Data Store Memory block memory definition and declaration	17-23
Ternary Boolean expressions transformed into assignment statements	17-24
Verification	17-25
SIL/PIL for protected models and SIL source code debugging using Microsoft Visual Studio Express	17-25
Model block SIL/PIL parameter renamed	17-26
ERT S-Function block no longer supported for AUTOSAR	17-26
SIL/PIL support for replacing boolean data type with int8	17-26
SIL/PIL support for generated access methods for C++ model class root-level I/O signals	17-26
Check bug reports for issues and fixes	17-27

R2014b

Code Generation from MATLAB Code	18-2
Processor-in-the-loop (PIL) verification and execution profiling for MATLAB code	18-2
Software-in-the-loop verification improvements for MATLAB Coder	18-2
Additional options for custom banners and comments in C and C++ code generated from MATLAB code	18-3
Highlighting of potential data type issues in code generation reports	18-3
Model Architecture and Design	18-7
AUTOSAR targeting updates including 4.1 ARXML, client/server with Simulink Functions, multi-instance components, and IFL/IFX libraries	18-7
AUTOSAR client and server modeling	18-7
Global From and Goto blocks for AUTOSAR modeling	18-8
AUTOSAR IRV branch from output signal allowed outside runnable	18-8
Data, Function, and File Definition	18-9
Constant sample time limitation for AUTOSAR models	18-9
Iteration variable in For Iterator block uses signal name	18-9
Data type replacement specification can be used across models	18-9
Definition file for grouped custom storage classes	18-9
Type definition location for custom storage classes	18-9
GetFunction and SetFunction included in checks for identifier clash	18-9

Code Generation	18-10
Enhanced reporting of eliminated blocks	18-10
Improved MISRA-C type cast compliance	18-10
Support Package for AUTOSAR Standard	18-10
AUTOSAR help navigation enhancements	18-11
Support for AUTOSAR Release 4.1	18-11
Multi-instance AUTOSAR atomic software components	18-12
AUTOSAR arxml import and export	18-12
AUTOSAR addPackageableElement replaces add*Interface functions ..	18-16
Code generation report with enhanced navigation and integrated access to code metrics data	18-16
Updated license requirements for viewing code generation report	18-17
Option for doxygen style comments in generated code	18-17
Dynamic memory allocation parameters renamed	18-18
Template makefile compatibility with execution time profiling	18-18
Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation	18-18
Deployment	18-20
Embedded Coder support packages for AUTOSAR, TI Concerto, and Freescale FRDM-KL25Z	18-20
Relational operator replacement	18-20
Code replacement involving vector and matrix data	18-20
Algorithm specification for addition and subtraction operator replacement	18-21
Improved code replacement with output type cast absorption	18-21
Lookup table function code replacement extended to 30 dimensions ...	18-22
Rounding mode support for lookup table function replacement	18-22
Algorithm parameter value sets in code replacement table entries	18-22
coder.replace support for functions specified with varargin input variable	18-23
Documentation installation with hardware support package	18-23
Support package for Altera SoC platform	18-23
Support package for BeagleBone Black hardware	18-23
Support for Eclipse IDE has been removed	18-23
Support for Green Hills MULTI IDE has been removed	18-24
Support for Texas Instruments C5000 DSPs will be removed	18-24
Performance	18-25
Reduced RAM and faster execution for modeling patterns including select- assign-iterate blocks, subsystem interfaces, and model references ..	18-25
Global variable localization optimizations	18-30
Verification	18-32
Top-model code testing with Model block SIL and PIL	18-32
SIL/PIL support for Simulink Function and Function Caller blocks	18-32
SIL debugging support for Linux	18-32
PIL support for test hardware approach	18-33
SIL/PIL support for model initialization dynamic memory allocation ...	18-33
Check bug reports for issues and fixes	18-34

Code Generation from MATLAB Code	19-2
Template to customize code generation output for MATLAB Coder	19-2
In-place function replacement with <code>coder.replace</code> in MATLAB	19-2
Single-line (<code>//</code>) comment style available for generated code	19-2
Software-in-the-loop verification for MATLAB Coder	19-3
Change of default value for <code>MATLABFcnDesc</code>	19-4
Model Architecture and Design	19-5
Capability to merge AUTOSAR authoring tool changes into Simulink models as part of round-trip iterations	19-5
AUTOSAR 4.0 static and constant memory, AUTOSAR-typed per-instance memory, and <code>VariationPointProxy</code>	19-7
Specify AUTOSAR runnable symbol name distinct from short-name	19-7
Improved AUTOSAR arxml support for measurement and calibration ...	19-8
AUTOSAR data dictionary support	19-8
Configure AUTOSAR Interface button removed from AUTOSAR Code Generation Options	19-9
Subsystem methods of <code>AUTOSAR arxml.importer</code> class removed	19-9
Data, Function, and File Definition	19-10
Custom storage class and optimized class declarations for C++ class code generation	19-10
Constant sample time limitations for root-level Outport blocks	19-10
Example model <code>rtwdemo_cppencap</code> renamed to <code>rtwdemo_cppclass</code>	19-11
Unit Delay block optimization	19-11
Code Generation	19-12
In-place function replacement with <code>coder.replace</code> in MATLAB and lookup table code replacement for Simulink	19-12
Global variable usage available in the static code metrics report	19-12
Single-line (<code>//</code>) comment style available for generated code	19-12
Code indentation support for namespace declarations in generated code	19-13
AUTOSAR C code generation enhancements	19-13
Static main program module for C++ class code generation	19-14
Error message for data type replacement and classic call interface conflict	19-14
Deployment	19-15
ARM Cortex-A optimized code generation using Ne10 library	19-15
Lookup table code replacement for Simulink	19-15
Replacement of functions that take vector and matrix arguments	19-15
Logical data type support for arguments of replaced functions	19-16
Code replacement data alignment for complex types	19-16
Intel IPP (ANSI) and Intel IPP (ISO) code replacement libraries are combined	19-16
Support for Eclipse IDE will be removed	19-16

Support for Green Hills MULTI IDE will be removed	19-17
Support package for ARM Cortex-A processors	19-17
Support package for Texas Instruments C6000 processors	19-17
Updates to support package for Texas Instruments C2000 processors ..	19-18
Updates to support package for Xilinx Zynq-7000 platform	19-18
Updates to support package for STMicroelectronics STM32F4 Discovery board	19-18
Wind River Tornado (VxWorks 5.x) example main program option to be removed in future release	19-19
Performance	19-20
Additional options for reuse of global variables	19-20
Enhanced global variable optimization options	19-20
for loops used to initialize arrays to zero	19-20
Verification	19-21
Software-in-the-loop simulation for physical models	19-21
SIL verification for subsystem code generation	19-21
SIL and PIL support for fixed-point data type override	19-23
SIL and PIL support for Invoke AUTOSAR Server Operation block	19-23
SIL and PIL support for structure parameters with storage class SimulinkGlobal	19-23
Model block SIL and PIL with export-function and asynchronous function- call models	19-23
Model block SIL and PIL with disabled inline parameters	19-24
SIL and PIL block improvements	19-24
Check bug reports for issues and fixes	19-25

R2013b

Code Generation from MATLAB Code	20-2
Software-in-the-loop verification for MATLAB Coder	20-2
Custom generated identifiers for emxArray utility functions	20-2
Model Architecture and Design	20-3
Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior	20-3
Reorganization of Model Advisor Embedded Coder checks	20-5
Model Advisor fixed-point checks with additional coverage and optimization awareness	20-5
Protected model Web view	20-5
RTW.AutosarInterface class to be removed in a future release	20-5
Subsystem methods of arxml.importer class to be removed in a future release	20-6
Data, Function, and File Definition	20-7

Simplified global types file <code>rtwtypes.h</code> with invariant content	20-7
C++ encapsulation support for name space control and template-based file customization	20-7
Shared utility naming control	20-8
Expanded support for identifier names	20-8
Terminate function setting honored for subsystems and referenced models	20-8
Code Generation	20-10
Support for AUTOSAR release 4.0.3 XML and generated code	20-10
Indent style and size control for code generation	20-10
Subsystem functions return value in generated code	20-10
Model reference step function void input and output arguments	20-10
Deployment	20-11
ARM Cortex-M optimized code with STM32F4-Discovery board example	20-11
Wind River VxWorks 6.9 support	20-12
Support package for Texas Instruments C2000 processors	20-12
Coder Target pane in Configuration Parameters dialog box	20-13
ZedBoard hardware support	20-14
Simplified multi-instance code interface and dynamic memory allocation for ERT targets	20-14
Addition and Subtraction Operator Code Replacement Assumes Cast-Before-Operation Behavior	20-15
Performance	20-17
Reusable custom storage class to reduce root I/O memory	20-17
Subsystem functions reused independently of output connection	20-17
Verification	20-18
SIL and PIL support fixed-point data types wider than 32 bits	20-18
SIL and PIL protected model support	20-18
Code execution profiling improvements	20-18
Check bug reports for issues and fixes	20-20

R2013a

Code Generation from MATLAB Code	21-2
Improved code replacement traceability for MATLAB code generation	21-2
Static code metrics report for MATLAB Coder	21-2
Model Architecture and Design	21-4
AUTOSAR user interface and round trip ARXML file import and export improvements	21-4

Code generation for variable-size scalar signals	21-6
Data, Function, and File Definition	21-7
Shortened system-generated identifier names	21-7
Improved data initialization with custom storage classes	21-7
Default specification for global types	21-7
Subsystem block parameter Function packaging option renamed	21-7
Code Generation	21-8
Model Advisor checks for code generation	21-8
Deployment	21-9
Concurrent execution API to target embedded multicore platforms	21-9
Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box	21-9
Downloadable support and blocks for Analog Devices DSPs	21-10
Texas Instruments C2000 Clocking Options	21-11
Support for Texas Instruments C2802x and Texas Instruments C2803x variants	21-12
Downloadable support and blocks for Xilinx Zynq-7000 platform	21-12
Support ending for Eclipse IDE in a future release	21-13
Support ending for remoteBuild method in a future release	21-13
Performance	21-14
Optimized function arguments for nonreusable subsystems	21-14
Reduced data copies for tunable parameter expressions	21-14
Removal of unused global variables	21-14
Verification	21-15
Debugging during SIL simulations	21-15
Simulation of multiple SIL Model blocks in a top model	21-15
API for testing rtiostream communications	21-15
SIL and PIL support for targets with multicore processors	21-16
Additional code annotation for justifying Polyspace checks	21-16
Code execution profiling improvements	21-16
Code-to-model traceability links for reusable subsystems in libraries	21-17
Check bug reports for issues and fixes	21-19

R2012b

Cyclomatic complexity measurement in static code metrics report	22-2
Custom code substitution for MATLAB functions using code replacement libraries	22-2

SIL and PIL support for signal logging, encapsulated C++, and AUTOSAR calibration parameters	22-2
Signal logging for SIL and PIL simulations	22-2
Use SIL and PIL simulations to verify encapsulated C++ code	22-3
Improved SIL and PIL verification for AUTOSAR-compliant code	22-3
AUTOSAR 4.0 nonscalar data support	22-3
Code annotation for justifying Polyspace checks	22-3
Texas Instruments Code Composer Studio IDE 5.1 support	22-4
External mode support for ERT targets with static main	22-4
Downloadable support for Green Hills MULTI	22-4
Support for Texas Instruments C2806x processors	22-5
Performance enhancement of Simulink data objects	22-6
AUTOSAR software component import and export enhancements	22-7
Import validation	22-7
Faster import and export of arxml files	22-7
Explicit access mode for AUTOSAR Sender and Receiver ports	22-7
Import port-based calibration parameters	22-7
Highlight virtual blocks in model Web view of code generation report	22-7
Code Execution Profiling Improvements	22-7
Updated Code Execution Profiling API	22-7
Code Execution Profiling Supports Single Object Output	22-10
Incremental Compilation with Changes in Code Coverage Settings ...	22-10
Check bug reports for issues and fixes	22-11

R2012a

AUTOSAR Enhancements	23-2
AUTOSAR Release 4.0	23-2
Support for Schema 2.0 Removed	23-2
Code Efficiency Enhancements	23-2
For Each Subsystem Loop Bound Passed by Value	23-2
Fully Inlined S-functions from Legacy Code Tool	23-2
Element-Wise Operations as Inputs to Intrinsic Functions	23-3
Enhancements to Custom Storage Classes in Simulink and mpt Packages	23-3

Code Generation Report Includes Simulink Web View	23-4
LDRA Testbed Code Coverage Annotations in Code Generation Report	23-4
Generated Identifiers Enhancements	23-4
Simplified Identifiers for Model Reference Code	23-4
Consistent Identifiers for Comparing Generated Code	23-5
Code Replacement Enhancements	23-5
Target Function Libraries Renamed to Code Replacement Libraries	23-5
Enhanced Code Replacement Traceability	23-5
Code Replacement Support for Simulink Matrix Division and Inversion Operators	23-6
Code Replacement Support for MATLAB Coder fix, hypot, round, and sign Functions	23-6
Integer Functions Now Return Real-World Values	23-6
SIL and PIL Enhancements	23-7
SIL and PIL Test Harness Files in Code Generation Report	23-7
PIL Support for Code Coverage with LDRA Testbed	23-8
Seamless Switching Between SIL and PIL for Top-Model and Model Block	23-8
Enhanced Hardware Implementation Support	23-8
Top-Model Output Limitations Removed	23-9
Model Block SIL/PIL Support for Absolute Time	23-9
Changes for ERT and ERT-Based Targets	23-9
Changes for Embedded IDEs and Embedded Targets	23-10
Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE	23-10
Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors	23-10
Support Removed for Freescale MPC5xx	23-11
Limitation: Parallel Builds Not Supported for Embedded Targets	23-11
New and Enhanced Demos	23-11
Check bug reports for issues and fixes	23-13

R2011b

Static Code Metrics in Code Generation Report	24-2
AUTOSAR Enhancements	24-2
Import and Export of AUTOSAR Sensor/Actuator Components	24-2
Improved Simulink Library Support for Multiple Runnables	24-2
AUTOSAR Schema Version 3.2	24-2
Export AUTOSAR XML as Single File	24-2

SIL and PIL Enhancements	24-2
Code Execution Profiling of Functions in Subsystems and Model Blocks	24-2
Code Coverage with LDRA Testbed	24-3
BitField and GetSet Custom Storage Classes	24-3
Model Blocks with Variable-Size Signals	24-3
Verification of Generated C++ Code	24-3
Generate Multitasking Code for Concurrent Execution on Multicore Processors	24-4
Changes for Embedded IDEs and Embedded Targets	24-4
64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0	24-4
Support Added for Wind River VxWorks 6.8	24-4
Support Added for Serial Communications Interface with Processor-in-the- loop (PIL) for Texas Instruments™ C28035 and C28335	24-5
New Target Function Library for Intel IPP/SSE (GNU)	24-5
Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors	24-5
Support Removed for Altium TASKING	24-5
Support Removed for Infineon C166	24-5
Support Ending for Green Hills MULTI in a Future Release	24-6
Support Ending for Freescale MPC5xx in a Future Release	24-6
Saturation Control of Stateflow Data	24-6
Custom Storage Class Properties for Managing Data Ownership and Definition	24-6
Export Data Declarations to Shared Header File for Code Generation with Model Reference	24-7
Target Function Library Code Replacement Enhancements	24-7
Code Replacement Tool for Creating and Managing TFL Tables	24-7
Ability to Align Data Objects to TFL-Specified Boundaries to Boost Code Performance	24-8
Support for Replacing Element-wise Matrix Multiply	24-8
Code Generation Enhancements	24-8
Redundant Condition Checks	24-8
Loop Fusion	24-9
Invariant Condition Check Lifting	24-9
Parameter Pooling for Stateflow and Interpreted MATLAB Function Blocks	24-9
Readability Improvement for Reusable Subsystem Input and Output	24-9
Enhanced Code Generation Optimization Using Minimum and Maximum Values	24-9
New Model Advisor Check for Code Efficiency of Logic Blocks	24-10
Control of Default Case Generation for Switch Statements in Generated Code for Stateflow Charts	24-10

Improvement to Build Process for Conflicting Identifiers	24-11
Update to Code Generation Verification Class cgv.Config	24-11
License Names Not Yet Updated for Coder Product Restructuring ...	24-11
New and Enhanced Demos	24-12
Check bug reports for issues and fixes	24-13

R2011a

Coder Product Restructuring	25-2
Product Restructuring Overview	25-2
Resources for Upgrading from Real-Time Workshop Embedded Coder ..	25-2
Migration of Embedded MATLAB Coder Features to MATLAB Coder	25-3
Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder	25-3
Interface Changes Related to Product Restructuring	25-4
Simulink Graphical User Interface Changes	25-4
Data Management Enhancements and Changes	25-4
Memory Section Enhancements	25-5
No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects	25-5
Parts of Data Class Infrastructure Not Available	25-5
No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer	25-6
Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release	25-6
AUTOSAR Enhancements	25-7
Calibration Parameters	25-7
Multiple Runnables from Virtual Subsystems	25-7
Support for Code Descriptor Elements	25-7
SIL and PIL Enhancements	25-8
Code Execution Profiling	25-8
PIL Block Parameter Tuning	25-8
Top-Model SIL/PIL and PIL Block Parameter Initialization	25-8
Model Block Parameter Tuning and Model Initialization	25-8
Code Generation Enhancements	25-9
Improved Code for Data Store Memory In-place Assignment	25-9
Improvements to Target Function Library Replacements	25-9
Improved Loop Fusion	25-9
Improved Array Indexing	25-9
Improvement on Matrix Parameter Pooling	25-9
Readability Improvements Involving Data References	25-9

Code Generation Verification (CGV) API Updates	25-10
Support for Adding Multiple Callback Functions	25-10
New Functionality Added to the cgv.CGV Class	25-10
MISRA-C Code Generation Objective	25-12
New Model Advisor Check for Code Efficiency of Lookup Table Blocks	25-12
Enhanced Code Generation Optimization	25-13
Target Function Library Replacement Based on Computation Method for Reciprocal Sqrt, Sine, and Cosine	25-13
Target Function Library Support for abs, min, max, and sign functions	25-13
C++ Encapsulation Allowed for Referenced Models in For Each Subsystems	25-13
Improved Code Generation for Portable Word Sizes	25-14
Improved Comments in the Generated Code	25-14
Replacement Data Types and Simulation Mode for Referenced Models	25-14
Changes for Embedded IDEs and Embedded Targets	25-14
Feature Support for Embedded IDEs and Embedded Targets	25-15
Execution Profiling during PIL Simulation	25-15
Location of Blocks for Embedded Targets	25-15
Location of Demos for Embedded IDEs and Embedded Targets	25-16
Multicore Deployment with Rate-Based Multithreading	25-17
Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)	25-17
Changes to Frame-Based Processing	25-17
New Support for Analog Devices Blackfin BF50x and BF51x Processors	25-18
Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors	25-19
Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI	25-19
Support for Texas Instruments Delfino C2834x Processors	25-19
Ending Support for Altium TASKING in a Future Release	25-20
Ending Support for Freescale MPC5xx in a Future Release	25-20
Ending Support for Infineon C166 in a Future Release	25-20
Removed Methods and Arguments	25-20
Changes to ver Function Product Arguments	25-20
New and Enhanced Demos	25-20
Check bug reports for issues and fixes	25-22

R2022b

Version: 7.9

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Removal of initialized but unused class properties in generated C/C++ code

Starting in R2022b, unused class properties or structure fields are removed along with their initialization statement from generated C/C++ code. Prior to R2022b, initialization of unused class properties or structure fields were preserved.

This enhancement reduces code complexity, reduces memory usage at run time, and improves code readability.

The table compares the code generated in R2022b with the code generated in R2022a.

MATLAB® Code	R2022b Generated Code	R2022a Generated Code
<pre>function out = myStruct(n) %# codegen s.a = [n n n]; % initialized and unused field s.b = n+2; s.c = n; % initialized and unused field out = myAdd([s s]); end function out = myAdd(s) coder.inline('never'); out = s(1).b + s(2).b; end</pre>	<pre>typedef struct { double b; } struct_T; /* * Arguments : double n * Return Type : double */ double myStruct(double n) { struct_T b_s[2]; struct_T s; s.b = n + 2.0; b_s[0] = s; b_s[1] = s; return myAdd(b_s); }</pre>	<pre>typedef struct { double a[3]; double b; double c; } struct_T; /* * Arguments : double n * Return Type : double */ double myStruct(double n) { struct_T b_s[2]; struct_T s; s.a[0] = n; s.a[1] = n; s.a[2] = n; s.b = n + 2.0; s.c = n; b_s[0] = s; b_s[1] = s; return myAdd(b_s); }</pre>

For more information, see “Removal of Unused Class Properties in Generated C/C++ Code”.

Reduction of violations for MISRA C:2012 and AUTOSAR C++14 rules in generated code

In R2022b, the generated code has fewer violations of several rules in the required categories of MISRA® C: 2012 and AUTOSAR C++14 coding standards. Some of these rules are:

- Dead code: MISRA C:2012 Rule 2.2, AUTOSAR C++14 Rule M0-1-9
- Lexical conventions: AUTOSAR C++14 Rule M2-10-1, AUTOSAR C++14 Rule A2-10-6, AUTOSAR C++14 Rule A2-3-1
- Identifiers: MISRA C:2012 Rule 5.6

- Compilation directive: MISRA C:2012 Rule 4.1, MISRA C:2012 Dir 4.12
- Side effects and expressions: MISRA C:2012 Rule 13.2, AUTOSAR C++14 Rule A5-0-1, AUTOSAR C++14 Rule M5-0-8
- Standard libraries: MISRA C:2012 Rule 21.3
- Other restrictions: MISRA C:2012 Dir 2.1, AUTOSAR C++14 Rule M0-1-3, AUTOSAR C++14 Rule M17-0-2, AUTOSAR C++14 Rule A12-0-1, AUTOSAR C++14 Rule A0-1-3, AUTOSAR C++14 Rule A18-0-1

For more information on how to generate code that has improved MISRA and AUTOSAR compliance, see “Generate C/C++ Code with Improved MISRA Compliance”.

Model Architecture and Design

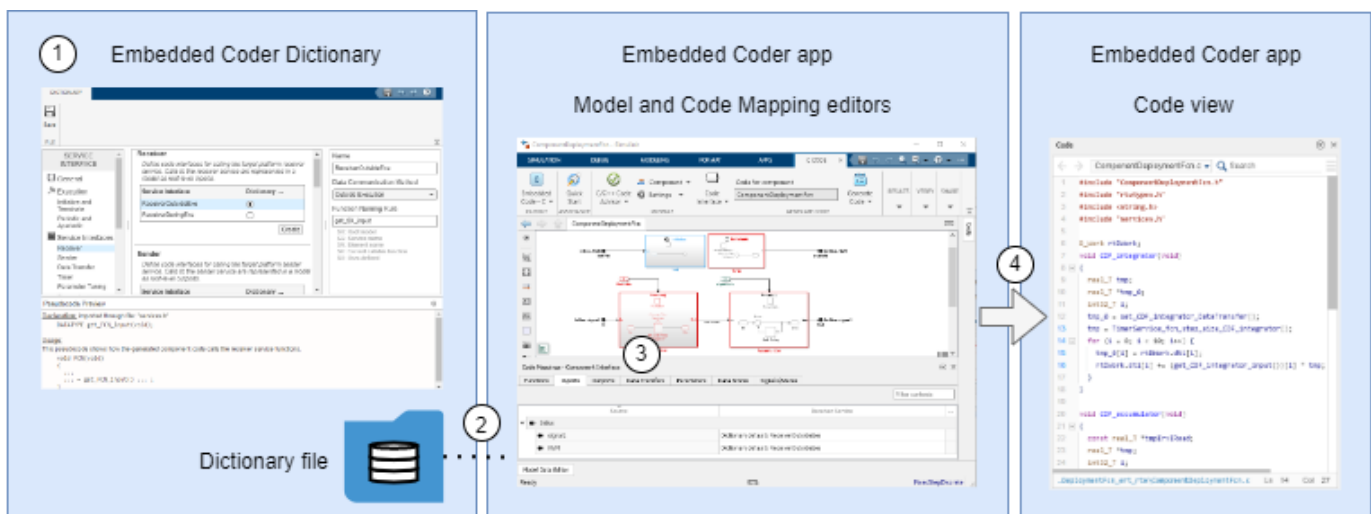
Deploy models as components that include comprehensive service interface support

Starting in R2022b, Embedded Coder provides a set of features that enhance how you model, configure, generate, verify, and integrate component model code intended to interact with service implementations of a target platform. You can set up a service interface configuration that includes comprehensive service support for a target platform that modelers can share. Service interfaces:

- Support deployment of periodic and aperiodic rates.
- Enable customization of generated interface code for single- and multicore deployment with built-in safeguards for maintaining data coherence.
- Support code customization of data transfers between functions outside of (before and after) function execution.
- Provide support for accessing time values in aperiodic tasks.

You generate code that includes comprehensive platform service support by completing these steps (highlighted in the following figure):

- 1 Create a shared Embedded Code Dictionary that defines a service interface configuration, including default interfaces, for your target platform.
- 2 Link a model to the Embedded Coder Dictionary.
- 3 If you want to override default mappings that are configured in your dictionary, map model elements to service interfaces.
- 4 Generate code that complies with the service interface configuration.



The complete set of features enables you to:

- Apply a new set of modeling guidelines for interfacing generated code with target platform software. Examples of target platform software include a function scheduler and services that send and receive data and provide access to the target environment clock tick. See “Modeling

guidelines and Model Advisor checks for component deployment using a service interface configuration” on page 1-9.

- Set up a shared Embedded Coder Dictionary that includes comprehensive service interface configurations, including behavior semantics, for generating component code intended to interact with services provided by specific target platforms. See:
 - “Control interface of generated code using data and service interface configurations in Embedded Coder Dictionary” on page 1-6
 - “Component service interface support for callable entry-point functions” on page 1-7
 - “Component service interface support for target platform data receiver and data sender services” on page 1-7
 - “Component service interface support for target platform data transfer service” on page 1-7
 - “Component service interface support for target platform timer service” on page 1-8
 - “Component service interface support for target platform parameter tuning and measurement services” on page 1-8
 - “New \$X naming rule token” on page 1-16
- Link a component model to a shared Embedded Coder Dictionary that includes service interface configurations. See “Control interface of generated code using data and service interface configurations in Embedded Coder Dictionary” on page 1-6.
- Configure a model for component or subcomponent deployment. See “Select code interface configuration using new configuration parameter” on page 1-13.
- Map elements of a component model to service interfaces defined in the shared dictionary linked to the model. See “Map model elements to service interfaces” on page 1-10.
- Use new Model Advisor checks to confirm that a model that is configured to use service interfaces complies with modeling guidelines and is ready for code generation. See “Modeling guidelines and Model Advisor checks for component deployment using a service interface configuration” on page 1-9.
- Generate and review the file structure and naming of code generation output that supports service interfaces for component deployment. See “Files and folders for target platform services” on page 1-15.
- View a version of the Code Interface Report that is enhanced to show details about component callable entry-point functions and service code interfaces. See “Code interface report for service interfaces” on page 1-16.
- Use software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations to test your generated service interface code on your development computer and a target processor or simulator, respectively. See “SIL/PIL Manager Verification Workflow”.
- Ease component code integration by using the code descriptor programming interface to get metadata about the code interface generated for a model. See “Retrieve metadata about service interface by using code descriptor programming interface” on page 1-20.

For examples, see “Deploy Export-Function Component Configured for C Service Interface Code Generation” and “Deploy Rate-Based Component Configured for C Service Interface Code Generation”.

For background and high-level workflow information, see “Embedded Coder Fundamentals”.

For information about constraints and current limitations, see “Service Interface Constraints and Limitations”.

Control interface of generated code using data and service interface configurations in Embedded Coder Dictionary

Starting in R2022b, you can control the interface of your generated code by creating a code interface configuration in an Embedded Coder Dictionary and mapping your model elements to the definitions in the code interface configuration. The Embedded Coder Dictionary contains one of these code interface configurations:

- **Service interface configuration** — The configuration contains service interfaces, storage classes, function customization templates, and memory sections. The new service interface configuration enables you to define comprehensive service interfaces to generate code that interacts with services provided by specific target platforms. For more information, see “Deploy models as components that include comprehensive service interface support” on page 1-4.
- **Data interface configuration** — The configuration contains storage classes, function customization templates, and memory sections. If you created an Embedded Coder Dictionary in an earlier release, when you open the dictionary in R2022b, the dictionary contains a data interface configuration with the existing code interface definitions.

The screenshot shows the Embedded Coder Dictionary configuration window. The title bar reads "DICTIONARY". On the left, there is a "FILE" menu with a "Save" option. The main area is titled "SERVICE INTERFACE" and contains a tree view on the left with the following items: General (selected), Execution (with sub-items: Initialize and Terminate, Periodic and Aperiodic), Service Interfaces (with sub-items: Receiver, Sender, Data Transfer, Timer, Parameter Tuning, Parameter Argument Tuning, Measurement), Internal Functions (with sub-items: Subcomponent Functions, Shared Utility), and Memory (with sub-item: Storage Class). The main pane is titled "General" and contains the following fields and sections:

- Location:** C:\work\InterfaceCoderDictionary.sldd
- Header File Name:** services.h
- Configure Service Interface:** Define code interface configuration for interacting with target platform services. A configuration can consist of interfaces for generated callable entry-point functions and interfaces for calling services, such as communication and timer services.
- Execution:** Create function customization templates that configure how model functions appear as callable entry points in generated code. The entry points are called by a target environment function scheduler.
- Service Interface:** Create service interfaces for model elements to call platform services for receiving data, sending data, transferring data between callable functions, accessing time, tuning parameters, and measuring signal data.
- Internal Functions:** Create function customization templates that specify how internal model functions appear when referenced by another model.
- Memory:** Create storage classes and memory sections for configuring how service interfaces that are configured to use the direct access data communication method access data.

At the bottom, there is a "Pseudocode Preview" section with a message: "Pseudocode preview is not applicable. Select a code definition."

Starting in R2022b, when you create an Embedded Coder Dictionary, you specify whether the dictionary contains a service interface configuration or a data interface configuration. For more information, see Embedded Coder Dictionary.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Component service interface support for callable entry-point functions

Starting in R2022b, for component deployment, you can associate a model with a service interface configuration that aligns with callable entry-point function requirements for a specific target platform. The service interface configuration includes function customization templates for:

- Periodic and aperiodic functions used for executing component algorithms
- Initialize and terminate functions used for handling startup and shutdown events

The code generator produces the callable entry-points based on code mappings from functions represented in a model to function customization templates configured in the shared Embedded Coder Dictionary linked to the model. For more information, see “Periodic and Aperiodic Function Interfaces” and “Startup, Reset, and Shutdown Function Interfaces”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Component service interface support for target platform data receiver and data sender services

Starting in R2022b, for component deployment, you can generate code that sends and receives data to and from the target environment by using environment-specific data communication methods. In a model, at the root level, you represent a sender service as an Outport block or a Bus Element Outport block. You represent a receiver service as an Inport block or a Bus Element Inport block. The code generator produces service interfaces based on your specified communication methods in the shared Embedded Coder Dictionary linked to the model. These service interfaces are also based on the mappings between Embedded Coder Dictionary interfaces and model elements specified in the Code Mappings editor.

For more information, see “Data Communication Methods”, “Service Interfaces”, “Create a Service Interface Configuration”, and “Generate Sender and Receiver C Interface Code for Component Deployment”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Component service interface support for target platform data transfer service

Starting in R2022b, for component deployment, you can generate code that supports data transfers between callable functions within a model, including data transfers that occur between functions outside of (before and after) function execution or during function execution. Within a model, you represent a data transfer as a signal line that connects the outport of one callable function to the inport of another callable function. The code generator produces service interfaces based on the

content of the model and the data transfer service interface configuration in the shared Embedded Coder Dictionary linked to the model.

For more information, see “Data Communication Methods”, “Service Interfaces”, “Create a Service Interface Configuration”, and “Generate C Data Transfer Service Interface Code for Component Deployment”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Component service interface support for target platform timer service

Starting in R2022b, for component deployment of aperiodic export-function models, you can generate code that supports access to the function clock tick used by the target environment. Within a model, you represent requests for the clock tick implicitly when you use Discrete Time Integrator and Weighted Sample Time blocks. The code generator assumes that the clock resolution is the fundamental step size of the model and produces a timer service interface based on content of the model and the timer service interface configuration in the shared Embedded Coder Dictionary linked to the model.

For more information, see “Data Communication Methods”, “Service Interfaces”, “Create a Service Interface Configuration”, and “Generate C Timer Service Interface Code for Component Deployment”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Component service interface support for target platform parameter tuning and measurement services

Starting in R2022b, you can generate service interface code that supports:

- Tuning parameters
- Tuning parameter arguments
- Measuring signal, state, and data store data

The code generator produces service interfaces based on data stored in a workspace or dictionary for the model and the parameter tuning, parameter argument tuning, and measurement service interface configurations in the shared Embedded Coder Dictionary linked to the model.

For more information, see “Service Interfaces”, “Create a Service Interface Configuration”, “Generate C Parameter Tuning Service Interface Code for Component Deployment”, and “Generate C Measurement Service Interface Code for Component Deployment”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Modeling guidelines and Model Advisor checks for component deployment using a service interface configuration

Starting in R2022b, MathWorks® provides a set of guidelines that you can use when deploying models as pluggable components whose generated code interacts with service implementations of a target platform. You can use Embedded Coder Model Advisor checks to verify compliance of your model with the modelling guidelines.

For information on how to set up a service interface configuration with Embedded Coder that includes comprehensive service support for a target platform, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

This table identifies the modeling guidelines and their corresponding Model Advisor checks, when applicable.

Modeling Guideline	Model Advisor Check
“cgsl_0401: Modeling styles for component deployment”	“Check modeling style for component deployment”
“cgsl_0402: Signal interfaces for component deployment”	“Check signal interfaces”
“cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment”	A Model Advisor check is not provided for this guideline.
“cgsl_0405: Data receive for component deployment”	A Model Advisor check is not provided for this guideline.
“cgsl_0406: Data send for component deployment”	A Model Advisor check is not provided for this guideline.
“cgsl_0408: Partial data send for component deployment”	A Model Advisor check is not provided for this guideline.
“cgsl_0409: Data transfer for component deployment”	The guideline cannot be verified by using a Model Advisor check.
“cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks”	The guideline cannot be verified by using a Model Advisor check.
“cgsl_0413: Reuse memory between component state and output for component deployment”	The guideline cannot be verified by using a Model Advisor check.
“cgsl_0414: Configure service interface for component model”	“Check configuration for component deployment”

Code Interface Configuration and Integration

Map model elements to service interfaces

In R2022b, code mappings enable you to map elements of a model to service interfaces defined in a shared dictionary linked to the model. You can control code interfaces at different levels of the model hierarchy by configuring the deployment type of the model.

Using the **Code Mappings Editor** or its associated programming interface, customize interfaces in the generated code that interact with target platform services by mapping interface elements in your model to service interfaces defined in a shared dictionary.

Model Element	Service	Example
Inports	Receiver	"Configure Sender and Receiver Service Interfaces for Model Inports and Outports"
Outports	Sender	
Data transfers represented by a signal connecting two function-call subsystems or exported scoped Simulink® functions	Data transfer	"Configure Data Transfer Service Interfaces for Data Transfer Signals"
Export functions	Timer	"Configure Timer Service Interfaces for Aperiodic Export Functions"
Model parameters	Parameter tuning	"Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments"
Model parameter arguments	Parameter argument tuning	
Signals, states, and data stores	Measurement	"Configure Measurement Service Interfaces for Signals, States, and Data Stores"

To configure these services, your model must be linked to a service interface definition. For more information, see "C Service Interfaces" and **Code Mappings editor**.

For more information about deploying component models that are configured with a service code interface, see "Deploy models as components that include comprehensive service interface support" on page 1-4.

Dimension preservation of multidimensional arrays for GetSet and access function storage classes

Previously, you could not generate code that preserved the dimensions of a multidimensional model data element when you set the storage class for that data element to the predefined storage class **GetSet** or to a custom storage class with **Data Access** set to **Function**. In R2022b, when the model configuration parameter **Array layout** is set to **Row-major**, you can preserve the dimensions of a multidimensional array data element when the element uses one of these storage classes.

In the Code Mappings editor, to preserve dimensions for an individual data element that uses a `GetSet` or access function storage class, or a category of such elements, select the **PreserveDimensions** property in the Property Inspector window.

In the Embedded Coder Dictionary, to preserve dimensions for a new custom storage class with **Data Access** set to `Function`, select the **Preserve array dimensions** property in the Property Inspector. This property is available only when the **Access Mode** property for the storage class is set to `Value`.

You can also select the **Preserve array dimensions** property in a data object property dialog box.

For more information, see “Preserve Dimensions of Multidimensional Arrays in Generated Code”.

Support for root level inports and outports as pointer members in C++ generated code

C++ code generation now supports configuring inports and outports at the root level of a model to appear in the generated code as pointer members. Configuring inports or outports as pointers reduces the number of data copies by allowing the generated model class to refer to externally managed memory.

When configuring inports or outports as pointer members, the model must have Model Configuration Parameters set to either generate an example ERT main program (`ert_main.cpp`) or generate code only. Additionally, the member access method for the Inports or Outports must be structure-based.

For more information about configuring C++ interfaces, see “Interactively Configure C++ Interface” and “Programmatically Configure C++ Interface”.

Functionality being removed or changed

Model parameters and parameter arguments returned separately by find function

Behavior change

The `find` function now returns model parameter arguments separately from model parameters.

Starting in R2022b, to return all elements in the model code mappings that are model parameter arguments, enter the following.

```
cm = coder.mapping.api.get('myConfigModel');
modelParamArgs = find(cm, 'ModelParameterArguments');
```

To return all elements in the model code mappings that are model parameters, enter the following.

```
cm = coder.mapping.api.get('myConfigModel');
modelParams = find(cm, 'ModelParameters');
```

In previous releases, specifying `ModelParameters` as the category argument returned both model parameters and model parameter arguments.

Embedded Coder Dictionary refreshes when loading model from earlier release

Behavior change

Package-based code definitions have changed. If your Embedded Coder Dictionary refers to code definitions that you store in a package, the dictionary refreshes when you load a model from an

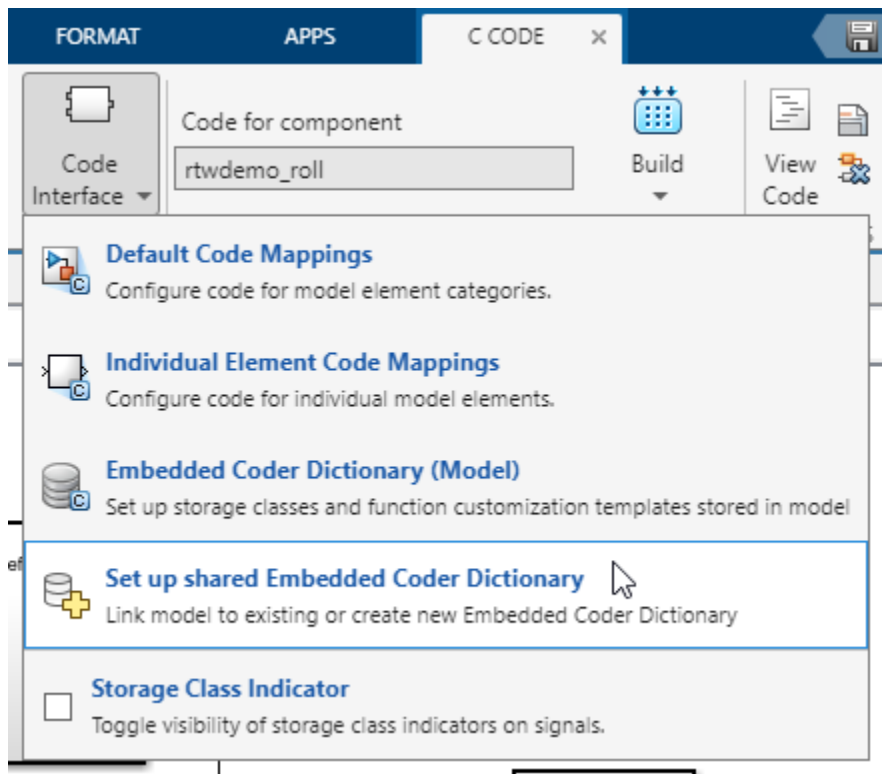
earlier release. To prevent the dictionary refresh, resave the model, or the Simulink data dictionary that contains the Embedded Coder Dictionary, in the current release.

Code Generation

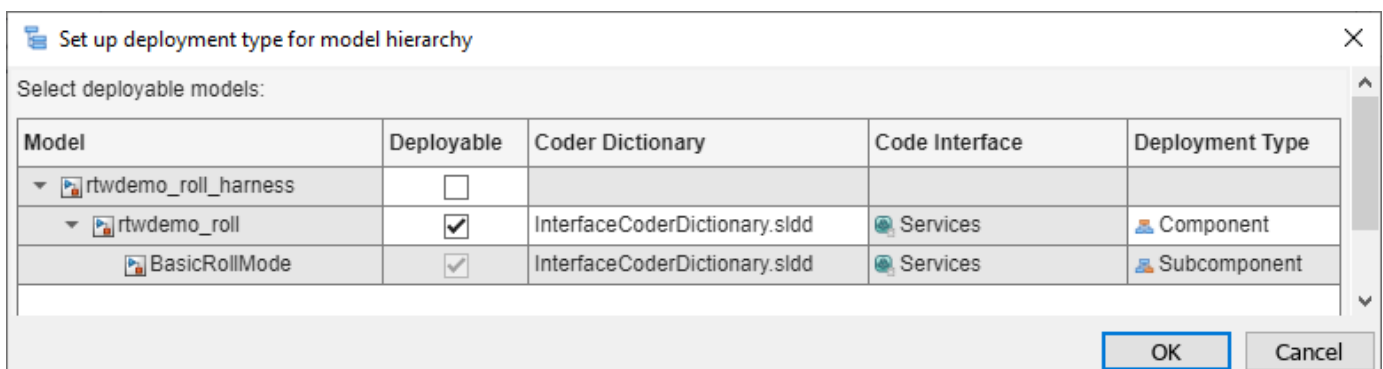
Select code interface configuration using new configuration parameter

In R2022b, you can configure a model to use a code interface configuration in one of these ways:

- In the **Embedded Coder** app, on the **C Code** tab, click **Code Interface > Set up shared Embedded Coder Dictionary**. Use the dialog box to select or create an Embedded Coder Dictionary.



- Specify the Embedded Coder Dictionary for a model hierarchy by using the Set up deployment type for model hierarchy dialog box. The table shows code interface configuration types contained in the dictionary.



- In the Configuration Parameters dialog box, set the configuration parameter **Shared coder dictionary** to the name of an Embedded Coder Dictionary SLDD file.

Shared coder dictionary:	InterfaceCoderDictionary.sldd	Set up...
Description:	A component-based software platform architecture.	

When you select an Embedded Coder Dictionary, the code interface configuration type that the dictionary contains controls the deployment types that are available to configure your model.

- For a data interface configuration, you can select the automatic or subcomponent deployment type.
- For a service interface configuration, you can select the component or subcomponent deployment type.

For more information, see “Select Code Generation Output for Target Platform Deployment” and “Configure C Code Deployment Types for Model Hierarchy”. For more information about deployment component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Generate an example main program parameter not available for models configured with a service interface configuration

When deploying a component, the goal is to produce algorithm code that can be integrated with a main program and scheduler of choice. From a task execution perspective, the generated code is portable across target environments. Given this goal, there is no need to generate an example main program. As such, starting in R2022b, for component models that you configure with a service code interface, you cannot set the model configuration parameter **Generate an example main program** (`GenerateSampleERTMain`).

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Generated C++11 example main program simplified

Starting in R2022b, for models configured with the following model configuration parameter settings, the code generator produces a simplified `ert_main.cpp` file that aligns with the concurrency and multithreading capabilities of the C++11 (ISO) standard library.

- **Allow tasks to execute concurrently on target** is selected.
- **MAT-file logging** is cleared.
- **System target file** is set to an ERT-based system target file.
- **Language** is set to C++.
- **Language standard** is set to C++11 (ISO).
- **Code interface packaging** is set to C++ class.
- **Generate an example main program** (`GenerateSampleERTMain`) is selected.

Prior to R2022b, generated example main program `ert_main.cpp` included a wrapper function, which served as a dispatcher. For example:

```
// Model wrapper function
void rtwdemo_cppclass_step(multi_rate & rtwdemo_cppclass_Obj, int_T tid)
{
    switch (tid) {
        case 0 :
            rtwdemo_cppclass_Obj.EngineEntrypoint();
            break;

        case 1 :
            rtwdemo_cppclass_Obj.EngineEntrypoint1();
            break;

        case 2 :
            rtwdemo_cppclass_Obj.EngineEntrypoint2();
            break;

        default :
            // do nothing
            break;
    }
}
```

The wrapper function used the switch statement to select the `model_stepN` function to call during run time. Starting in R2022b, the code generator improves performance of generated main program by eliminating the wrapper function and calling each entry-point function directly.

For more information, see “Model Multicore Concurrent Tasking Application”, Generate an example main program, and “Deploy Applications to Target Hardware”.

Include requirement comments in the generated code

When you generate C/C++ code from MATLAB code containing requirement links (Requirements Toolbox™), you can include comments in the generated code that contain information about the requirements and the linked MATLAB code ranges. When you view the generated code from a code generation report, the comments are hyperlinks that you can use to navigate to the requirement or the linked MATLAB code range. For more information, see “Requirements Traceability for Code Generated from MATLAB Code” (Requirements Toolbox).

Files and folders for target platform services

When you generate code for a component model that uses a service code interface configuration, the code generator creates these subfolders:

- `codeGenerationFolder/modelBuildFolder/services` — Contains `services.h`, the header file that specifies function prototypes for target platform services.
- `codeGenerationFolder/modelBuildFolder/services/lib` — Contains `buildInfo.mat`, which you use for building the component model library that represents the generated code compiled against `services.h`.

For more information about generated files and folders, see “Manage Build Process Folders”.

To generate code for the component model and build the component model library, set the `GenCodeOnly` configuration parameter to `'off'` and use the `slbuild` command. If code for the component model is already generated, you can build the component model library by using the `codebuild` command with the path to the `buildInfo.mat` file.

If you only generate code for the component model library, you can build the component model library outside the MATLAB environment by using a CMake workflow. You can create a:

- CMake configuration (`CMakeLists.txt`) file by using the `codebuild` function
- ZIP file by using the `packNGo` function

For more information, see:

- “Deploy Generated Code”
- “Deploy Component Algorithm as Component Model Library by Using CMake”

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Code interface report for service interfaces

In R2022b, when you generate code by using a service code interface configuration, the code interface report documents how the generated code uses services such as data transfer and timer services. The code interface report includes:

- A high-level description of service interface generation
- Interface details for model execution functions, including the `model_initialize`, `model_step`, and `model_terminate` functions
- Interface details for services the model uses, such as data transfer and timer services

For more information, see “Analyze Generated Service Code Interface”. For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Generate code for Reusable custom storage classes with symbolic dimension inputs

Starting in R2022b, you can generate code for the `Reusable` custom storage classes with symbolic dimensions as inputs. Prior to R2022b, code generation for `Reusable` custom storage classes with symbolic dimensions as inputs was not supported.

For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#).

New \$X naming rule token

Use the `$X` token in naming rules for generated sender, receiver, data transfer, and timer service interface access functions. The `$X` token represents the name of the entry-point function that encloses the access function.

For example, this code uses the function naming rule `get_$X_input` for receiver services and `set_$X_output` for sender services.


```

void CD_accumulator(void)
{
    int32_T i;
    for (i = 0; i < 10; i++) {
        CD_sig.delay[i] += (get_CD_accumulator_input())[i];
        (set_CD_accumulator_output())[i] = CD_param.tunable_gain * CD_sig.delay[i];
    }
}

```

The sender and receiver services, `set_CD_accumulator_output` and `get_CD_accumulator_input` respectively, include the name of the enclosing entry-point function, `CD_accumulator`.

For more information about naming rule tokens, see “Identifier Format Control”. For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Example models attached to examples and renamed

In R2022b, these example models have been renamed and are available in the examples indicated in this table.

R2022a model name	New model name	Example
rtwdemo_condinput	ConditionalInput	“Use Conditional Input Branch Execution”
rtwdemo_deadpathElim	DeadPathElimination	“Eliminate Dead Code Paths in Generated Code”
rtwdemo_foreachreuse	ForEachReuse	“Generate Reusable Code from For Each Subsystems”
rtwdemo_col_dlut3d_selplane	ColumnDLUT3DSelectPlane	“Direct Lookup Table Algorithm for Row-Major Array Layout”
rtwdemo_col_dlut3d_selector	ColumnDLUT3DSelectVector	“Direct Lookup Table Algorithm for Row-Major Array Layout”
rtwdemo_row_dlut3d_selplane	RowDLUT3DSelectPlane	“Direct Lookup Table Algorithm for Row-Major Array Layout”
rtwdemo_row_dlut3d_selector	RowDLUT3DSelectVector	“Direct Lookup Table Algorithm for Row-Major Array Layout”
rtwdemo_mdleftop	TopModelCode	“File Packaging for Models (Code and Data)”
rtwdemo_mdleftbot	ReferenceModelCode	“File Packaging for Models (Code and Data)”
rtwdemo_row_interpalg	RowInterpolationAlgorithm	“Interpolation Algorithm for Row-Major Array Layout”
rtwdemo_row_lut2d	RowLUT2D	“Interpolation Algorithm for Row-Major Array Layout”
rtwdemo_udt	UserDefinedDataTypes	“Define Abstract Numeric Types and Rename Types”

R2022a model name	New model name	Example
rtwdemo_sil_block	SILBlock	"Test Generated Code with SIL and PIL Simulations"
rtwdemo_sil_modelblock	SILModelBlock	"Test Generated Code with SIL and PIL Simulations"
rtwdemo_sil_counter	SILCounter	"Test Generated Code with SIL and PIL Simulations"
rtwdemo_sil_topmodel	SILTopModel	"Test Generated Code with SIL and PIL Simulations"
rtwdemo_cppclass	CppClassRateBased	"Configure C++ Class Interface for Rate-Based Models"
rtwdemo_cppclass_expfcn	CppClassExportFunction	"Configure C++ Class Interface for Export-Function Models"
rtwdemo_cppclass_workflow	CppClassWorkflowKeyIgnition	"Generate C++ Code from Simulink Models"
rtwdemo_concurrent_execution	MulticoreConcurrentTasking	"Model Multicore Concurrent Tasking Application"
rtwdemo_multirate_multitasking	MultirateMultitasking	"Model Single-Core, Rate-Monotonic Multitasking Application"
rtwdemo_multirate_singletasking	MultirateSingleTasking	"Model Single-Core, Single-Tasking Application"
rtwdemo_explicitinvocation_atomicsubsys	AtomicSubsystem	"Generate Code for Atomic Subsystems"

New Simulink Model Advisor check for numeric efficiency

You can use the Model Advisor to identify when code generated from a Simulink model will be more efficient if you enable the **Support long long** parameter. This numeric efficiency check alerts you when signals and ports in your model will result in expensive multi-word types in generated code because the `long long` data type is not enabled.

In the Model Advisor, select and run **By Product > Embedded Coder > Check usage of 'long long' data type**. For more information, see "Embedded Coder Checks".

Only explicit usage of signals and ports having data types with word lengths greater than the `long` data type are detected. This check does not flag operation outputs that implicitly have word lengths greater than `long` data type, such as the output of a multiply operation.

Code replacement validation detects ambiguous overflow and rounding modes

In R2022b, when you create a code replacement entry for an operation that can overflow or lose precision during rounding, you must specify the **Integer saturation mode** (`SaturationMode`) or **Rounding mode** (`RoundingModes`) for the entry. When you validate these entries in R2022b, they produce a warning if the required settings are not specified. The new validation check enables you to

identify entries that could lead to unintended replacements in the generated code and produce different results from the model.

Previously, these entries were reported as valid even if the saturation and rounding settings were not specified. When the settings were not specified, operations with different saturation or overflow needs could have mapped to the same code replacement entry, leading to generated code that produced different results from the model. For more information, see “Integer saturation mode” and “Rounding modes”.

Compatibility Considerations

Some code replacement entries that were previously reported as validated produce warnings in R2022b. To make the entries valid, specify the **Integer saturation mode** or the **Rounding mode** for the entries. In a future release, the validation check will produce errors instead of warnings.

Deployment

Retrieve metadata about service interface by using code descriptor programming interface

You can now ease component code integration configured with a service code interface by using the code descriptor programming interface to get metadata about the code interfaces generated for a model. You can use this metadata to declare and define your target platform service functions.

To use the code descriptor programming interface, first create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDesc = coder.getCodeDescriptor(BuildDirectory);
```

Use these methods of the `coder.codedescriptor.CodeDescriptor` object to retrieve metadata about service function declarations.

Goal	Method
Return the service interface object.	<code>getServices</code>
Return the declaration of service function interface in the generated code	<code>getServiceFunctionDeclaration</code>
Return the prototype of generated service function interface.	<code>getServiceFunctionPrototype</code>

Use these methods of the `coder.descriptor.ServiceInterface` object to retrieve metadata about a specified service function.

Goal	Method
Return a list of generated entry-point functions that call a target platform service function.	<code>getCallableFunctionsThatCallServiceFunction</code>
Return a list of the service functions called from a generated entry-point function.	<code>getCalledServiceFunctions</code>
Return the data communication method that the specified service function uses.	<code>getServiceDataCommMethod</code>
Return the service interface object for a service interface type.	<code>getServiceInterface</code>
Return the name of the header file that contains the service interface prototypes.	<code>getServicesHeaderFileName</code>

For more information, see “Get Metadata About Service Interface”.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 1-4.

Target Language Compiler search functions for regular expressions

Starting in R2022b, you can use these Target Language Compiler (TLC) functions to perform operations on regular expressions. For more information, see “Regular Expressions”.

TLC Built-In Functions

Built-In Function Name	Description
CONTAINS(expr1, expr2)	Returns TLC_TRUE if expr1 contains expr2, and TLC_FALSE otherwise. expr1 and expr2 must be strings. For example, CONTAINS("I walk up, they walked up, we are walking up.", "walk(\\w*) up") returns TLC_TRUE.
REGEXP_MATCH(expr1, expr2)	Returns the substrings in expr1 that match the pattern expr2. expr1 and expr2 must be strings. For example, REGEXP_MATCH("I walk up, they walked up, we are walking up.", "walk(\\w*) up") returns ["walk up", "walked up", "walking up"].
REGEXPREP(expr1, expr2, expr3)	Returns a new string that replaces instances of the substring expr2 in string expr1 with the substring expr3. expr1, expr2 and expr3 must be strings. This function supports tokens in replacement string. For example, REGEXPREP("I walk up, they walked up, we are walking up.", "walk(\\w*) up", "ascend\$1") returns "I ascend, they ascended, we are ascending."

For more information, see "Target Language Compiler Directives".

Introducing Embedded Coder Support Package for Linux Applications

Embedded Coder Support Package for Linux® Applications is available from release R2022b. Deploy and prototype AUTOSAR adaptive application components on a Linux target.

The support package includes an application **Linux Runtime Manager**. Use the application to deploy and calibrate the AUTOSAR adaptive model on a Linux target as an adaptive application. You can also use the application to start, stop, or suspend a running AUTOSAR adaptive application on a target.

For more details on installing the support package, see "Support Package Installation".

You can also convert a DDS Blockset model into an AUTOSAR Adaptive model by using the `linux.utils.migrateDds2Adaptive` function and deploy it on the target.

Calibration File Customization

Starting from R2022b, the **Generate Calibration Files** tool remembers the last used settings, such as version of the ASAP2 file, and include or exclude comments, turn off or on the ASAP2 file and CDF file generation. These settings are saved in the MATLAB preferences.

For more information, see "Generate ASAP2 and CDF Calibration Files".

The Embedded Coder allows you to add, delete, modify, find, filter, fetch measurements, characteristics, functions, and compu-methods by using the programming interface.

Also, the new enhancements allow you to

- Insert custom code fragments in different sections of the ASAP2 file.
- Modify the Name and Comments for the project and module sections.
- Provide address extension for the ECU address to measurements, characteristics, and axis points.

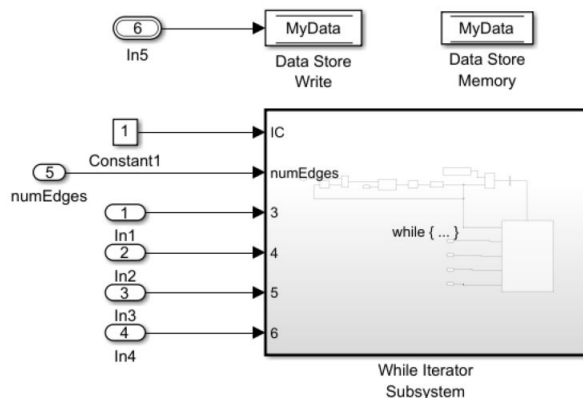
- Insert functions hierarchy by adding function as subfunction in another function.

For more information, see “Customize Generated ASAP2 File”.

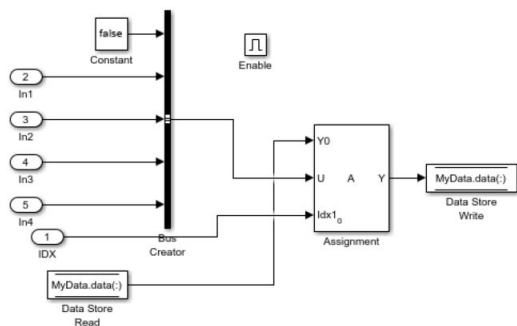
Performance

Data Store Memory block reuse in reusable subsystems inside While Iterator subsystems

Starting in R2022b, the generated code contains fewer data copies for models in which Data Store Memory blocks are read from a reusable subsystem that is inside a While Iterator subsystem. Inside the reusable subsystem, an Assignment block passes the output values to the Data Store Write block in order to write them into the Data Store Memory locations. For example, the `mDSMReuse` model has a While Iterator subsystem and a Data Store Memory block.



The While Iterator subsystem contains the reusable subsystem `Calculate`. Inside the subsystem, bus values are read from a top-level Data Store Memory block `MyData`. The subsystem output values are passed to the Data Store Write block by an Assignment block in order to write them into `MyData`.



In R2022a, the code generator produced this code in `Calculate.c`:

```
if (rtu_Enable) {
    if ((*rtd_WhileIterator_IterationMarker) < 2ULL) {
        *rtd_WhileIterator_IterationMarker = 2U;
        (void)memcpy(&localB->Assignment[0], &rtd_MyData->data[0], 10U *
            (sizeof(SubBus)));
    }
    localB->Assignment[rtu_IDX].flag = false;
    localB->Assignment[rtu_IDX].a1 = rtu_In1;
    localB->Assignment[rtu_IDX].a2 = rtu_In2;
    localB->Assignment[rtu_IDX].a3 = rtu_In3;
    localB->Assignment[rtu_IDX].a4 = rtu_In4;
}
```

```

        (void)memcpy(&rtd_MyData->data[0], &localB->Assignment[0], 10U * (sizeof
                    (SubBus)));
    }

```

The generated code unnecessarily first copied bus elements to the local variable, `localB`, and then updated the `rtd_MyData` variable.

In R2022b, the code generator produces this code in `Calculate.c`:

```

if (rtu_Enable) {
    if ((*rtd_WhileIterator_IterationMarker) < 2ULL) {
        *rtd_WhileIterator_IterationMarker = 2U;
    }
    rtd_MyData->data[rtu_IDX].flag = false;
    rtd_MyData->data[rtu_IDX].a1 = rtu_In1;
    rtd_MyData->data[rtu_IDX].a2 = rtu_In2;
    rtd_MyData->data[rtu_IDX].a3 = rtu_In3;
    rtd_MyData->data[rtu_IDX].a4 = rtu_In4;
}

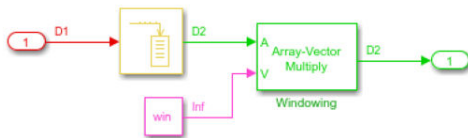
```

The code does not contain the local variable `localB` and the data copies, which improves RAM efficiency of generated code. For more information, see “Data Copy Reduction for Data Store Read and Data Store Write Blocks”.

Removed redundant multirate block output buffers

Before R2022b, the code generator generated redundant output buffers for models that contained a multi-rates block, and another block whose output sample time was the same as the multi-rates block output sample time. Starting in R2022b, you can generate optimized code that does not contain the unnecessary output buffer for the multirate block whenever possible. Eliminating redundant buffers reduces data copies and improves RAM consumption. To enable this optimization, select the **Reuse local block outputs** parameter.

Consider the `mBufferReuse` model that has a Buffer block whose input signal (indicated by the red signal) has D1, and the output signal (indicated by the green signal) has D2 sample time. The output signal of `Windowing` also has D2 sample time.



In R2022a, the code generator produced this code:

```

/* S-Function (sdspdmult2): '<Root>/Windowing' incorporates:
 * Buffer: '<Root>/Buffer'
 * Constant: '<Root>/Constant'
 */
idxS = 0;
for (i = 0; i < 2; i++) {
    idxV = 0;
    for (k = 0; k < 256; k++) {
        mBufferReuse_Y.Outputport[idxS] = mBufferReuse_B.bufferUp[idxS] *
            mBufferReuse_ConstP.Constant_Value[idxV];
    }
}

```



```

        idxS++;
        idxV++;
    }
}

```

This code included a redundant output buffer `mBufferReuse_B.bufferUp` for the Buffer block and the root output buffer `mBufferReuse_Y.Outputport`.

In R2022b, the code generator produces this code:

```

/* S-Function (sdspdmult2): '<Root>/Windowing' incorporates:
 * Buffer: '<Root>/Buffer'
 * Constant: '<Root>/Constant'
 */
idxS = 0;
for (i = 0; i < 2; i++) {
    idxV = 0;
    for (k = 0; k < 256; k++) {
        mBufferReuse_Y.Outputport[idxS] *= mBufferReuse_ConstP.Constant_Value[idxV];
        idxS++;
        idxV++;
    }
}

```

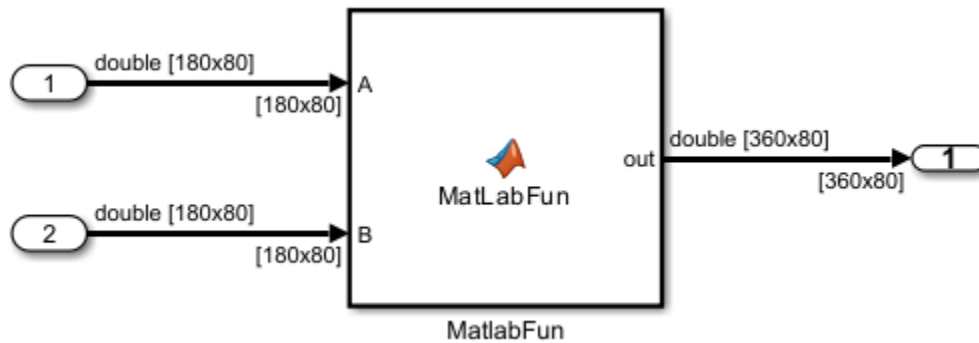
The generated code does not contain the `mBufferReuse_B.bufferUp` buffer, which reduces RAM consumption of the code. For more information, see “Enable and Reuse Local Block Outputs in Generated Code”.

Buffer reuse optimization for referenced models

Before R2022b, for models containing referenced models, the code generator generated unique buffers for holding referenced model outputs. Starting in R2022b, you can generate optimized code that reuses signal buffers or generates reusable buffers for referenced model outputs whenever possible. To enable this optimization, in the Configuration Parameters dialog box, select the new parameter **Reuse output buffers of Model blocks**. For more information, see “Reduce Memory Usage for Models Containing Referenced Models”.

Improved cache efficiency of generated code containing loop distribution, interchange, and reversal

In R2022b, you can generate optimized code containing loop interchange and distribution. These loop transformations increase the number of cache hits and improve the code execution time. The optimizations apply to code generation targets for which the cache information is available to the code generator. To increase the availability of cache information to the code generation target, specify the target hardware information by using the configuration parameters “Device vendor” and “Device type”.



For example, in this model, the MATLAB Function block contains code that performs operations on the elements of the two input matrices of dimension [180x80] by using for-loops.

```
function out = MatLabFun(A, B)

sizeRow=90;
sizeCol=80;

for i = 2 : sizeRow
    for j = 2 : sizeCol
        B(i*2,j) = B((i*2)-1,j)+i*j;
        for k = 2 : sizeCol
            A(i*2,k) = A(i-1,k)+i*j;
        end
    end
end
end

out = [A;B];
end
```

In R2022a, the code generator produced code that contains one loop nest that evaluates the loop with iteration variable `B_tmp` at the innermost position.

```
/* Output and update for atomic system: '<Root>/MatlabFun' */
void MatlabFun(void)
{
    int32_T B_tmp;
    int32_T B_tmp_tmp;
    int32_T i;
    int32_T j;

    /* Inport: '<Root>/In1' */
    memcpy(&rtDW.A[0], &rtU.In1[0], 14400U * sizeof(real_T));

    /* Inport: '<Root>/In2' */
    memcpy(&rtDW.B_m[0], &rtU.In2[0], 14400U * sizeof(real_T));
    for (i = 0; i < 89; i++) {
        for (j = 0; j < 79; j++) {
            B_tmp_tmp = (i + 2) << 1;
            B_tmp = (j + 1) * 180 + B_tmp_tmp;
            rtDW.B_m[B_tmp - 1] = (real_T)((i + 2) * (j + 2)) + rtDW.B_m[B_tmp - 2];
            for (B_tmp = 0; B_tmp < 79; B_tmp++) {
                int32_T A_tmp;
```

```

        A_tmp = (B_tmp + 1) * 180;
        rtDW.A[(B_tmp_tmp + A_tmp) - 1] = (rtDW.A[A_tmp + i] + ((real_T)i + 2.0))
            + ((real_T)j + 2.0);
    }
}
}

/* Outputport: '<Root>/Out1' */
for (i = 0; i < 80; i++) {
    for (j = 0; j < 180; j++) {
        B_tmp_tmp = 180 * i + j;
        B_tmp = 360 * i + j;
        rtY.Out1[B_tmp] = rtDW.A[B_tmp_tmp];
        rtY.Out1[B_tmp + 180] = rtDW.B_m[B_tmp_tmp];
    }
}

/* End of Outputport: '<Root>/Out1' */
}

```

In R2022b, the loop in the generated code is distributed to two loop nests. The loop nests are interchanged to evaluate the loop with iteration variable j at the innermost position.

```

void MatlabFun(void)
{
    int32_T A_tmp;
    int32_T B_tmp;
    int32_T i;
    int32_T j;

    /* Inport: '<Root>/In1' */
    memcpy(&rtDW.A[0], &rtU.In1[0], 14400U * sizeof(real_T));

    /* Inport: '<Root>/In2' */
    memcpy(&rtDW.B_m[0], &rtU.In2[0], 14400U * sizeof(real_T));
    for (j = 0; j < 79; j++) {
        for (i = 0; i < 89; i++) {
            B_tmp = ((i + 2) << 1) + (j + 1) * 180;
            rtDW.B_m[B_tmp - 1] = (real_T)((i + 2) * (j + 2)) + rtDW.B_m[B_tmp - 2];
        }
    }

    for (B_tmp = 0; B_tmp < 79; B_tmp++) {
        for (i = 0; i < 89; i++) {
            for (j = 0; j < 79; j++) {
                A_tmp = (B_tmp + 1) * 180;
                rtDW.A[(((i + 2) << 1) + A_tmp) - 1] = (rtDW.A[A_tmp + i] + ((real_T)i +
                    2.0)) + ((real_T)j + 2.0);
            }
        }
    }

    /* Outputport: '<Root>/Out1' */
    for (j = 0; j < 80; j++) {
        for (i = 0; i < 180; i++) {
            B_tmp = 180 * j + i;
            A_tmp = 360 * j + i;
            rtY.Out1[A_tmp] = rtDW.A[B_tmp];
        }
    }
}

```

```
        rtY.Out1[A_tmp + 180] = rtDW.B_m[B_tmp];
    }
}

/* End of Outport: '<Root>/Out1' */
}
```

This interchange improves the locality of reference for the loop nest and improves cache performance.

Generate SIMD code for Discrete FIR Filter block

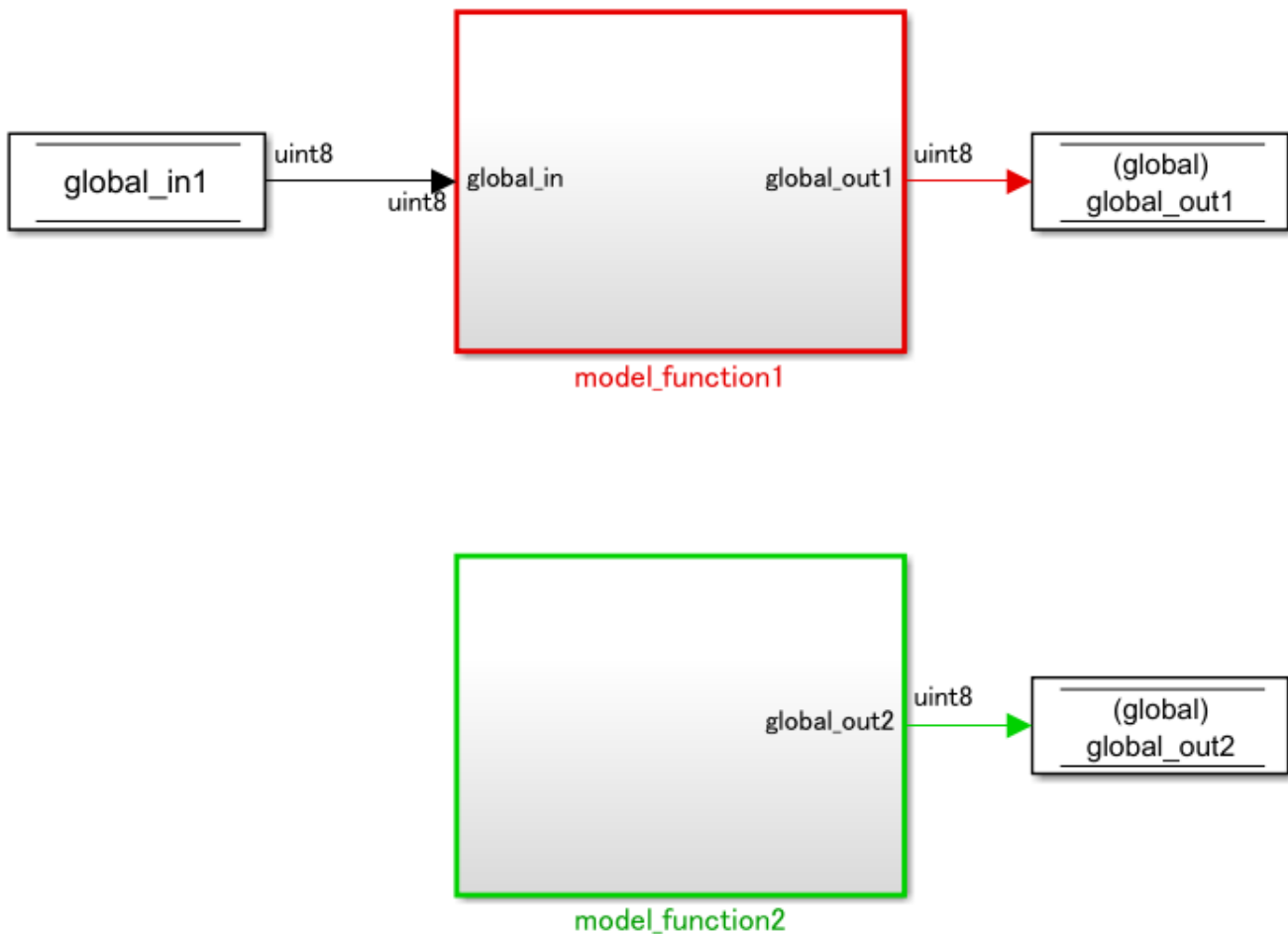
In R2022b, if you have DSP System Toolbox™, you can generate SIMD code for the Discrete FIR Filter block. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel® platforms. To generate SIMD code from the Discrete FIR Filter block, set these configuration parameters:

- **Leverage target hardware instruction set extensions** — Specify an instruction set to use.
- **Optimize reductions** — Select the **Optimize reductions** parameter.
- **Priority** — Select Maximize execution speed.

Your model must meet the requirements for code generation described in Discrete FIR Filter and “Generate SIMD Code from Simulink Blocks”.

Improved function argument generation eliminates extra global variable assignment

In R2022b, the code generator eliminates unnecessary global variable assignments when the code can use the global variable as a function call argument instead.



This example model contains two model functions that write global outputs. In R2022a, the code generator produced code in this pattern in the model step function. The code contained an extra value assignment for the variable `global_out1`.

```
model_function1(global_in1, &global_out2);
global_out1 = global_out2;
model_function2(&global_out2);
```

In R2022b, for some cases, the code generator produces code in this pattern in the model step function. The variable `global_out1` is an argument of the first function and the code does not contain the extra line that assigns the variable value.

```
model_function1(global_in1, &global_out1);
model_function2(&global_out2);
```

This code enhancement occurs when:

- The first-called function defines the variable on the right side of the assignment and passes the variable to the second-called function.
- The two functions pass the variable as an argument by reference.
- The second-called function reassigns the value of the variable, which means that the second function does not use the value defined by the first function.

SIMD code for bitwise and shift operations

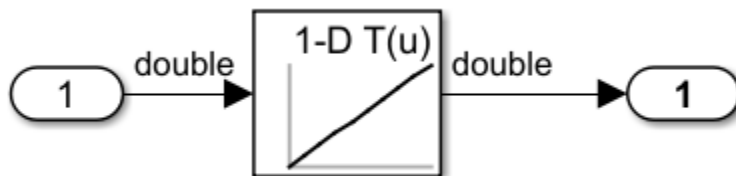
In R2022b, you can generate SIMD code for bitwise operations and shift operations. When you select an instruction set by using the Leverage target hardware instruction set extensions parameter, the generated code includes the associated instructions for these bitwise operations and shift operations:

- Bitwise AND
- Bitwise OR
- Bitwise XOR
- Shift arithmetic

For more information, see “Generate SIMD Code from Simulink Blocks”.

Code replacement for lookup tables that support differently sized table and breakpoint objects

In R2022b, you can replace code from Lookup table blocks that support differently sized table and breakpoint objects by using a code replacement library. If a Lookup table block uses a lookup table object that has the new parameter **Allow multiple instances of this type to have different table and breakpoint sizes** selected, the generated function signature for the lookup table uses pointer arguments for the table and breakpoint data. The pointer arguments allow multiple instances of the table to have different table and breakpoint sizes.



For example, this model uses a 1-D Lookup table block that uses a lookup table object for the data specification. The generated step function calls the lookup table function using the table and breakpoint data fields from the lookup table object as arguments.

```

look1_lu16n15_linlcapw(LookupModel_LookupCRL_U.In1,
    LookupModel_LookupCRL_P.dlutObj.BP1,
    LookupModel_LookupCRL_P.dlutObj.Table,
    LookupModel_LookupCRL_P.dlutObj.N1 - 1U);
  
```

When the lookup table object does not use the new parameter **Allow multiple instances of this type to have different table and breakpoint sizes**, the generated lookup table object uses arrays for the breakpoint and table data. The function call passes the corresponding arguments as arrays.

```

typedef struct {
    uint32_T N1;
    real_T BP1[10];
    real_T Table[10];
} dlutObj_type;
  
```

To replace the lookup table function call in this case, you use a code replacement entry that specifies the conceptual function arguments as matrix arguments for the table and breakpoint data arguments.

In R2022a and earlier, you used this method and did not have the option to use pointers for the table and breakpoint arguments in the lookup table function replacement.

In R2022b, when the new parameter **Allow multiple instances of this type to have different table and breakpoint sizes** is selected for the lookup table object, the generated lookup table object uses pointers for the breakpoint and table data. The function call passes the corresponding arguments as pointers.

```
typedef struct {
    uint32_T N1;
    real_T *BP1;
    real_T *Table;
} dlutObj_type;
```

To replace the lookup table function call when using the new parameter, in the code replacement entry, configure the conceptual arguments for the table and breakpoint data as pointers. To use pointers for conceptual arguments, you must create the entry programmatically. For more information, see “Lookup Table Function Code Replacement”.

Code execution profiling for models that use GRT system target files

For models that use GRT system target files, you can produce execution time profiles for generated code by running software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations. Previously, code execution profiling was possible only for models that used ERT or ERT-based system target files.

For more information, see “Configure Code Execution Profiling”.

Task scheduling visualization with XCP external mode simulations

During XCP external mode simulations, use the Simulation Data Inspector to observe task scheduling and related CPU core activity. To regenerate displays, use the `schedule` function.

For more information, see “Visualize Task Scheduling” and “Visualize Task Scheduling in XCP External Mode Simulation”.

Optimized bandwidth usage during XCP external mode profiling

In an XCP-based external mode simulation, after the execution of a task, the target application uploads data samples from the profiling buffer. The application associates the data samples from the profiling buffer with the same simulation time. If the buffer contains only a few samples, use of the communication channel bandwidth is suboptimal.

To improve use of bandwidth during data uploading, specify the display of absolute time:

```
set_param(modelName, 'CodeProfilingXCPUseAbsoluteTime', 'on')
```

When the external mode simulation runs, the target application uploads data samples only when the profiling buffer is full. The Simulation Data Inspector displays streamed values with respect to absolute time instead of simulation time.

For more information, see “Display Absolute Time”.

Verification

SIL or PIL block workflow

In a SIL or PIL block workflow, when you right-click a subsystem and select **C/C++ Code > Build This Subsystem**, the software immediately starts the subsystem build process that creates a SIL or PIL block for the generated subsystem code. In earlier releases, the software opens a window, and you need to click the **Build** button. For more information, see “SIL or PIL Block Simulation”.

Reusable subsystems with input signals that map to const variables

The SIL/PIL atomic subsystem workflow now supports reusable subsystems with input signals that map to const variables in the generated code. Previously, the SIL/PIL simulation produced an error. For example:

```
Inport Const ('TestModel/AtomicSub/Const') is read-only in the generated code,  
and is therefore not supported by SIL or PIL simulation. Change its storage  
class so that it is writable in the generated code.
```

For more information, see “Unit Test Subsystem Code with SIL/PIL Manager”.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2022a

Version: 7.8

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Removal of unused class properties in generated C/C++ code

In R2022a, the default behaviour of code generator is to remove unused class properties or structure fields in the generated C/C++ code.

The `PreserveUnusedStructFields` property supports standalone build types: static library, dynamically linked library, and executable.

To preserve the unused properties of the classes or structures in the generated code, use either of these settings:

- Set the `PreserveUnusedStructFields` property to `true`.
- Open the MATLAB Coder app. On the **Memory** tab, select the **Preserve unused fields and properties** option.

This table compares the generated code to the `PreserveUnusedStructFields` set to `true` and the `PreserveUnusedStructFields` set to `false`.

MATLAB Code	Generated Code with <code>cfg.PreserveUnusedStructFields = false;</code> (default)	Generated Code with <code>cfg.PreserveUnusedStructFields = true;</code>
<pre> classdef myClass properties a b c end methods function obj = myClass(x) coder.inline('never') obj.a = x; obj.b = x + 1; obj.c = x + 2; end end end function y = myAdd(n) o = myClass(n); y = o.a + o.b; end </pre>	<pre> % unused property 'c' is removed class myClass { public: void init(double x); double a; double b; }; </pre>	<pre> % unused property 'c' is present class myClass { public: void init(double x); double a; double b; double c; }; </pre>

For more information, see [Removal of Unused Class Properties in the Generated C/C++ Code](#).

Reduction of violations for MISRA C:2012, MISRA C++:2008, and AUTOSAR C++14 rules in generated code

In R2022a, the generated code has fewer violations of several rules in the required categories of MISRA C:2012, MISRA C++:2008, and AUTOSAR C++14 coding standards. Some of these rules are:

- MISRA C:2012 Rule 5.6 (Polyspace Bug Finder), MISRA C:2012 Rule 10.3 (Polyspace Bug Finder), MISRA C:2012 Rule 21.2 (Polyspace Bug Finder), MISRA C:2012 Rule 21.8 (Polyspace Bug Finder)
- MISRA C++:2008 Rule 3-2-3 (Polyspace Bug Finder), MISRA C++:2008 Rule 4-10-2 (Polyspace Bug Finder), MISRA C++:2008 Rule 5-0-3 (Polyspace Bug Finder), MISRA C++:2008 Rule 5-19-1 (Polyspace Bug Finder), MISRA C++:2008 Rule 6-5-4 (Polyspace Bug Finder)
- AUTOSAR C++14 Rule A0-1-2 (Polyspace Bug Finder), AUTOSAR C++14 Rule A7-1-6 (Polyspace Bug Finder), AUTOSAR C++14 Rule A7-2-3 (Polyspace Bug Finder), AUTOSAR C++14 Rule A8-5-2 (Polyspace Bug Finder), AUTOSAR C++14 Rule A12-7-1 (Polyspace Bug Finder)

For more information on how to generate code that has improved MISRA and AUTOSAR compliance, see [Generate C/C++ Code with Improved MISRA Compliance](#).

Stack usage profiling for code generated from MATLAB code

To determine the size of stack memory that is required to run generated code, you can run a software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution that generates a stack usage profile. The execution creates a code stack profiling report that shows minimum, average, and maximum memory demand. For each function call, the execution streams memory usage measurements to the Simulation Data Inspector, which enables you to analyze stack usage variation. You can use stack usage profiles to observe the effect of compiler optimization and data input. For more information, see [Stack Usage Profiling for Code Generated From MATLAB Code](#).

Identification of performance bottlenecks in generated code

The code execution profiling report generated by a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution contains a new section **Execution Times in Percentages**. The section displays function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code. For more information, see [Code Execution Profiling Report](#).

Model Architecture and Design

Symbolic dimension inputs for Squeeze block

Starting in R2022a, you can generate code for the Squeeze block that has symbolic dimensions as inputs. Prior to R2022a, generating code for Squeeze block that had symbolic dimensions as input was not supported.

For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#).

Embedded Coder Dictionary interface improvements

In R2022a, the Embedded Coder Dictionary interface is enhanced to better reflect how your code interface definitions control the generated code for your target platform. In the left pane of the dictionary, your definitions are organized in sections. The set of code definitions in your dictionary configuration is called an application platform definition because the definitions define how the generated application code interacts with the platform.

The **Functions** section contains a subsection for creating function customization templates. The **Memory** section contains sub-sections for creating storage classes and memory sections. For a dictionary stored in a .SLDD file, the sections also contain subsections for selecting the default definitions for categories of functions and model data elements. For more information, see [Embedded Coder Dictionary](#).

Code Interface Configuration and Integration

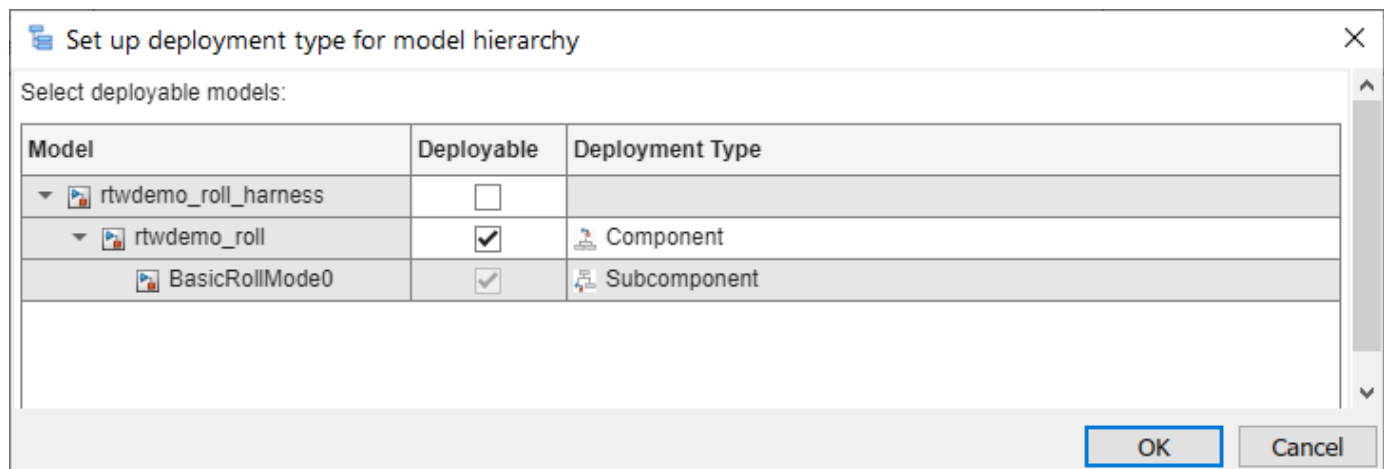
Control code interface generated for models by specifying deployment types

In R2022a, when you use a model hierarchy to generate code, you can control the code interfaces at the different levels of the hierarchy by setting the deployment type for each model. The code generator uses the deployment type to:

- Enforce peer and nesting rules in the model hierarchy
- Map model elements to code interface definitions
- Generate code that uses the appropriate interface to connect to other parts of the hierarchy

You can specify these deployment types for your models:

- **Component** - The top model for code generation. The code generator produces a standalone algorithm. The component code exposes its interface to other components in the system.
- **Subcomponent** - A model reference that a component model uses. The generated code entry points are symbolically scoped to the parent component.
- **Automatic** - Embedded Coder determines the deployment type based on the model hierarchy context.
- **Simulation only** - A model that is for simulation only. You do not generate code for a simulation only model. For example, a plant model that you use for simulation testing is non-deployable.



You can programmatically set the deployment type for a model using the `setDeploymentType` function of a `coder.mapping.api.CodeMapping` object. For example, to set the deployment type of the `sldemo_fuelsys/fuel_rate_control` model to `Component`, use the `setDeploymentType` function.

```
coder.mapping.create('fuel_rate_control');
cm = coder.mapping.api.get('fuel_rate_control');
setDeploymentType(cm, 'Component');
```

To view the deployment type for a model, use the `getDeploymentType` function.

For more information, see [Configure Deployment Types for Model Hierarchy](#).

Changes to class namespaces and default class name in C++ generated code

Nested namespace support

C++ code generation now supports nested namespaces for the model class.

For more information, see [Interactively Configure C++ Interface](#) and [Programmatically Configure C++ Interface](#).

Default class name in C++ generated code is model name

Beginning in R2022a, the default class name in C++ generated code is the name of the model. Previously, the class names in the generated code used a default class name of the form *modelModelClass*. The new default is of the form *model*.

Compatibility Considerations

This change to the generated model class names might cause integration scripts that use the class names to break. Update integration scripts to the new generated class names. For more information, see [C++ Data and Function Interfaces](#).

Calibration file customization

Starting in R2022a, the code generator produces an ASAP2 file that reflects these enhancements:

- Includes a default event list in the IF_DATA section.
- Excludes pointer variables.
- Aligns content of the Record_layouts.a2l file with the version of the ASAP2 file.

You can further customize the ASAP2 file as follows:

- Exclude 64-bit integer elements from ASAP2 file.
- Exclude structure elements from ASAP2 file.
- Specify additional address information.

For more information, see [Customize Generated ASAP2 File](#).

Memory section mapping for grouped entry-point functions

In R2021b, if the model configuration parameter **Generate separate internal data per entry-point function** was enabled and on the **Data Defaults** tab, category **Signals, states, and internal data** is mapped to a memory section, the code generator produced a warning.

For example, consider the `rtwdemo_memsec` model, for which the code generator produced this code:

```
/* Internal Data Grouped For Same Function */  
FuncInternalData0 rtFuncInternalData0; /* '<Root>/Unit Delay' */
```


In R2022a, the code generator produces a memory section for each entry point function's grouped internal data in the generated code.

For example, for the `rtwdemo_memsec` model, the code generator produces this code:

```
/* Internal Data Grouped For Same Function */  
  
/* This memory is of moderate speed and cost */  
#pragma MEDIUM_MEM(rtFuncInternalData0)  
  
FuncInternalData0 rtFuncInternalData0; /* '<Root>/Unit Delay' */
```

For more information, see [Control Data and Function Placement in Memory by Inserting Pragmas](#).

Code Generation

Regular expression token decorators to modify certain tokens

Starting in R2022a, you can use regular expressions in token decorators to modify \$G, \$N, and \$R tokens. Enclose the token decorator in double quotes and use two regular expressions separated by a forward slash. The code generator uses the first regular expression to match substrings of the token and uses the second regular expression to replace those substrings.

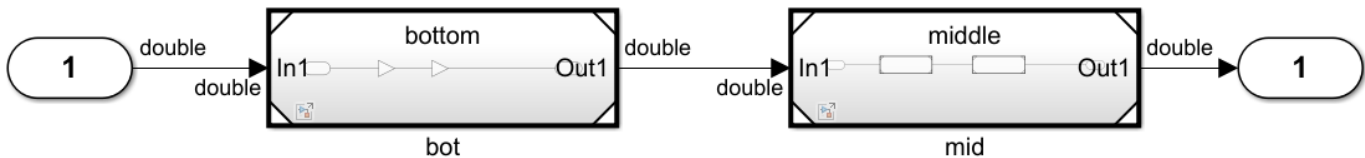
For example, this identifier naming rule takes the root model name \$R and replaces instances of a with b: \$R["a/b"].

For more information, see Identifier Format Control.

Improved comments for code that initializes instance-specific values for model arguments

Starting in R2022a, the code generator adds comments to the code that initializes the instance-specific values for model arguments. The comments display the parameter name and full path where the parameter is defined.

Consider the following model, top.



The top model references two other models, middle and bottom. The middle model contains two references to the bottom model. The bottom model has two model parameters, P and Q. There are a total of six instance-specific values for these parameters across three instances of the bottom model. Suppose the six parameters are set to 3.

Before R2022a, the code generator produced this code:

```

/* instance parameters */
InstP rtInstP = {
    {
        3.0,
        3.0
    },
    /* instance parameters for '<Root>/bot' */
    {
        {
            3.0,
            3.0
        },
        {
            3.0,
            3.0
        }
    }
    /* instance parameters for '<Root>/mid' */
};

```

The generated comments showed only one level of depth in the model hierarchy and did not identify the parameters by name. The parameters were difficult to tune, especially when multiple parameters had the same value.

Starting in R2022a, the code generator generates comments for each parameter. The comments improve the traceability of the model arguments by displaying each parameter name and the full path to the model that defines each parameter.

```

/* instance parameters */
InstP rtInstP = {
    {
        /*
         * top/bot
         *   bottom:P
         */
        3.0,

        /*
         * top/bot
         *   bottom:Q
         */
        3.0
    },
    {
        {
            /*
             * top/mid
             *   middle/bot1
             *     bottom:P
             */
            3.0,

            /*
             * top/mid
             *   middle/bot1
             *     bottom:Q
             */
            3.0
        },
        {
            /*
             * top/mid
             *   middle/bot2
             *     bottom:P
             */
            3.0,

            /*
             * top/mid
             *   middle/bot2
             *     bottom:Q
             */
            3.0
        }
    }
};

```

For more information about generating comments, see [Configure Code Comments](#).

New parentheses level for MISRA standard compliance and code readability

Starting in R2022a, the code generator provides a new option for parenthesization style that enables you to generate more readable code by reducing the parentheses. The generated code is compliant with MISRA C:2012 Standards as defined in Rule 12.1.

To apply the new parentheses level, for model configuration parameter **Parentheses level**, select Standards (Parentheses for Standards Compliance).

For more information, see Parentheses level and MISRA C:2012 Rule 12.1 (Polyspace Code Prover).

Improved code readability by adding "U" suffix to unsigned integer constants

Starting in R2022a, the code generator produces code that applies the "U" suffix to the unsigned integer constants. This suffix improves the code readability and is in accordance with the MISRA C:2012 Rule 7.2 coding standard. Prior to R2022a, the code generator provided type casts instead of the "U" suffix in some cases.

Before R2022a	After R2022a
<pre>if((int32_T)reproMissingSuffix_U.In1 < 100) /* Output: '<Root>/Out1' */ reproMissingSuffix_U.Out1 = 100U;</pre>	<pre>if(reproMissingSuffix_U.In1 < 100U) /* Output: '<Root>/Out1' */ reproMissingSuffix_U.Out1 = 100U;</pre>

For more information, see MISRA C:2012 Rule 7.2 (Polyspace Code Prover).

Changes to initialization

Starting in R2022a, the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** parameters apply only to data that will be defined in a generated C file and for which you do not specify initialization in an Initialize Function block.

- If you specify values in an Initialize Function block, the code generator explicitly initializes those values and ignores the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** parameters.
- If the data is not defined by any generated C file, but you provide it by external code, for instance, due to the use of a storage class with the **Imported** scope, the code generator ignores these parameters and does not explicitly initialize this data to zero unless you specify the data in an Initialize Function block.

Before R2022a, the code generator did not explicitly initialize values that had a custom storage class with **Data initialization** set to None. Starting in R2022a, the code generator explicitly initializes these values if you specify them in an Initialize Function block.

AUTOSAR C++14 Rule A12-4-2 violation resolution

When you set **Language** as C++ and **Standard math library** as C++11 (ISO), to resolve some violations of AUTOSAR C++14 Rule A12-4-2 (Polyspace Bug Finder), the code generator adds the `final` keyword in the class definition.

In R2021b, the code generator produced this code:

```
class ModelClass {
...
}
```

In R2022a, the code generator produces this code:

```
class ModelClass final {
...
}
```

For more information, see [Configure Standard Math Library for Target System](#).

AUTOSAR C++14 Rule A12-0-1 violation resolution

When you set **Language** as C++ and **Standard math library** as C++11 (ISO), to resolve some violations of AUTOSAR C++14 Rule A12-0-1 (Polyspace Bug Finder), the code generator declares the move constructor and move assignment operator and the copy and move operation.

In R2021b, the code generator produced this code:

```
class ModelClass {
    ~ModelClass();
    ModelClass(ModelClass const &) = delete;
    ModelClass& operator=(ModelClass const &) = delete;
...
}
```

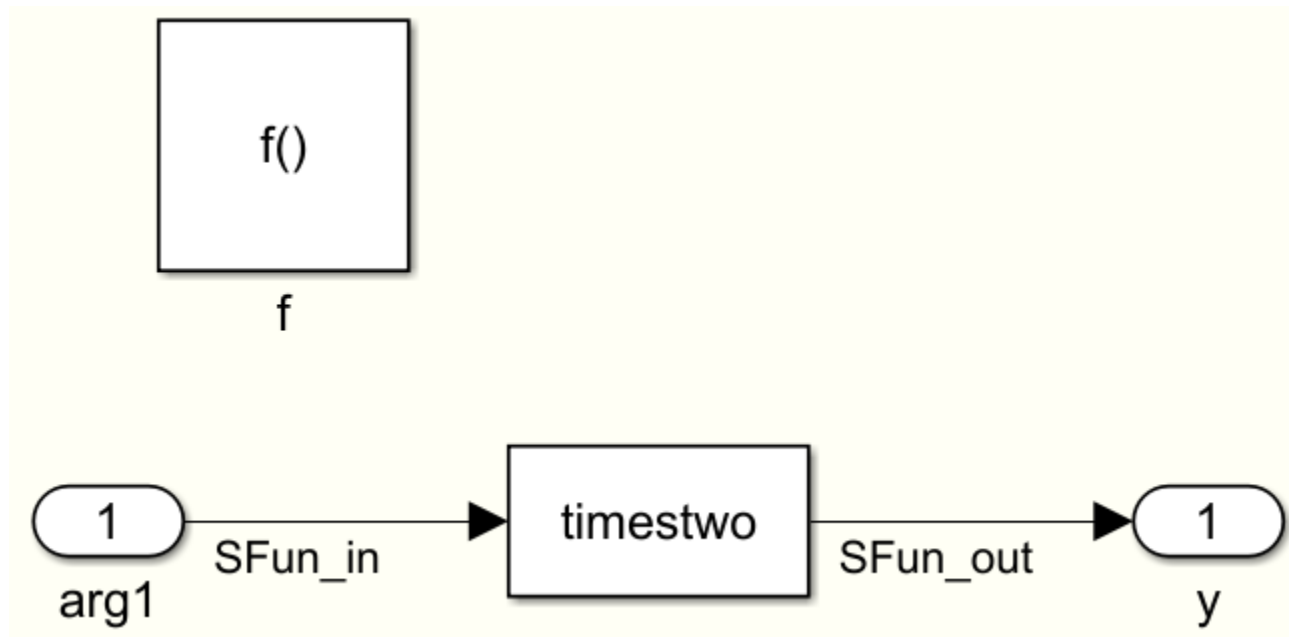
In R2022a, the code generator produces this code:

```
class ModelClass {
    ~ModelClass();
    ModelClass(ModelClass const &) = delete;
    ModelClass(ModelClass &&) = delete;
    ModelClass& operator=(ModelClass const &) = delete;
    ModelClass& operator= (ModelClass &&) = delete;
...
}
```

For more information, see [Configure Standard Math Library for Target System](#).

Removed redundant S-function output buffer

Before R2022a, the code generator generated a redundant output buffer for an S-Function block that was the last block of a Simulink Function block in a Stateflow® chart. Suppose an S-Function block is defined as follows and resides inside a Simulink Function block that is inside a Stateflow chart.



Before R2022a, the code generator produced this code:

```
/* Model step function */
void model_step(void)
{
  /* Inport: '<Root>/In1' */
  model_B.arg1 = model_In1;

  /* Chart: '<Root>/Chart' incorporates:
   * SubSystem: '<S1>/timestwo'
   */
  /* S-Function (timestwo): '<S2>/S-Function' */
  /* Multiply input by two */
  model_B.SFun_out = model_B.arg1 * 2.0;

  /* Output: '<Root>/Out1' */
  model_Out1 = model_B.SFun_out;
}
```

This code included a redundant output buffer `model_B.SFun_out` for the S-Function block and the root output buffer `model_Out1`.

Starting in R2022a, the code generator omits the S-Function output buffer if the S-Function output ports are configured as `REUSABLE_AND_LOCAL`.

```
/* Model step function */
void model_step(void)
{
  /* Inport: '<Root>/In1' */
  model_B.arg1 = model_In1;

  /* Chart: '<Root>/Chart' incorporates:
   * SubSystem: '<S1>/timestwo'
   */
  /* Output: '<Root>/Out1' incorporates:
```

```

    * S-Function (timestwo): '<S2>/S-Function'
    */
    /* Multiply input by two */
    model_Out1 = model_B.arg1 * 2.0;
}

```

RAM consumption is reduced in the generated code. For more information, see [S-Functions That Specify Port Scope and Reusability](#).

C++ Code Generation for client-server interfaces

R2022a enables you to generate C++ code for client-server function interfaces. For more information, see [Client-Server Communication Interfaces](#).

C++ code generation for new Message Triggered Subsystem and Message Polling Subsystem blocks to control event-triggered execution of messages

Generate C++ code for message-triggered subsystems by using the new Simulink Message Triggered Subsystem and Message Polling Subsystem blocks. These blocks are each a type of conditionally executed subsystem that uses messages as the control signal. The information contained in the messages is accessible inside the subsystem.

- A Message Triggered Subsystem block executes whenever a message is available at the control port, independent of block sample time.
- A Message Polling Subsystem block polls messages from a queue periodically based on its sample time and executes only when a message is available.

A Message Triggered Subsystem block can be used to define an independent function at the root level of an export-function model. For more information, see [Export-Function Models Overview](#). For more information about the Message Triggered Subsystem and Message Polling Subsystem blocks, see [Message Triggered Subsystem](#). For more information about how to generate C++ code for these event-triggered subsystems, see [Client-Server Communication Interfaces](#).

CustomSymbolStrUtil parameter available for C++ and AUTOSAR code generation

In R2021a, the Shared utilities identifier format parameter was removed. This parameter is now available for C++ and AUTOSAR code generation. For C code generation, use the Embedded Coder dictionary to create a function customization template that specifies the naming rule, then apply the template by using the Code Mappings editor. For more information, see [Configure Naming of Generated Functions](#).

Functionality being removed or changed

Include guards required in imported header files

Behavior change

Starting in R2022a, updates in how Embedded Coder handles the file packaging of generated code modules might impact your generated code:

- In most cases, a generated source file includes header files that export the symbols used in that source file directly, rather than transitively through another header file.
- To prevent linker errors, you must add include guards, such as `#pragma once`, to the beginning of an imported header file.
- The number of redundant `#include` statements might be reduced.
- The order of `#include` statements might change.

For more information on adding guards to imported header files, see [Control File Placement of Custom Data Types](#).

Deployment

TLC function STRNREP for string replacement

You can use the new Target Language Compiler (TLC) function STRNREP for string manipulations. The TLC function STRNREP(expr1, expr2, expr3, value) accepts the string expr1. The function returns a new string that replaces the substring expr2 present in expr1 with the string expr3 for the number of instances specified in value.

For example, STRNREP("abcabcabc", "abc", "ABC", 2) returns ABCABCabc. For more information, see Target Language Compiler Directives.

Configuration Parameter dialog box no longer lists VxWorksExample as a setting for parameter Target operating system

For model configuration parameter **Target operating system**, the Configuration Parameter dialog box no longer lists parameter value VxWorksExample. For backward compatibility, setting the command line version of the parameter, TargetOS, to VxWorksExample, is still valid. For alternative parameter settings, see **Target operating system**.

Texas Instruments C2000: Support for Texas Instruments F28003x processor

The support package now provides code generation support for the TI F28003x processor, peripherals such as ADC, ePWM, GPIO, SCI, SPI, I2C, Watchdog, X-BAR, and interrupts. The support package also supports external mode over XCP on Serial and PIL simulation. For more information, see F28003x (c28003xlib) (Embedded Coder Support Package for Texas Instruments C2000 Processors).

Texas Instruments C2000: Support for F28M35x (C28x), F28M36x (C28x), and ARM Cortex-M3 Core

- The Embedded Coder Support Package for Texas Instruments™ C2000™ Processors now supports TI Concerto processors such as F28M35x (C28x), F28M36x (C28x), and ARM Cortex-M3 Core. Previously, these processors were supported only in the Embedded Coder Support Package for Texas Instruments C2000 F28M3x Concerto™ Processors. The Embedded Coder Support Package for Texas Instruments C2000 F28M3x Concerto Processors will stop supporting the TI Concerto processors in a future release.
- ADC, AnalogIO, COMP, eCAP, ePWM, eQEP, GPIO, I2C, SCI, SPI, Software Interrupt Trigger, and SPI Master Transfer support for F28M35x (C28x) and F28M36x (C28x) core.
- GPIO, Hardware Interrupt, TCP, UDP, and UART blocks are supported for ARM Cortex-M3 core.

Embedded Coder Support Package for STMicroelectronics Discovery Boards renamed to Embedded Coder Support Package for STMicroelectronics STM32 Processors

Starting in R2022a, the Embedded Coder Support Package for STMicroelectronics® Discovery Boards has been renamed to Embedded Coder Support Package for STMicroelectronics STM32 Processors to support STM32F7xx, STM32G4xx, and STM32H7xx MCU boards.

Support for STMicroelectronics STM32F7xx, STM32G4xx, and STM32H7xx-based Boards

- Use the Embedded Coder Support Package for STMicroelectronics STM32 Processors to generate and build code using an STM32CubeMX project file for STMicroelectronics STM32F7xx, STM32G4xx, and STM32H7xx-based boards.
- The ADC, PWM, GPIO Read, GPIO Write, and Hardware Interrupt blocks now support for Embedded Coder Support Package for STMicroelectronics STM32 Processors.
- The support package now also includes support for Monitor and Tune (External mode) and PIL simulation over serial.

Performance

SIMD code for reduction operations

In R2022a, you can generate SIMD code for reduction operations by using the new configuration parameter **Optimize reductions**. The generated code uses the reduction operations from the instruction set that you specify by using the **Instruction set extensions** parameter.

You can generate SIMD code for these blocks:

- Sum
- Product
- Minimum
- Maximum

If you have MATLAB Coder™ you can generate SIMD code for these MATLAB operations:

- Sum
- Product
- Minimum
- Maximum
- Handwritten loops for the previous operations

Consider this model `sumElements` that has a Sum of Elements block and an input of size [1 42].



In R2021b, the `sumElements_step` function contained this code:

```

tmp = -0.0;
for (i = 0; i < 42; i++) {
    tmp += sumElements_U.In1[i];
}
sumEl_Y.Out1 = tmp;
  
```

In R2022a, when you specify the **Instruction set extensions** SSE2 and select the parameter **Optimize reductions**, the `sumElements_step` function contains this code:

```

__m128d tmp;
real_T tmp_0[2];
int32_T i;
tmp = _mm_set1_pd(0.0);
for (i = 0; i <= 42; i += 2) {
    tmp = _mm_add_pd(tmp, _mm_loadu_pd(&sumElements_U.In1[i]));
}

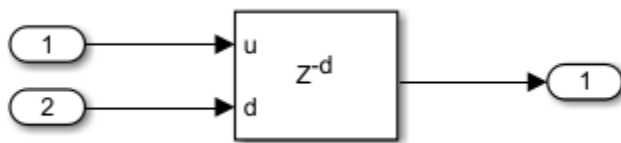
_mm_storeu_pd(&tmp_0[0], tmp);
sumElements_Y.Out1 = tmp_0[0] + tmp_0[1];
  
```

The function `_mm_add_pd` processes two 64-bit values in parallel. This increase in number of bits that process in parallel improves the execution speed of the code. For more information, see [Generate SIMD Code from Simulink Blocks](#) and [Generate SIMD Code for MATLAB Functions](#).

Code replacement for circular buffer index for Delay blocks

In R2022a, you can replace the calculation of the circular buffer index in the generated code for a Delay block with a target-specific implementation. Create a function replacement entry and set the **Function** to the new option `circularIndex`.

Consider this model `DelayModel` with a Delay block that has the parameter **Use circular buffer for state** selected.



In R2021b, the `DelayModel_step` function contained this code:

```

if (frameIdx < i) {
    DelayModel_Y.Out1[frameIdx] = DelayModel_DW.Delay_DSTATE[currIdx];
    DelayModel_Y.Out1[frameIdx + 32] = DelayModel_DW.Delay_DSTATE[currIdx + 100];
    DelayModel_Y.Out1[frameIdx + 64] = DelayModel_DW.Delay_DSTATE[currIdx + 200];
    currIdx++;
    if (currIdx >= 100) {
        currIdx = 0;
    }
}

```

In R2022a, when the model uses a code replacement library that has an entry for the `circularIndex` function, the `DelayModel_step` function calls the custom function `circindex_impl`:

```

if (frameIdx < i) {
    DelayModel_Y.Out1[frameIdx] = DelayModel_DW.Delay_DSTATE[currIdx];
    DelayModel_Y.Out1[frameIdx + 32] = DelayModel_DW.Delay_DSTATE[currIdx + 100];
    DelayModel_Y.Out1[frameIdx + 64] = DelayModel_DW.Delay_DSTATE[currIdx + 200];
    currIdx = circindex_impl(currIdx, 1, 100);
}

```

To calculate the circular buffer index more efficiently, you can specify a target-specific implementation. For more information, see [Buffer Index Calculation Code Replacement](#).

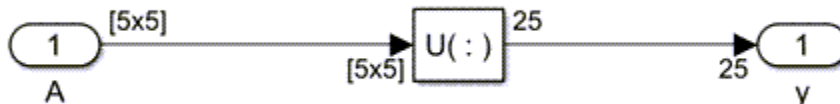
Code replacement for lookup tables by using index search algorithm parameter

In R2022a, you can replace code from Lookup table blocks by matching the setting for the block parameter **Begin index search using previous index result**. In a code replacement entry for a lookup function, specify the new algorithm parameter **Begin index search using previous index result** as `{off, on}`, `off`, or `on`. For more information, see [Lookup Table Function Code Replacement](#).

Code generation by inlining redundant function calls

Starting in R2022a, your generated code might be improved for functions by inlining the code. Prior to R2022a, the code generator was not able to generate inlined code in some cases. This optimization improves run time by eliminating the overhead of function calls in the generated code.

For example, consider the model `mRTWCGLateInlineReshapeRowmajor`.



In R2021b, the code generator produced this code:

```

static void m_reshapeRowMajorDataColumnWise(const real_T tmp[25], int32_T tmp_0,
real_T tmp_1[25])
{
    int32_T i;

    /* Reshape: '<Root>/Reshape' */
    tmp_2 = 0;
    tmp_3 = 0;
    for (i = 0; i < tmp_0; i++) {
        tmp_1[i] = tmp[5 * tmp_2 + tmp_3];
        tmp_2++;
        if (tmp_2 > 4) {
            tmp_2 = 0;
            tmp_3++;
        }
    }
}

void mRTWCGLateInlineReshapeRowmajor_step(void)
{
    /* Reshape: '<Root>/Reshape' incorporates:
    * Inport: '<Root>/A'
    * Outport: '<Root>/y'
    */
    m_reshapeRowMajorDataColumnWise(mRTWCGLateInlineReshapeRowmaj_U.A, 25,
    mRTWCGLateInlineReshapeRowmaj_Y.y);
}
  
```

In R2022a, the code generator produces this code:

```

void mRTWCGLateInlineReshapeRowmajor_step(void)
{
    int32_T i;

    /* Reshape: '<Root>/Reshape' */
    tmp = 0;
    tmp_0 = 0;
    for (i = 0; i < 25; i++) {
        /* Outport: '<Root>/y' incorporates:
        * Inport: '<Root>/A'
        */
        mRTWCGLateInlineReshapeRowmaj_Y.y[i] = mRTWCGLateInlineReshapeRowmaj_U.A[5 *
        tmp + tmp_0];
        tmp++;
        if (tmp > 4) {
            tmp = 0;
            tmp_0++;
        }
    }
}
}
  
```

In R2021b generated code, the function `mRTWCGLateInlineReshapeRowmajor_step` called another function `m_reshapeRowMajorDataColumnWise`. This redundant function call is removed from R2022a generated code. For more information, see [inlining](#).

Stack usage profiling for code generated from Simulink models

To determine the size of stack memory that is required to run generated code, you can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation that generates a stack usage profile. The simulation creates a code stack profiling report that shows minimum, average, and maximum memory demand. For each step, the simulation streams memory usage measurements to the Simulation Data Inspector, which enables you to analyze stack usage over time. You can use stack usage profiles to observe the effect of compiler optimization and data input. For more information, see [Stack Usage Profiling for Code Generated from Simulink Models](#).

Identification of performance bottlenecks in generated code

The code execution profiling report generated by a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation contains a new section **Execution Times in Percentages**. The section displays function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code. For more information, see [Code Execution Profiling Report Sections](#).

Code execution profiling for multiple Model blocks

In a top-model simulation, you can generate execution-time metrics for multiple Model blocks in SIL or PIL mode. Previously, code execution profiling was restricted to one Model block. For more information, see [View Code Execution Profiles for Multiple Model Blocks](#).

Verification

Unit-testing atomic subsystem code in AUTOSAR software component

In an AUTOSAR software component, perform unit tests on code generated from an atomic subsystem. Using the SIL/PIL Manager, you can:

- Test numeric equivalence between the subsystem and code generated from the subsystem.
- Analyze coverage for the generated code.
- Export an equivalence test to Simulink Test™.

For more information, see Test Atomic Subsystem Generated Code and Verify AUTOSAR C Code with SIL and PIL (AUTOSAR Blockset).

Functionality being removed or changed

SIL/PIL support for BullseyeCoverage will be removed

Still runs

Software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation support for BullseyeCoverage will be removed in a future release.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2021b

Version: 7.7

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Communication I/O information display during SIL or PIL execution

Use the MATLAB Coder configuration parameter **SIL/PIL Verbosity** (`SILPILVerbosity`) to specify the display of communication I/O information during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. For more information, see [Troubleshooting Host-Target Communication](#).

Visualization of task scheduling

If you enable code execution profiling for software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution, you can use the Simulation Data Inspector to visualize task scheduling and the order of function calls. At the end of the SIL or PIL execution, run the `schedule` function. For more information, see [Visualize Task Scheduling](#).

Reduction of violations for MISRA C++:2008 and AUTOSAR C++14 rules in generated code

In R2021b, the generated code has fewer violations of several rules in the required categories of MISRA C++:2008 and AUTOSAR C++14 coding standards. Some of these rules are:

- Dead code: MISRA C++:2008 Rule 0-1-2 (Polyspace Bug Finder), MISRA C++:2008 Rule 0-1-4 (Polyspace Bug Finder)
- Lexical conventions: AUTOSAR C++14 Rule A2-3-1 (Polyspace Bug Finder)
- Conditionals and loops: AUTOSAR C++14 Rule A5-16-1 (Polyspace Bug Finder), AUTOSAR C++14 Rule M6-4-1 (Polyspace Bug Finder), AUTOSAR C++14 Rule A6-5-2 (Polyspace Bug Finder)
- Initialization: AUTOSAR C++14 Rule A8-5-2 (Polyspace Bug Finder)
- Enumerations: AUTOSAR C++14 Rule A7-2-2 (Polyspace Bug Finder)
- Namespaces and classes: AUTOSAR C++14 Rule A2-10-4 (Polyspace Bug Finder), AUTOSAR C++14 Rule A9-3-1 (Polyspace Bug Finder), AUTOSAR C++14 Rule M7-3-1 (Polyspace Bug Finder), AUTOSAR C++14 Rule A7-1-9 (Polyspace Bug Finder)
- Other restrictions: AUTOSAR C++14 Rule A16-7-1 (Polyspace Bug Finder), AUTOSAR C++14 Rule A18-5-2 (Polyspace Bug Finder), MISRA C++:2008 Rule 4-10-2 (Polyspace Bug Finder)

For more information on how to generate code that has improved MISRA and AUTOSAR compliance, see [Generate C/C++ Code with Improved MISRA Compliance](#).

Model Architecture and Design

Built-in storage class for multi-instance data

Starting in R2021b, the built-in Simulink package of code definitions includes the new `MultiInstance` storage class. When you map data to the storage class, the generated code uses a structure to store multi-instance data and uses unstructured variables to store single-instance data. In your Embedded Coder dictionary, you can duplicate the `MultiInstance` storage class to make a copy of the storage class that you can modify. For more information, see [Flexible Storage Class for Different Model Hierarchy Contexts](#).

Symbolic dimension inputs for Bitwise Operator, Saturation, and Data Type Propagation blocks

Starting in R2021b, you can generate code for Bitwise Operator, Saturation, and Data Type Propagation blocks that have symbolic dimensions as inputs. Prior to R2021b, generating code for these blocks that had symbolic dimensions as inputs was not supported.

For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#).

Code Interface Configuration and Integration

Storage class with pointer data access in Embedded Coder Dictionary

Previously, to configure generated code to access data elements of a model by using a pointer, you had to apply the built-in storage class `ImportedExternPointer` to the element or create your own storage class that had pointer access by using the Custom Storage Class Designer. Starting in R2021b, you can create a storage class that has pointer access by using the Embedded Coder Dictionary.

In the Embedded Coder Dictionary, create a storage class and set **Data scope** to `Imported`. Then set **Data access** to `Pointer`. When you map a modeling element to this storage class, the generated code reads to and writes from that data element by using a pointer.

For more information, see [Embedded Coder Dictionary](#).

Unstructured Embedded Coder Dictionary storage class application to model reference root I/O

Previously, when you applied a new storage class from the Embedded Coder Dictionary to the input or output elements of a referenced model, the code generator reported an error. Starting in R2021b, when you create an unstructured storage class by using the Embedded Coder Dictionary, you can apply that new storage class to the root input and output elements of a referenced model.

For more information, see [Embedded Coder Dictionary](#).

Embedded Coder Dictionary storage class application to signals and parameters with symbolic dimensions

Previously, when you applied a storage class that you created in the Embedded Coder Dictionary to a signal or parameter that had dimension information specified as a symbol, the code generator reported an error. Starting in R2021b, you can apply an Embedded Coder Dictionary storage class to a signal or parameter that has symbolic dimensions when **Data initialization** for the storage class is not `Static`. The generated code preserves the symbolic dimensions.

For more information, see [Embedded Coder Dictionary](#).

Changes to model hierarchy requirements

Starting in R2021b, the code generator allows a model reference hierarchy to have different specifications for these model configuration parameters:

- **Support: variable-size signals** (`SupportVariableSizeSignals`)
- **Ignore test point signals** (`IgnoreTestpoints`)

The code generator allows a single-instance model reference hierarchy to have different specifications for memory sections for these categories of data defaults in the Code Mappings editor:

- Imports

- Outports
- Signals, states, and internal data
- Shared local data stores
- Constants

For information about the model configuration parameter available for Simulink Coder, see “Changes to model hierarchy requirements”.

For more information, see Set Configuration Parameters for Code Generation of Model Hierarchies.

Calibration file customization

Starting in R2021b, you can customize the ASAP2(a2l) file. The **Code Mappings Editor - C** enables you to customize the calibration properties of measurement and characteristic objects. For example, you can set the properties **Calibration Access** and add a **Display Identifier** by using the Code Mappings editor. For more information, see Configure Model Data Elements for ASAP2 File Generation.


You can group the measurements and characteristic objects in the ASAP2(a2l) file based on the properties of the data elements. For more information, see Customize Generated ASAP2 File.

TLC code storage classes in default mapping

In R2021b, custom storage classes that use TLC code are available in the default mapping. Previously, the default mapping did not support storage classes that had **Type** set to **Other**, which is required for TLC code. In R2021b, the default mapping supports storage classes that have **Type** set to **Other**. For more information, see Finely Control Data Representation by Writing TLC Code for a Storage Class.

Configure additional properties from the Code Mappings editor

Starting in R2021b, you can now configure additional code mapping properties from within the Code Mappings editor. These properties were previously accessible only in the Property Inspector.

To configure the properties, click the  icon in the row containing the element you want to configure.

The screenshot shows the MATLAB/Simulink Code Mappings editor for a C code generation project. The main workspace displays a Simulink block diagram with inputs 'In1' and 'In2'. 'In1' is connected to 'UPPER Constant1' and 'LOWER Constant2', which feed into 'RelOp1' and 'RelOp2' respectively. These outputs are combined in an 'OR LogOp' block, which then feeds into 'Data Store Write' and 'Data Store Memory' blocks. 'In2' is connected to a '1-D T(u)' block, which feeds into a gain block 'K1'.

The 'Code Mappings - C' panel at the bottom shows a hierarchical view of the mappings:

Source	Storage Class
In1	ImportedExtern
In2	Model default: ImportedExternPointer
In3	Model default: ImportedExternPointer
In4	Model default: ImportedExternPointer

A property inspector for the 'In1' mapping is open, showing the following settings:

- Identifier: input1
- Measurement: Export
- BitMask:
- CalibrationAccess: NoCalibration
- CompuMethodName:
- DisplayIdentifier:
- Format:

View In Bus Element and Out Bus Element blocks in a hierarchy in the Code Mappings editor

Beginning in R2021b, the Code Mappings editor displays data related to In Bus Element and Out Bus Element blocks in a hierarchical view. In previous releases, this data displayed as a flat list in the Code Mappings editor.

Configuring C/C++ function prototypes for subsystems not recommended

Editing the C/C++ function prototype configuration for models already configured for subsystem build, or configuring function prototypes for new subsystems using the `RTW.configSubsystemBuild` function or the associated UI now throws a warning.

To configure the function prototypes for a subsystem, convert the subsystem to a Model block. For models configured for C code generation, the conversion of the subsystem to a Model block migrates the model to use code mappings. See [Copy Code Mappings When Converting Subsystems to Referenced Models](#). For models configured for C++ code generation, you must manually configure the code mapping settings on the converted model.

To verify that a subsystem can be converted to a Model block, use the function `Simulink.SubSystem.convertToModelReference`.

Reusable storage class in Code Mappings editor

Prior to R2021a, you could specify buffer reuse on root-level inputs, data stores, signals, and states by associating the signals with a `Simulink.Signal` object and setting the **Storage Class** property to `Reusable`. The `Simulink.Signal` object was required to be defined outside the model, either in the base workspace or in a Simulink data dictionary.

Starting in R2021b, for individual root-level inputs, data stores, signals, and states, you can specify buffer reuse on them by setting **Storage Class** to `Reusable` in the Code Mappings editor. This optimization decreases data copies and memory consumption and increases code execution speed. The code generator does not require that the data element resolves to a `Simulink.Signal` object.

For reused data elements, you can specify the same value for the **Identifier** property. If you do not specify a value for the **Identifier**, the code generator uses the same signal label to name the reusable signal in the generated code. The code generator does not reuse signals if:

- The signals have the same labels but different identifiers.
- The signals have the same identifier but different values for these properties: **DataScope**, **HeaderFile**, **DefinitionFile**, and **Owner**. If there is a mismatch of values in any of the properties, the code generator stops and produces an error if the model configuration parameter **Detect non-reused custom storage classes** is set to error.

For more information, see [Specify Buffer Reuse for Signals in a Path](#).

Generated C++ model class name can be the model name

In R2021b, you can customize the generated model class name as the name of the model. Previously, the class name in the generated code used a default class name of the form `modelModelClass` or a custom name that was different from the name of the model. For more information on how to customize a model class name, see [Interactively Configure C++ Interface](#).

Code Generation

Accessibility of step entry-point functions generated for models designed for multitasking and concurrency streamlined

Prior to R2021b, for models configured for multitasking (**Treat each discrete rate as a separate task** is selected) or concurrency (**Allow task to execute concurrently on target** is selected), the code generator placed a `model_step` wrapper function, which served as a dispatcher, in generated algorithmic code files `model.c` or `model.cpp` and `model.h`. The wrapper function uses a switch statement to select the `model_stepN` function to call during run time. For multitasking models, you could suppress generation of the wrapper function by setting the TLC variable `RateBasedStepFcn` to 1.

Starting in R2021b, by default, the code generator streamlines accessibility and improves performance of step entry-point functions generated for models designed for multitasking and concurrent execution. The code generator produces a step entry-point function for each rate. The Code Interface Report lists the individual functions. A `main` program can call each of the entry-point functions directly. This change does not apply to models configured to use the classic call interface.

For existing application code that depends on the wrapper function, the code generator places the wrapper function in these generated files:

- For models configured for multitasking and have parameter **Generate an example main program** cleared - `rtmodel.c` or `rtmodel.cpp` and `rtmodel.h`
- For models configured for concurrent execution - `ert_main.c` or `ert_main.cpp`

For more information, see [Manage Build Process Files](#), [Analyze the Generated Code Interface](#), and [Configure C Code Generation for Model Entry-Point Functions](#).

Compatibility Considerations

In a future release, the code generator will stop generating the wrapper function. Update application code to call the rate-specific entry-point functions directly.

If your application code depends on use of the wrapper function, you can use these options temporarily:

- To use the generated static `main` program on bare board target hardware (**Generate an example main program** is selected and **Target operating system** is set to `BareBoardExample`), you can update the `#include` statement in the `main` program to specify `rtmodel.h` instead of `model.h`.
- To use a custom `main` program on a native threads target operating system (**Generate an example main program** is selected and **Target operating system** is set to `NativeThreadsExample`), you can do one of the following:
 - Generate the wrapper function in your custom `main` program.
 - Copy the wrapper function from the generated example native threads `main` program and paste the function into your application `main` program.

For more information, see [Deploy Generated Standalone Executable Programs To Target Hardware](#).

Code view for MATLAB Function block

Starting in R2021b, when you open a MATLAB Function block, the MATLAB Function Block Editor opens in the same Simulink window as the parent model of the MATLAB Function block. When you generate code from a MATLAB Function block, you can view the code alongside the function by using the Code view. The Code view enables you to trace between your generated code and your MATLAB function code in the same window as your model. For more information, see [Verify Generated Code by Using Code Tracing](#).

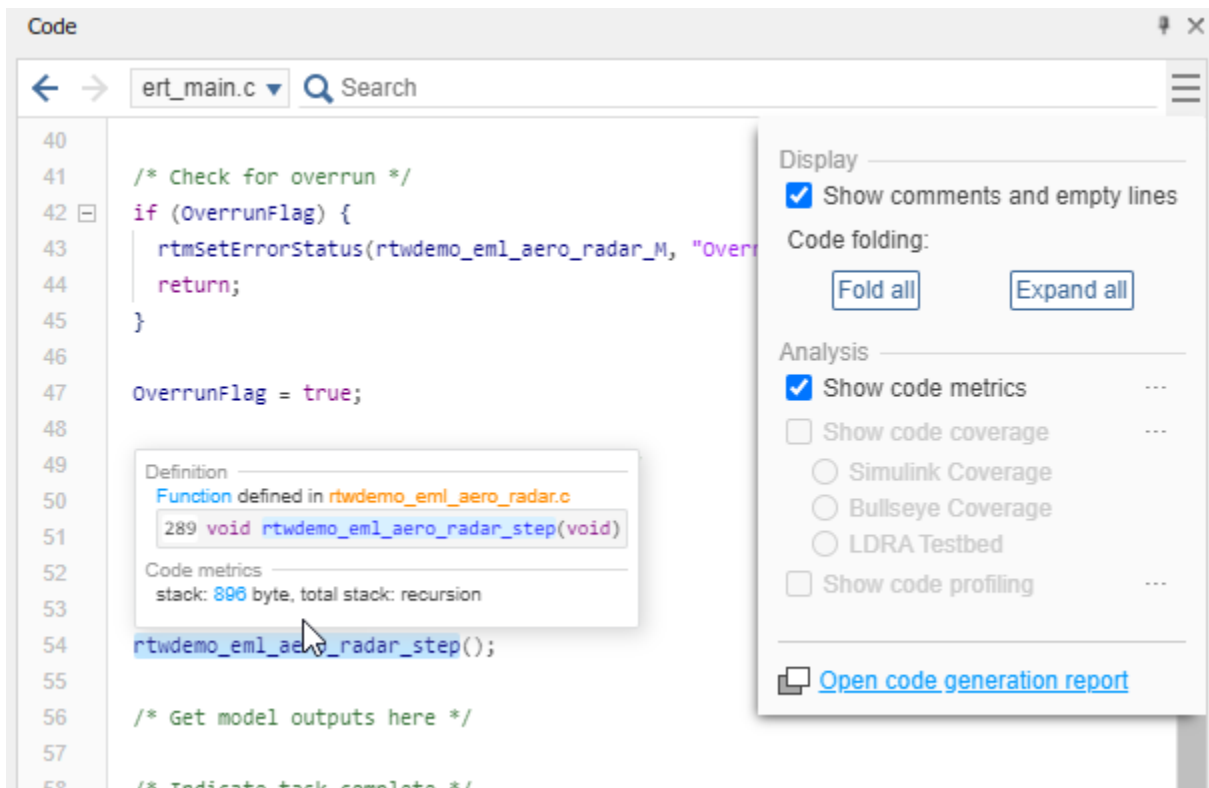
The Code view opens by default when you generate code from your model. To open the Code view manually, on the **C Code** tab, click **View Code**.

The screenshot shows the MATLAB Function Block Editor interface. The top toolbar includes tabs for SIMULATION, DEBUG, MODELING, APPS, FUNCTION, and C CODE. The C CODE tab is active, and the 'View Code' button is highlighted. The main workspace is split into two panes:

- Left Pane (MATLAB Code):** Shows the MATLAB function code for 'RadarTracker'. Key sections include:
 - Initialization of persistent variables `P` and `xhat`.
 - Computation of `Phi`, `Q`, and `R`.
 - Propagation of the covariance matrix `P` and track estimate `xhat`.
 - Computation of observation estimates (`Rangehat`, `Bearinghat`).
 - Computation of the observation vector `yhat` and linearized measurement matrix `M`.
 - Computation of the residual `residual`.
 - Computation of the Kalman Gain `W`.
 - Update of the estimate `xhat`.
- Right Pane (C Code):** Shows the generated C code for 'rtwdemo_eml_aero_radar.c'. It mirrors the MATLAB code with comments and mathematical operations. Key sections include:
 - Initialization of `M_tmp`, `b_idx_1`, and `b_idx_2`.
 - Computation of `Rangehat` and `Bearinghat`.
 - Computation of `yhat` and `M` for each time step.
 - Computation of `residual`.
 - Computation of `W` and update of `xhat`.

The status bar at the bottom indicates 'Ready', 'View diagnostics', '100%', and 'ode5'.

If you configure your model to generate code metrics, code coverage, or code profiling data, you can view the results in the Code view.



Enhanced code to reduce MISRA C:2012 Rule 10.3 and Directive 4.1 violations

Starting in R2021b, Embedded Coder produces code that reduces violations of the MISRA C:2012 Rule 10.3 and Directive 4.1.

For more information, see MISRA C:2012 Rule 10.3 (Polyspace Code Prover) and MISRA C:2012 Dir 4.1 (Polyspace Code Prover).

Changes to generated C++ header files

Due to infrastructural improvements, there might be minor, nonfunctional differences in headers generated for C++ classes. For more information about generated header files, see [Manage Build Process Files](#).

const member functions for C++ class interface

Starting in R2021b, the code generator emits a member functions as `const` when both of these conditions are true:

- The function does not modify a class member variables.
- The function does not call a non-const functions.

The code generator does not emit a `const` member function if the function:

- Calls a TLC function through fully inlined S-functions
- Is `getRTM()`
- Is `model_derivatives`

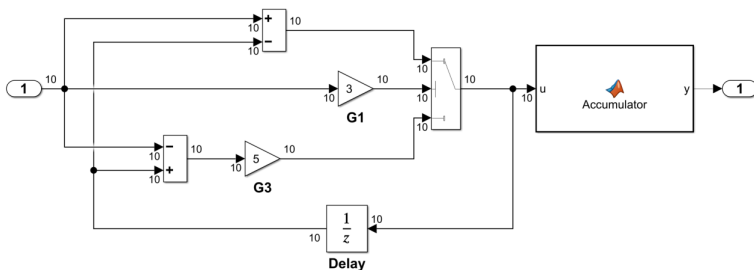
Generating member functions as `const` reduces MISRA C++ 2008 Rule 9-3-3 violations. For more information, see MISRA C++:2008 Rule 9-3-3 (Polyspace Bug Finder).

Minimized variable visibility for C++ code

Starting in R2021b, generated C++ code contains variable declarations that have minimized block scope. Minimized scope of variable declarations increases the likelihood of generating C++ code that is compliant with the Rule 3-4-1 of the MISRA C++:2008 guidelines. This optimization is applicable to these statements:

- `if`
- `for`
- `while`
- `switch`

Consider the model `mMinimizeVariableScopeBasic`.



In R2021a, the code generator produced this code:

```
void rtwdemo_forloopModelClass::step()
{
    int32_T k;
    mMinimizeVariableScopeBasic_Y.Out1 = 0.0;
    for (k = 0; k < 10; k++) {
        ...
    }
}
```

The variable `k` was declared outside the scope of the `for` loop where it was used.

In R2021b, the code generator produces this code:

```
void rtwdemo_forloopModelClass::step()
{
    mMinimizeVariableScopeBasic_Y.Out1 = 0.0;
    for (int32_T k = 0; k < 10; k++) {
        ...
    }
}
```

The variable `k` is declared and initialized within the scope of the `for` loop where it is used.

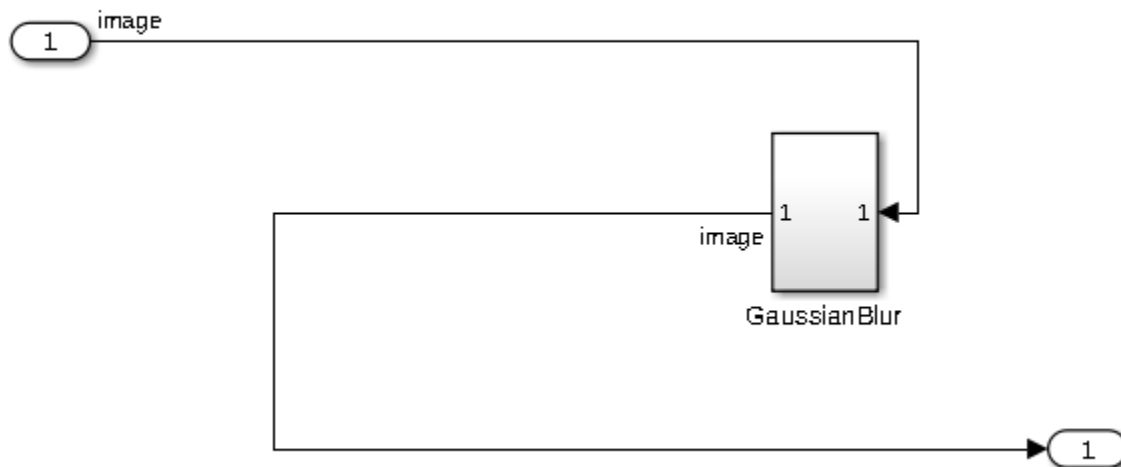
When you set the parameter value of `AdvancedOptControl` to `-SLCI` for compatibility with code inspection, the code generator does not generate minimally scoped variables.

For more information, see MISRA C++:2008 Rule 3-4-1 (Polyspace Bug Finder).

Image data by using OpenCV class `cv::Mat`

Computer Vision Toolbox™ Interface for OpenCV in Simulink enables you to specify an image as a `Simulink.ImageType` (Computer Vision Toolbox) data type and generate code for your model. For ERT-targets, select the new model configuration parameter **Data Type Replacement > Implement images using OpenCV Mat class** to generate production C++ code where images are represented by using the OpenCV class `cv::Mat` instead of the C++ class `images::datatypes::Image` implemented by The MathWorks®. By default, this parameter is not selected.

For example, consider this model:



If you do not select **Implement images using `cv::Mat`** parameter, the code generator produces this code for root-level I/O and converted block I/O:

```
/* Block signals (default storage) */
struct B_CVCodegen_T {
  cv::Mat ToOpenCV;          /* '<Root>/ToOpenCV' */
  cv::Mat GaussianBlur;     /* '<Root>/GaussianBlur' */
};

/* External inputs (root inport signals with default storage) */
struct ExtU_CVCodegen_T {
  images::datatypes::Image In1; /* '<Root>/In1' */
};

/* External outputs (root outports fed by signals with default storage) */
struct ExtY_CVCodegen_T {
  images::datatypes::Image Out1; /* '<Root>/Out1' */
};
```

```

/* Model step function */
void mCVCodegenModelClass::step()
{
  /* S-Function (FromOpenCV): '<Root>/FromOpenCV' incorporates:
   * Output: '<Root>/Out1'
   */
  {
    uint8_t *y0 = static_cast<uint8_t*>(imageGetDataFcn (&mCVCodegen_Y.Out1));
    mCVCodegen_B.GaussianBlur = cv::Mat(384, 512, CV_MAKETYPE(CV_8U, 3), y0);

    /*S-Function Block: <Root>/ToOpenCV */
    {
      const uint8_t *u0 = static_cast<uint8_t*>(imageGetDataFcn (&mCVCodegen_U.In1));
      mCVCodegen_B.ToOpenCV = cv::Mat(384, 512, CV_MAKETYPE(CV_8U, 3), u0);

      /* S-Function Block: '<Root>/GaussianBlur' */
      opencv::GaussianBlur(&mCVCodegen_B.GaussianBlur, &mCVCodegen_B.ToOpenCV);
    }
  }
}

```

There are two buffers of the `images::datatypes::Image` class and two buffers of the class `cv::Mat` class. Shallow copies convert data from the `images::datatypes::Image` class to `cv::Mat` class.

If you select **Implement images using cv::Mat**, the code generator produces this code:

```

/* External inputs (root inport signals with default storage) */
struct ExtU_CVCodegen_T {
  cv::Mat In1; /* '<Root>/In1' */
};

/* External outputs (root outports fed by signals with default storage) */
struct ExtY_CVCodegen_T {
  cv::Mat Out1; /* '<Root>/Out1' */
};

/* Model step function */
void mCVCodegenModelClass::step()
{
  /* S-Function Block: '<Root>/GaussianBlur' */
  opencv::GaussianBlur(mCVCodegen_Y.Out1, &mCVCodegen_U.In1);
}

```

Output buffers for the block and the shallow copies are eliminated because the root-level I/O is represented by `cv::Mat` class.

For more information about `Simulink.ImageType`, see the “Computer Vision Toolbox Interface for OpenCV in Simulink: Specify image data type in Simulink model” (Computer Vision Toolbox) release note.

Shared types and parameters storage in same header file

Starting in R2021b, you can store shared types and parameters in the same header file. You can store the following types in the same file as `Simulink.Parameter`.

- `Simulink.Alias`
- `Simulink.NumericType`
- `Simulink.LookupTable`

In this example, the code generator stores an `Alias` and a `Parameter` in the same header file in the shared utilities folder.

```

#ifndef RTW_HEADER_myHeader_h_
#define RTW_HEADER_myHeader_h_

```

```
#include "rtwtypes.h"

typedef real_T myAlias;
typedef creal_T cmyAlias;

// Exported data declaration
// Declaration for custom storage class: ExportToFile
extern real_T myParam;

#endif                                     // RTW_Header_myHeader_h
```

Prior to R2021b, these combinations caused errors when you built your code. For more information, see [Choose Storage Class for Controlling Data Representation in Generated Code](#).

Bidirectional traceability in Code view by default

Starting in R2021b, the Code view provides bidirectional traceability between your model and the generated code by default. Previously, you had to select the configuration parameters **Code-to-model** or **Model-to-code**. To enable code tracing in the code generation report, you still select these parameters. For more information, see [Trace Simulink Model Elements in Generated Code](#).

Deployment

New TLC variable `OverrideSampleERTMain` for disabling generation of example main program

When developing a custom system target file, you can override the default code generator behavior in the Target Language Compiler (TLC) for creating an example main program (`ert_main.c` or `ert_main.cpp`). For example, apply the override if you already have or want to generate your own main program module. Starting in R2021b, to apply this override, use the new TLC variable `OverrideSampleERTMain`.

For example, if you want to generate or write your own main program, suppress generation of the default example main program by including this line of code in your TLC setup script:

```
%assign CompiledModel.OverrideSampleERTMain = TLC_TRUE
```

Prior to R2021b, to override the generation of an example main program, you used the TLC variable `GenerateSampleERTMain`. This variable still works. The code generator produces slightly different results depending on whether you set `OverrideSampleERTMain` to `TLC_TRUE` or set `GenerateSampleERTMain` to `TLC_FALSE`.

For more information, see [Generate Source and Header Files with a Custom File Processing \(CFP\) Template](#).

Texas Instruments C2000: Code generation support for Configurable Logic Block (CLB) and CLB X-Bar in Embedded Coder Support Package for Texas Instruments C2000 Processors

The Embedded Coder Support Package for Texas Instruments C2000 Processors now provides code generation support for the CLB Crossbar (CLB X-BAR) and provides the option to configure CLB and integrate generated CLB files from the CLB tool for the F2838x(C28x), F28002x, and F28004x processors.

Texas Instruments C2000: External Mode Simulation Using XCP on CAN Interface

In the Embedded Coder Support Package for Texas Instruments C2000 Processors, you can now configure a model for simulating in the external mode to perform signal logging and parameter tuning using XCP on CAN.

Support for STMicroelectronics STM32F4xx-based Boards

- You can use the Embedded Coder Support Package for STMicroelectronics Discovery Boards to generate and build code using an STM32CubeMX project file for STMicroelectronics STM32F4xx-based boards.
- ADC, PWM, GPIO Read, GPIO Write, and Hardware Interrupt blocks are supported for STMicroelectronics STM32F4xx-based boards.
- External mode over serial and PIL simulation is also supported.

Performance

Generation of SIMD code by using new configuration parameter

In R2021b, when you generate code for Intel or AMD hardware, you can generate single instruction, multiple data (SIMD) code by specifying your SIMD instruction sets by using the new configuration parameter **Leverage target hardware instruction set extensions**.

For new models that use the supported target hardware, the parameter is set to SSE2 by default. The generated code uses SIMD intrinsics. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel and AMD platforms.

Previously, to generate SIMD code, you used code replacement libraries. In R2021b, use the new parameter to select one of these instruction sets:

- SSE
- SSE2
- SSE4.1
- AVX
- AVX2
- FMA
- AVX512F

When you generate code, Embedded Coder loads your specified instruction set and the instruction sets that it requires. The replacements appear in the generated code for blocks that meet the supported conditions for SIMD. For more information, see [Generate SIMD Code from Simulink Blocks](#).

You can no longer specify the SIMD instructions for the **Code replacement libraries** parameter because the SIMD instruction sets are now available by using the **Leverage target hardware instruction set extensions** parameter. The SIMD code replacement libraries include:

- Intel SSE
- Intel AVX
- Intel AVX-512

Models that you saved in a previous version are not changed and still use these code replacement libraries.

Image Processing Toolbox functions enhanced with multithreading and algorithm improvements

Starting in R2021b, if you use a compiler that supports the Open Multiprocessing (OpenMP) application interface, you can generate multithreaded C/C++ functions for some Image Processing Toolbox functions that are included in MATLAB code or in Simulink models that have MATLAB Function blocks or MATLAB System blocks. Some of the Image Processing Toolbox functions have algorithm improvements in the generated code. These enhancements improve the function execution speed.

To enable multithreading, select the model configuration parameters **Specify custom optimizations** and **Generate parallel for loops**.

The optimized functions that have multithreading capabilities are:

- `hsv2rgb`
- `imadjust`
- `imfilter`
- `label2rgb`

The functions that have algorithm improvements are:

- `imfill`
- `imreconstruct`
- `medfilt2`

In R2021a, the code generator produced this C code snippet for a MATLAB function containing an image processing function `imadjust`:

```
...
for (k = 0; k < 1310720; k++) {
    out_tmp = (k % 1024 + ((k / 1024) << 10)) + 1310720 * p;
    out[out_tmp] =
        rt_powd_snf((fmax(lIn, fmin(hIn, varargin_1[out_tmp])) - lIn) /
                    (hIn - lIn), g) * (hOut - lOut) + lOut;
}
...
```

The loop executed sequentially.

In R2021b, the code generator produces this code snippet:

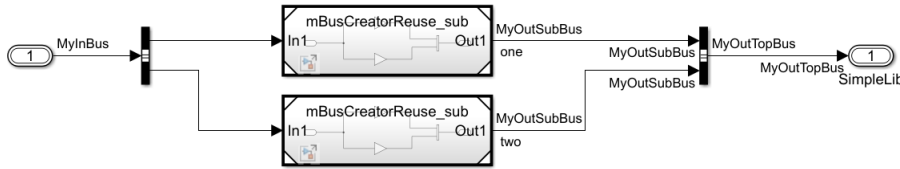
```
...
#pragma omp parallel for num_threads(omp_get_max_threads())
    for (i = 0; i < 1310720; i++) {
        out[i] = rt_powd_snf((fmax(lIn, fmin(hIn, img[i])) - lIn) / (hIn - lIn), g) *
            (hOut - lOut) + lOut;
    }
...
```

The generated code has the pragma for OpenMP (Open Multiprocessing) before the body of the loop. OpenMP enables shared-memory and multicore platforms to execute loops in parallel. This parallel execution improves the execution speed of the generated code. For more information, see [Speed Up for-Loop Implementation in Code Generated by Using parfor and Algorithm Acceleration Using Parallel for-Loops \(parfor\)](#).

Reduced data copies for models that have Bus Creator blocks

Prior to R2021b, the generated code contained redundant data copies for models that have Bus Creator blocks, which combined the outputs of reusable subsystems and model references into buses. Starting in R2021b, if the top model and referenced models do not have function prototype control specifications, the code generator generates optimized code by eliminating the redundant data copies. Eliminating the redundant data copies reduces RAM and ROM consumption and improves execution speed.

Consider the model `mBusCreatorReuse`, which has two instances of the referenced model `mBusCreatorReuse_sub` connected to a Bus Creator block.



In R2021a, the code generator produced this code:

```
/* Model step function */
void mBusCreatorReuse_step(void)
{
    /* local block i/o variables */
    MyOutSubBus rtb_one;
    MyOutSubBus rtb_two;

    /* ModelReference: '<Root>/Model' incorporates:
     * Inport: '<Root>/In1'
     */
    mBusCreatorReuse_sub(&rtU.In1.one, &rtb_one);

    /* ModelReference: '<Root>/Model1' incorporates:
     * Inport: '<Root>/In1'
     */
    mBusCreatorReuse_sub(&rtU.In1.two, &rtb_two);

    /* Output: '<Root>/SimpleLib' incorporates:
     * BusCreator: '<Root>/Bus Creator'
     */
    rtY.SimpleLib.one = rtb_one;
    rtY.SimpleLib.two = rtb_two;
}

```

The generated code contained unnecessary data copies to the local variables `rtb_one` and `rtb_two`.

In R2021b, the code generator generates this code:

```
/* Model step function */
void mBusCreatorReuse_step(void)
{
    /* ModelReference: '<Root>/Model' incorporates:
     * Inport: '<Root>/In1'
     */
    mBusCreatorReuse_sub(&rtU.In1.one, &rtY.SimpleLib.one);

    /* ModelReference: '<Root>/Model1' incorporates:
     * Inport: '<Root>/In1'
     */
    mBusCreatorReuse_sub(&rtU.In1.two, &rtY.SimpleLib.two);
}

```

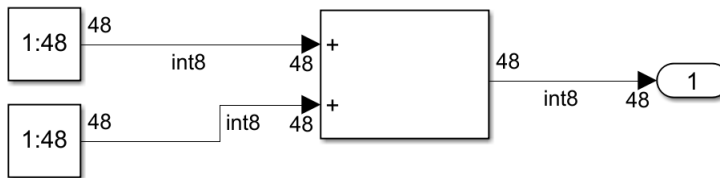
The generated code does not contain the local variables `rtb_one`, `rtb_two`, and the data copies. Now, the code generator stores the input elements of the Bus Creator block into the root output structure fields `rtY.SimpleLib.one` and `rtY.SimpleLib.two` directly.

SIMD optimization for more integer data types

Prior to R2021b, the generated code for Simulink models contained SIMD optimizations for 32- and 64- bit integer data types. Starting in R2021b, for Intel SSE® or AVX® processors, the generated code for models contains SIMD optimizations for 8- and 16- bit integer data types.

To enable this optimization, set the model configuration parameter **Leverage target hardware instruction set** to SSE2, SSE4.1, or AVX2.

Consider the model mAddInt8, which has an Add block.



In R2021a, the code generator produced this C code by using the Intel SSE (Windows) code replacement library:

```

/* Model step function */
void mAddInt8_step(void)
{
    int32_T i;

    /* Outport: '<Root>/Outport' incorporates:
     * Constant: '<Root>/Constant1'
     * Constant: '<Root>/Constant2'
     * Sum: '<Root>/Add1'
     */
    for (i = 0; i < 48; i++) {
        mAddInt8_Y.Outport[i] = (int8_T)(mAddInt8_P.Constant1_Value[i] +
            mAddInt8_P.Constant2_Value[i]);
    }
}
  
```

The loop incremented by one for the variable *i*.

In R2021b, the code generator produces this SIMD vectorized code when you set **Leverage target hardware instruction set** to SSE2:

```

/* Model step function */
void mAddInt8_step(void)
{
    int32_T i;
    for (i = 0; i <= 32; i += 16) {
        /* Outport: '<Root>/Out1' incorporates:
         * Constant: '<Root>/Constant'
         * Constant: '<Root>/Constant1'
         */
        _mm_storeu_si128((__m128i *)&mAddInt8_Y.Out1[i], _mm_add_epi8(
            _mm_loadu_si128((__m128i *)&mAddInt8_P.Constant_Value[i]),
            _mm_loadu_si128((__m128i *)&mAddInt8_P.Constant1_Value[i]));
    }
}
  
```

The loop increments by 16 because the input data type is `int8`. Incrementing by 16 instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is

`int16`, the loop increments by 8. This optimization increases the execution speed of the generated code. For more information, see [Generate SIMD Code from Simulink Blocks](#).

Root outputport initialization code performance improvements

Starting in R2021b, generated code contains optimizations for root outputport initialization. These optimizations result in smaller object files, reduced ROM consumption, and faster run-time performance.

Prior to R2021b, initialization code for root outputports contained separate `for` loops of the same size:

```
/* external outputs */
{
    int32_T i;
    for (i = 0; i < 2350; i++) {
        mForLoopFused_Y.Out3[i] = -2;
    }
}
{
    int32_T i;
    for (i = 0; i < 2350; i++) {
        mForLoopFused_Y.Out4[i] = -2;
    }
}
```

Starting in R2021b, the code generator merges these `for` loops:

```
/* external outputs */
{
    int32_T i;
    for (i = 0; i < 2350; i++) {
        mForLoopFused_Y.Out1[i] = -2;
        mForLoopFused_Y.Out2[i] = -2;
    }
}
```

This merge results in smaller object files, reducing ROM consumption. The merged `for` loop is also faster at run-time.

Prior to R2021b, the generated code for models that had many zero constants contained a large structure that initialized each constant individually:

```
const busObj mStringInBusCGpatterns_rtZbusObj = {
    0.0, /* elem1 */
    "", /* stringElem */
    "" /* stringObjElem */
} ; /* busObj ground */

/* Model initialize function */
void mStringInBusCGpatterns_initialize(void)
{
    /* external outputs */
    (void) memset((void *)&mStringInBusCGpatterns_Y, 0,
        sizeof(ExtY_mStringInBusCGpatterns_T));
    mStringInBusCGpatterns_Y.Out3 = mStringInBusCGpatterns_rtZbusObj;
}
```

Starting in R2021b, the code generator initializes these constants in bulk by using the `memset` function:

```
/* Model initialize function */
void mStringInBusCGpatterns_initialize(void)
{
    /* external outputs */
    (void) memset((void *)&mStringInBusCGpatterns_Y, 0,
                 sizeof(ExtY_mStringInBusCGpatterns_T));
}
```

This initialization results in smaller object files, reducing ROM consumption.

Prior to R2021b, the code generator could use the `memset` function on the entire root output structure, but not on individual outputs:

```
const botBus mAoSIndirectMask_rtZbotBus = {
    {
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
        0.0 }
    ,
    0.0
};
/* external outputs */
(void) memset(&mAoSIndirectMask_Y.Out1[0], 0,
             2U*sizeof(botBus));
```

Starting in R2021b, the code generator can use the `memset` function on individual outputs:

```
/* external outputs */
(void)memset(&mAoSIndirectMask_Y, 0, sizeof(ExtY_mAoSIndirectMask_T));
```

Using the `memset` function on individual outputs results in smaller object files, reducing ROM consumption. For more information, see [Optimize Generated Code Using `memset` Function](#).

Readability improvement for root output initialization code

R2021b introduces readability improvements for root output initialization code. These improvements change the code syntactically but not semantically, resulting in root output initialization code that is more consistent with other types of generated code.

Prior to R2021b, when you specified an identifier naming rule, the code generator did not apply this rule to root outputs. In this example, R2021a ignores an identifier length rule when generating `mModelUsingBusDT_rtZbusTypeForMFile`:

```
const busTypeForMFile mModelUsingBusDT_rtZbusTypeForMFile = {
    0.0F,
    0.0
};
void mModelUsingBusDT_initialize(void)
{
    mModelUsingBusDT_Y.Out1 = mModelUsingBusDT_rtZbusTypeForMFile;
}
```

Starting in R2021b, the code generator applies the rule and truncates the identifier to `mModelUsingBusDT_rtZbusTypeForM`:

```
const busTypeForMFile mModelUsingBusDT_rtZbusTypeForM = {
    0.0F,
```

```
    0.0
};
void mModelUsingBusDT_initialize(void)
{
    mModelUsingBusDT_Y.Out1 = mModelUsingBusDT_rtZbusTypeForM;
}
```

For more information, see [Customize Generated Identifier Naming Rules](#).

Prior to R2021b, root output initialization code used the `*` and `.` operators to access elements of structures:

```
(*mrootioindividual_Y_Out1) = 0.0;
(*mrootioindividual_Y_Out2).re = 0.0;
(*mrootioindividual_Y_Out2).im = 0.0;
```

Starting in R2021b, the code uses the `->` operator:

```
*mrootioindividual_Y_Out1 = 0.0;
mrootioindividual_Y_Out2->re = 0.0;
mrootioindividual_Y_Out2->im = 0.0;
```

This is consistent with code generation for external inputs.

Optimize code by unrolling parallel for-loops

Starting in R2021b, you can specify a value for the model configuration parameter **Loop unrolling threshold** parameter value to automatically unroll parallel for-loops (`parfor`-loops).

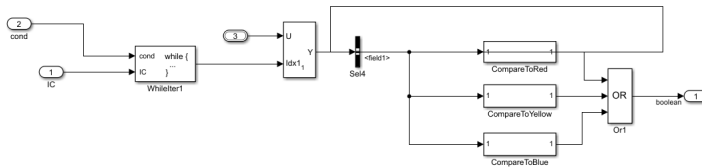
When the code generator unrolls a `parfor`-loop, it produces a copy of the loop body for each iteration. For a small number of loop iterations that perform some simple calculation, parallelization is inefficient as it introduces overheads, which includes time taken for thread creation, data synchronization between threads, and thread deletion. Unrolling the loops that have a large number of iterations can significantly increase code generation time and generate inefficient code.

The default value of the **Loop unrolling threshold** parameter is 5. By modifying the threshold, you can fine-tune loop unrolling. To modify the threshold, change the value of the parameter **Loop unrolling threshold**. For more information, see [Unroll Parallel for-Loop That Has Small Number of Iterations](#).

Improved common subexpression elimination

Prior to R2021b, for models that contained redundant subexpressions that were used to access the same `struct` fields, the generated code repeatedly executed the subexpressions and accessed the `struct` fields. Starting in R2021b, the generated code contains a temporary variable that holds the value of these subexpressions and eliminates accessing the same fields repeatedly. This optimization improves the execution speed of the generated code. The model configuration parameter **Eliminate superfluous local variables (expression folding)** enables this optimization.

Consider the model `mWhileBusAccess`.



In R2021a, the code generator produced this code:

```
void mWhileBusAccess_step(void)
{
    int32_T s1_iter;
    boolean_T loopCond;
    s1_iter = 1;
    loopCond = true;
    while (loopCond && ((uint32_T)s1_iter <= 300U)) {
        rtY.Out1 = ((rtU.busSignal1.field3[s1_iter - 1].field2.field1 == Red) ||
                    (rtU.busSignal1.field3[s1_iter - 1].field2.field1 == Yellow) ||
                    (rtU.busSignal1.field3[s1_iter - 1].field2.field1 == Blue));
        loopCond = (rtU.cond != 0U);
        s1_iter++;
    }
}
```

The generated code contained subexpressions to repeatedly access the same `struct` field because the same bus was attached to three distinct blocks.

In R2021b, the code generator produces this code:

```
void mWhileBusAccess_step(void)
{
    int32_T s1_iter;
    boolean_T loopCond;
    Colors Out1_tmp;
    s1_iter = 1;
    loopCond = true;
    while (loopCond && ((uint32_T)s1_iter <= 300U)) {
        Out1_tmp = rtU.busSignal1.field3[s1_iter - 1].field2.field1;
        rtY.Out1 = ((Out1_tmp == Red) || (Out1_tmp == Yellow) || (Out1_tmp == Blue));
        loopCond = (rtU.cond != 0U);
        s1_iter++;
    }
}
```

The generated code contains the temporary variable `Out1_tmp` for holding the value of the subexpressions that access the same `struct` field, thereby eliminating the redundancy. For more information, see [Eliminate superfluous local variables \(Expression folding\)](#).

Optimized SIMD code that performs fused multiply add operations

Starting in R2021b, if you use a processor that supports fused multiply-add (FMA) instructions, you can generate optimized SIMD code that performs fused multiply-add operations. The fused multiply-add operations are for sequential multiplication-addition arithmetic operations involving `single` and `double` data types. Fused multiply-add operations are performed in one step with a single rounding than performing a multiplication operation followed by an addition. Using this optimization improves the execution speed of the generated SIMD code.

To enable FMA optimization, set the model configuration parameter **Leverage target hardware instruction set** to `FMA`. For more information, see [Optimize SIMD Code by Performing Fused Multiply Add Operations](#).

Redundant data copies elimination by reusing S-function block buffers

Starting in R2021b, you can use the `LibBlockInputSignalBufferDstPort` function to generate code with fewer data copies for a model containing an S-function block that implements an in-place (that is, uses the same input and output variable) C function when one of these conditions is true:

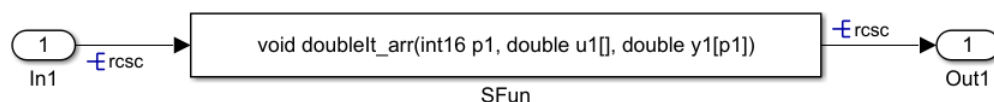
- The block has signals with buffer reuse specifications by using `Simulink.Signal` objects. For more information, see [Specify Buffer Reuse for Signals in a Path](#).
- The block has function prototype control specifications to use the same buffer for input ports and output ports. For more information, see [Configure Name and Arguments for Individual Step Functions](#).
- The block connects to a MATLAB Function block with in-place (that is, uses the same input and output variable) specification. For more information, see [Specify Buffer Reuse for MATLAB Function Blocks in a Path](#).
- The block connects to a Unit Delay block.
- The block has a bus data type as input and output.
- The model uses a Data Store Memory block for reading and writing S-function block input and output. For more information, see [Data Copy Reduction for Data Store Read and Data Store Write Blocks](#).

To reuse the input port of an S-function, in the S-function source code, set these flags:

- `ssSetInputPortOverWritable`: Specify that input ports can be overwritten by one of their output ports.
- `ssSetInputPortOptimOpts`: Declare an inport port as reusable local or global.
- `ssSetOutputPortOptimOpts`: Declare an output port as reusable local or global.
- `ssSetOutputPortOverwritesInputPort`: Specify which output port overwrites which input port.

To check if the input buffer of the S-function block is reused by the output port, add an if condition in the TLC file based on the return value of the `LibBlockInputSignalBufferDstPort` function.

For example, consider the model `mRCSC_RootInOut`. The model contains an S-function block that has signals with buffer reuse specifications.



The S-function block implements an in-place C function through the `doubleIt_arr` wrapper function.

```
void doubleIt_arr_inplace(int dim, double* inOutVal) {
    int i = 0;
    for (i = 0; i < dim; i++){
        inOutVal[i] *= 2;
    }
}
void doubleIt_arr(int dim, double* inVal, double* outVal)
{
```



```
// This check is required for Simulation Correctness because TLC code is ignored in Simulation
if (inVal != outVal){
    int i = 0;
    for (i=0;i<dim;i++){
        outVal[i] = inVal[i];
    }
    doubleIt_arr_inplace(dim, outVal);
}
```

To reuse the input buffer of an S-function block, set the reusable flags in the S-function source code. To check if the input buffer of the S-function block is reused by the output port, add an if condition in the TLC file as follows:

```
%if (LibBlockInputSignalBufferDstPort(0) == -1)
{
    for(int i = 0; i < %<p1_val>; i++)
    {
        (%<y1_ptr>)[i] = (%<u1_ptr>)[i];
    }
}
%endif
```

In R2021a, the code generator produced this code:

```
/* Model step function */
void mRCSC_RootInOut_step(void)
{
    /* S-Function (ex_sfundoubleit_arr): '<Root>/SFun' */
    {
        for (int i = 0; i < 2; i++) {
            ((&(rcsc[0]))) [i] = ((&(rcsc[0]))) [i];
        }
    }
    doubleIt_arr(2, (real_T*)(&(rcsc[0])), (&(rcsc[0])));
}
```

The generated code contained a **for-loop** and copy operation to reuse the S-function block input buffer for the output port because the `LibBlockInputSignalBufferDstPort` did not identify the reuse of the input buffer that had a reusable storage class specification.

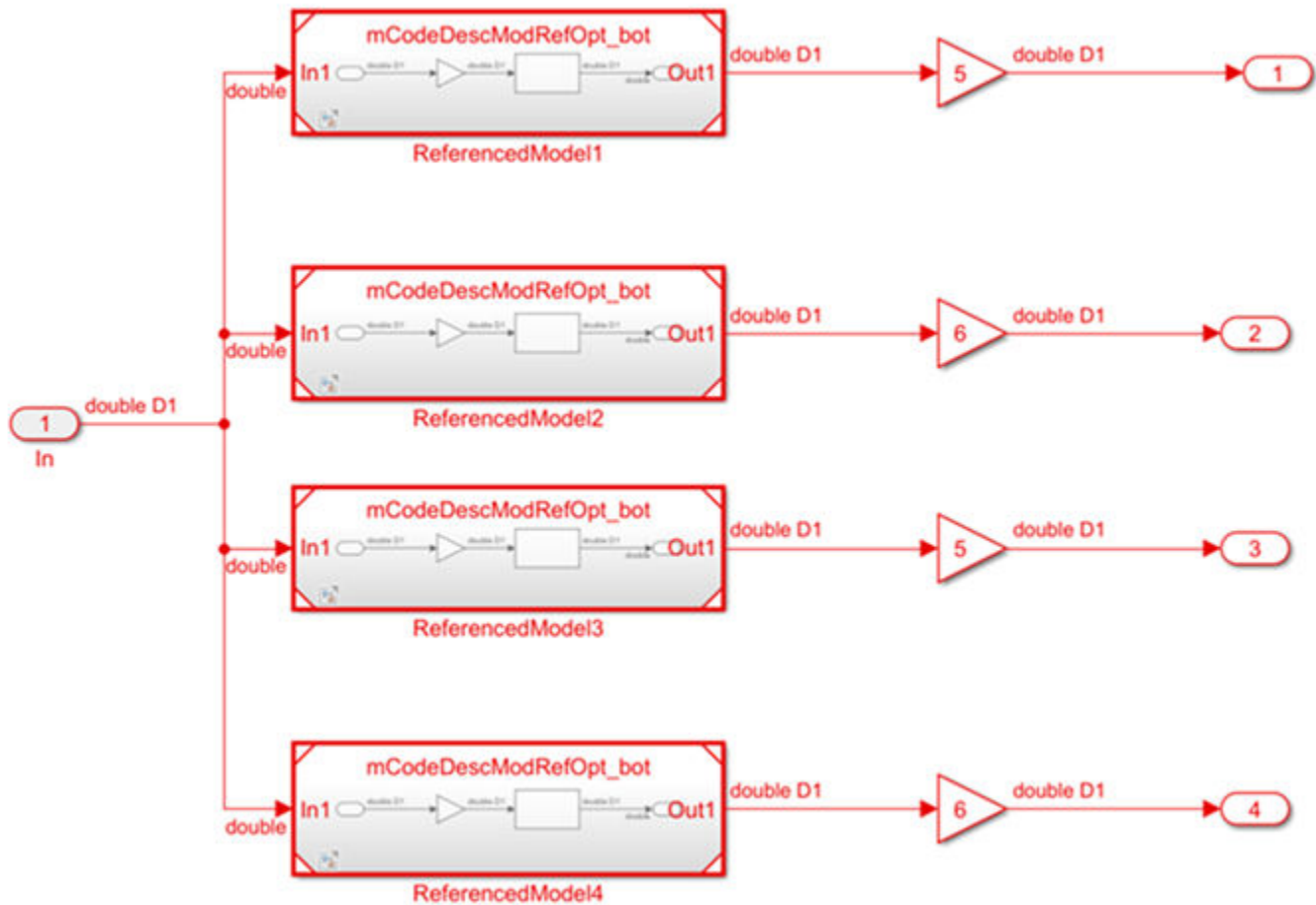
In R2021b, the code generator produces this code:

```
/* Model step function */
void mRCSC_RootInOut_step(void)
{
    /* S-Function (ex_sfundoubleit_arr): '<Root>/SFun' */
    doubleIt_arr(2, (real_T*)(&(rcsc[0])), (&(rcsc[0])));
}
```

The generated code does not contain the **for-loop** and redundant copy operation. The `LibBlockInputSignalBufferDstPort` function identifies that buffer reuse occurs, therefore the code generator directly reuses the input buffer for the output port. Reducing the redundant data copies reduces RAM and ROM consumption and improves execution speed. For more information, see [Advanced Functions and S-Functions That Specify Port Scope and Reusability](#).

Optimized code for models containing referenced models

Starting in R2021b, the generated code has fewer local variables for modeling patterns that have referenced models. The unnecessary local variables are eliminated, which improves the efficiency of generated code. For example, consider a model `mCodeDescModRefOpt_top` that has a referenced model `mCodeDescModRefOpt_bot`.



In R2021a, the code generator produced this code:

```
void mCodeDescModRefOpt_top_step(void)
{
    real_T rtb_ReferencedModel1;
    real_T rtb_ReferencedModel2;
    real_T rtb_ReferencedModel3;
    real_T rtb_ReferencedModel4;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel1);
    rtY.Out2 = 5.0 * rtb_ReferencedModel1;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel2);
    rtY.Out1 = 6.0 * rtb_ReferencedModel2;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel3);
    rtY.Out3 = 5.0 * rtb_ReferencedModel3;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel4);
    rtY.Out4 = 6.0 * rtb_ReferencedModel4;
}
```

In R2021b, the code generator produced this code:

```
void mCodeDescModRefOpt_top_step(void)
{
    real_T rtb_ReferencedModel1;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel1);
    rtY.Out2 = 5.0 * rtb_ReferencedModel1;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel2);
    rtY.Out1 = 6.0 * rtb_ReferencedModel2;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel3);
    rtY.Out3 = 5.0 * rtb_ReferencedModel3;
    mCodeDescModRefOpt_bot(rtU.In, &rtb_ReferencedModel4);
    rtY.Out4 = 6.0 * rtb_ReferencedModel4;
}
```

The generated code contains lesser local variables in R2021b.

For more information on model references, see [Generate Code for Model Reference Hierarchy](#).

Nonstatic data class member initialization of instance-specific parameters

Starting in R2021b, the code generator supports nonstatic data class member initialization in C++11 for instance-specific parameters. The instance-specific parameters must map to a class member. In the Property Inspector, set **Data Access** to **Direct**. In the Code Mappings editor set **Data Visibility** to **private**. For more information, see [Interactively Configure C++ Interface and Code Mappings - C++ Editor](#).

In R2021a, the code generator defined a structure containing the values of the instance-specific parameters. It then passed that structure to the model class constructor. In R2021b, the code generator still behaves this way when generating C++03.

```
// instance parameters
untitledModelClass::InstP_mCPPInstP_T mCPPInstP_InstP_init = {
    // Variable: K
    // Referenced by: '<Root>/<Gain>'

    3.0
};

// Constructor
untitledModelClass::untitledModelClass() :
    mCPPInstP_InstP(mCPPInstP_InstP_init),
    mCPPInstP_U(),
    mCPPInstP_Y(),
    mCPPInstP_M()
{}
```

In R2021b, when generating C++11, the code generator directly specifies the default class member initialization for the instance-specific parameters.

```
private:
    InstP_mCPPInstP_T mCPPInstP_InstP = {
        // Variable: K
        // Referenced by: '<Root>/Gain'
        3.0
    };
```

This direct specification results in more concise code that is more efficient at run time.

Code replacement for trigonometric functions that use lookup table approximation

Starting in R2021b, you can optimize code generated from a Trigonometric Function block that uses the Lookup algorithm by using a code replacement library. In the code replacement entry for the `sin`, `cos`, `sincos`, or `atan2` function, set **Algorithm** to `Lookup`. You can also specify the angle unit for the function as `radian` or `revolution` by using the new parameter **Angle unit**. For more information, see [Algorithm-Based Code Replacement and Algorithm](#).

Verification

Communication I/O information display during SIL or PIL simulation

Use the command-line configuration parameter `SILPILVerboseOutput` to specify the display of communication I/O information during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. For more information, see [Troubleshooting Host-Target Communication](#).

Signal and state logging for SIL and PIL simulations

R2021b provides these software-in-the-loop (SIL) and processor-in-the-loop (PIL) enhancements:

- Logging of nonvirtual bus data for top-model and Model block simulations.
- Logging of signal and state data for the atomic subsystem workflow.

For information about current limitations, see [SIL and PIL Limitations](#).

LDRA tool suite code coverage analysis

During software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations, you can perform code coverage analysis by using the third-party LDRA tool suite, version 9.8.4. Previously, version 9.4.6 was supported.

For the **Third-party tool** configuration parameter, the option `LDRA Testbed` is replaced by `LDRAcover` or `LDRA tool suite`. For more information, see [Configure Code Coverage with Third-Party Tools](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2021a

Version: 7.6

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Multiple signature for software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution

Starting from R2021a, you can generate static and dynamic libraries from multiple signatures of an entry-point function. You can validate the generated multisignature libraries by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution.

For example, if you have an entry-point function `myAdd`, you can generate a multisignature SIL/PIL MEX function. Set the code verification mode to SIL and generate the code by using this `codegen` command:

```
cfg = coder.config('lib');
cfg.VerificationMode = "SIL";
codegen -config cfg myAdd -args {0,0} -args {int8(0),int8(0)} -report
```

This command generates a multisignature SIL MEX function `myAdd_sil.mex`. You can verify the output of this function by providing different input values to the function.

```
myAdd_sil(3,4)

### Starting SIL execution for 'myAdd'
   To terminate execution: clear myAdd_sil
ans =
     7

myAdd_sil(int8(3),int8(4))

ans =
   int8
     7
```

For more information, see [Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution](#).

Reduction of violations for MISRA C++:2008 and AUTOSAR C++14 rules in generated code

In R2021a, the generated code has fewer violations of several rules in the required categories of MISRA C++:2008 and AUTOSAR C++14 coding standards. Some of these rules are:

- for loops: MISRA C++:2008 Rule 6-5-4 (Polyspace Bug Finder), MISRA C++:2008 Rule 6-5-5 (Polyspace Bug Finder), MISRA C++:2008 Rule 6-5-6 (Polyspace Bug Finder), and AUTOSAR C++14 Rule A6-5-2 (Polyspace Bug Finder)
- else-if statements: MISRA C++:2008 Rule 6-4-2 (Polyspace Bug Finder)
- Minimal scoping: MISRA C++:2008 Rule 3-4-1 (Polyspace Bug Finder)
- Casting: MISRA C++:2008 Rule 5-0-10 (Polyspace Bug Finder) and MISRA C++:2008 Rule 5-0-20 (Polyspace Bug Finder)
- Dead code: MISRA C++:2008 Rule 0-1-9 (Polyspace Bug Finder) and AUTOSAR C++14 Rule A0-1-1 (Polyspace Bug Finder)

- Namespaces: MISRA C++:2008 Rule 7-3-1 (Polyspace Bug Finder)

For more information on how to generate code that has improved MISRA and AUTOSAR compliance, see [Generate C/C++ Code with Improved MISRA Compliance](#).

Format generated code by using clang-format

Starting in R2021a, you can format the generated code by using an existing `clang-format` file or have the code generator create a `clang-format` file. Set this option by creating a `coder.EmbeddedCodeConfig` object and setting `ClangFormatFile` to `Existing` or `Generate`. By default, this option is set to `Generate`, which creates a `clang-format` file for your build. If the option is set to `Existing` and is unable to find the required file, the code generator uses a built-in format specification file.

In R2021a, the `CodeFormattingTool` flag enables you to choose how to format the code. This flag has these settings:

- `Clang-format`: The code generator formats your code by using `clang-format`.
- `Auto`: Uses an internal heuristic to determine if the generated code is formatted by `clang-format` or a MathWorks formatting tool. To determine whether your code is formatted by `clang-format`, in a `coder.config` object, set the `Verbosity` option to `'Verbose'`.
- `MathWorks`: Causes the code generator to revert to the legacy code formatting setting.

The `CodeFormattingTool` setting is available for all configuration objects, namely `coder.EmbeddedCodeConfig`, `coder.CodeConfig`, and `coder.MexCodeConfig`.

Compatibility Considerations

In R2021a, the default formatting of the generated code might change. If this causes issues in your workflow, set the `CodeFormattingTool` under your configuration object as `MathWorks`.

Model Architecture and Design

Step entry-point functions generated for rate-based and concurrent execution models declared in `model.h`

Starting in R2021a, in header file `model.h`, the code generator declares a step entry-point function for each task in a model. Prior to R2021a:

- For rate-based models, placement of the entry-point step function declarations depended on the setting of the model configuration parameter **Generate an example main program**. If you cleared the parameter, the code generator placed the entry-point step function declarations in `private.h`. If you selected the parameter, the code generator placed the entry-point step function declarations in `model.h`.
- For a model configured for concurrent execution, the code generator declared the step entry-point functions in header file `private.h`.

The R2021a code generator behavior improves code integration by making the step entry-point function for each task accessible in `model.h`, regardless of the concurrency and **Generate an example main program** configuration.

For more information about modeling styles and code generation, see Design Models for Generated Embedded Code Deployment.

Code Interface Configuration and Integration

C++ class interface configuration by using a code mappings workflow

R2021a introduces interactive and programmatic code mappings workflows to configure the generated C++ class interface from a Simulink model. You can use the Code Mappings editor or programmatic interface to configure the following aspects of a C++ class interface:

- Class information - Class name & namespace
- Class member information - Class member access & visibility
- Class method information - Class method names & arguments

For more information, see **Code Mappings editor**, Interactively Configure C++ Interface, and Programmatically Configure C++ Interface.

Compatibility Considerations

Previously, the configuration parameters **Parameter visibility**, **Parameter access**, **External I/O visibility**, and **External I/O** and configuration dialog boxes enabled the configuration of generated C++ class interfaces. The functionality of these parameters and accessibility of the dialog boxes is now located in the Code Mappings editor.

The `RTW.ModelCPPClass`, `RTW.ModelCPPArgsClass`, and `RTW.getClassInterfaceSpecification` classes and their associated methods are not recommended. Replace usage of these functions with the `coder.mapping.api.CodeMappingCPP` object.

Legacy models continue to be supported and automatically migrate into the code mappings environment.

Instance specific parameter support for C++ class interfaces

R2021a supports the generation of a C++ class interface for a model with instance-specific parameters. You can specify this behavior for a model by using the Code Mappings editor or programmatic interface to configure model parameter arguments. For more information, see Interactively Configure C++ Interface.

Auto data initialization for new storage classes

In the Embedded Coder Dictionary, when you create a storage class, the configuration options available for the **Data Initialization** property are limited to `Dynamic`, `Static`, or `None`. You can apply to signals and states only those storage classes that have dynamic initialization. You can apply to parameters only those storage classes that have static initialization. In R2021a, the default data initialization setting for a new storage class is `Auto`. With this setting, you can apply the same storage class to signals, states, and parameters. The generated code statically initializes parameter data and dynamically initializes signal and state data.

For more information, see **Embedded Coder Dictionary**.

Dimension preservation of multidimensional arrays for Simulink.Bus object

In R2020b, when the model configuration parameter **Array layout** was set to Row-major, you could preserve dimensions for data elements in the model by configuring storage classes with the **Preserve array dimensions** property. This specification did not apply to elements in the `Simulink.Bus` object.

In R2021a, when the model configuration parameter **Array layout** is set to Row-major, you can preserve dimensions of the elements in the `Simulink.Bus` object by using either of these methods:

- In the Simulink Bus Editor, select the **Preserve element dimensions** property.
- In the MATLAB interface, specify the `PreserveElementDimensions` property. For example:

```
MyBus = Simulink.Bus
MyBus.PreserveElementDimensions = 1
```

This property does not affect scalar or vector bus elements.

For more information, see [Preserve Dimensions of Bus Elements in Generated Code](#).

Calibration file generation

Starting in R2021a, you can generate multiple versions (including latest version 1.7) of an A2L file according to the ASAM ASAP2 standard. The new tool enables you to customize the A2L file. For example you can include or exclude comments, include the name of the A2L file, and include the location where to save the A2L file.

Using the **Generate Calibration Files** tool, you can generate a CDFX file according to the ASAM CDF (Calibration Data Format) standard that contains the description of tunable model parameters values and the associated metadata.

For more information, see [Generate ASAP2 and CDF Calibration Files](#).

Code configuration for data dictionary defaults

Using the `CoderDictionary` object, you can now query and set the code settings of dictionary defaults in an Embedded Coder dictionary within a Simulink data dictionary. For more information, see `coder.mapping.api.CoderDictionary`.

ASAP2 system target file being removed

Warns

Support for the `asap2.tlc` system target file will be removed in a future release. Starting in R2021a, use the **Generate Calibration Files** tool to generate ASAP2 files. For more information, see [Generate ASAP2 and CDF Calibration Files](#).

Functionality being removed or changed

Functionality	Result	Use Instead	Compatibility Considerations
Advanced model configuration parameters for applying memory sections to data and functions: Package, Refresh package list, Initialize/Terminate, Execution, Shared utility, Constants, Inputs/Outputs, Internal data, Parameters, and Validation results	Parameters have been removed.	Use the Code Mappings editor or code mappings programmatic interface to configure memory sections for data and functions. See Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings and Control Data and Function Placement in Memory by Inserting Pragmas.	Adjust scripts so that they use the code mappings programmatic interface. See C Data and Function Configuration.
Shared utilities identifier format model configuration parameter	Parameter has been removed.	Use the Code Mappings editor to configure naming rules for shared utility functions. See Migration of Memory Section and Shared Utility Settings from Configuration Parameters to Code Mappings and Configure C Code Generation for Model Entry-Point Functions.	Adjust scripts so that they use the code mappings programmatic interface. See C Data and Function Configuration.
ASAP2 interface model configuration parameter	Parameter has been removed.	Use the Generate Calibration Files tool to generate ASAP2 related code and ASAP2 file. See Generate ASAP2 and CDF Calibration Files.	Use the <code>coder.asap2.export</code> function to generate ASAP2 file. See <code>coder.asap2.export</code> .

Code Generation

Enhanced generated code to reduce MISRA C:2012 Rule 12.2 violations

In R2021a, the code generator produces code that reduces some violations of the MISRA C:2012 Rule 12.2. For more information, see MISRA C:2012 Rule 12.2 (Polyspace Code Prover).

Removal of typedef from C++ struct definitions

In R2020b, the code generator included the `typedef` keyword in C++ struct definitions. For example:

```
// External inputs (root inport signals with default storage)
typedef struct {
    real_T In1;           // '<Root>/In1'
    real_T In2;           // '<Root>/In2'
    real_T In3;           // '<Root>/In3'
    real_T In4;           // '<Root>/In4'
} ExternalInputs;
```

In R2021a, the code generator removes the `typedef` keyword in struct definitions to make the C++ generated code compliant with the AUTOSAR C++ 14 rule A7-1-6. For example:

```
// External inputs (root inport signals with default storage)
struct ExternalInputs {
    real_T In1;           // '<Root>/In1'
    real_T In2;           // '<Root>/In2'
    real_T In3;           // '<Root>/In3'
    real_T In4;           // '<Root>/In4'
};
```

The `typedef` keyword is removed from struct definitions in `model.h` and `model_types.h` files. The generated code reduces some violations of the AUTOSAR C++14 Rule A7-1-6 (Polyspace Bug Finder).

Some struct definitions in the `_sharedutils` folder, such as `rtwtypes.h` and alias definitions that use `typedef`, are not affected. Shared bus types are affected.

Braced variable initialization for C++ 11 library

In R2020b, during C++ code generation, the code generator initialized local and global variables by using assignment operator (`=`) and braces `{ }`. For example:

```
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;
```

In R2021a, to make the C++ generated code compliant with AUTOSAR C++ 14, the code generator initializes local and global variables in braces `{ }`. For example:

```
real_T const_val[4]{ 1.0, 2.0, 3.0, 4.0 } ;
```

When you set **Language** as C++ and **Standard math library** as C++11 (ISO), the code generator enables braced initialization. The generated code reduces most violations of the AUTOSAR C++14 Rule A8-5-2 (Polyspace Bug Finder).

Code generation and SIL or PIL simulations for protected models from R2018b and later releases

Previously, code generation and SIL or PIL simulations for protected models from R2018b and later releases were supported for these system target file types:

- ERT
- ERT-based
- AUTOSAR Classic
- GRT (Embedded Coder required)

R2021a extends the support to GRT-based system target files. For more information, see [Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code](#).

Performance

Code execution profiling information in Code view

In R2021a, when you run your model in the SIL/PIL Manager app, you can view code execution profiling information in the Code view. To collect execution-time metrics, in the Configuration Parameters dialog box, select **Measure task execution time** and set **Measure function execution times** to **Coarse** or **Detailed**. Simulate the model in simulation-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode.

The code execution profiling information is displayed in the Code view. To view execution profiling details for a function call, place your cursor over the function call in the Code view. You can still access the code execution profiling information in the Profiling display and in the code execution profiling report. For more information, see [View and Compare Code Execution Times](#).

Visualization of task scheduling

If you enable code execution profiling for a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, you can use the Simulation Data Inspector to visualize task scheduling and the order of function calls. At the end of the SIL or PIL simulation, perform one of these actions:

- Run the `schedule` function.
- From the **SIL/PIL** tab, in the **Results** gallery, click **Generate Schedule**.

For more information, see [Visualize Task Scheduling and Analyze Results and Export Test Cases](#).

Removal of instrumentation overhead from execution-time profiling by using target package

To improve execution-time profiling of generated code that is run on deterministic hardware, you can run a processor-in-the-loop (PIL) simulation that discards the time overhead introduced by the code instrumentation. In R2021a, with the `target` package, you can:

- Use the target hardware to estimate the average overhead value.
- Specify the value manually.

R2021a provides these new classes:

- `target.ProfilingTaskOverhead`
- `target.ProfilingFunctionOverhead`
- `target.ProfilingFreezingOverhead`

The `target.Processor` class has a new property, `Overheads`.

For more information, see [Remove Instrumentation Overheads from Execution Time Measurements](#).

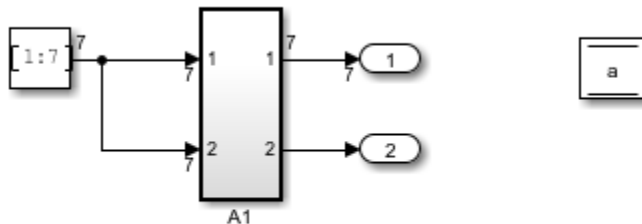
Enhanced code for models containing mask blocks or Data Store Memory blocks

Embedded Coder now generates code that improves code quality for the models containing mask blocks or Data Store Memory blocks. For example, one change you might see is the addition of `const` qualifiers for read-only variables.

Compatibility Considerations

In R2021a, the function arguments in the generated code are reordered.

Consider this model.



In R2020b, the `mdstore_A1` function contained this code:

```
void mdstore_A1(const real_T rtu_In1[7], real_T rty_Out1[7], real_T *rty_Out2,
                DW_A0_mdstore2_T *localDW, real_T *rtd_b)
{
    int32_T i;
    ...
}
```

In R2021a, the `mdstore_A1` function contains this code:

```
void mdstore_A1(const real_T rtu_In1[7], real_T rty_Out1[7], real_T *rty_Out2,
                const real_T *rtd_b, DW_A0_mdstore2_T *localDW)
{
    int32_T i;
    ...
}
```

GCC ARM Cortex-A code replacement library contains other ARM libraries

Previously, when you selected an ARM Compatible device vendor and the ARM Cortex-A device type, you could choose from these ARM code replacement libraries:

- ARM Cortex-A
- GCC ARM Cortex-A

- Inlined ARM Neon Intrinsics

In R2021a, these three libraries are included in the GCC ARM Cortex-A code replacement library. As a result, the more efficient entries that were in the Inlined ARM Neon Intrinsics library replace the less efficient entries that were in the GCC ARM Cortex-A library in R2020b. These entries are for matrix addition, subtraction, and element-wise multiplication operations. As a result, when you specify a GCC ARM Cortex-A library in R2021a, the generated code contains fewer data copies and wrapper functions for SIMD operations than it did in R2020b. If you have a DSP System Toolbox Support Package for ARM Cortex-A Processors, the entries in the ARM Cortex-A library are also included in the GCC ARM Cortex-A library. For more information, see [Code replacement libraries and What Is Code Replacement?](#).

Multithreading capabilities for more Image Processing Toolbox functions

In R2021a, if you use a compiler that supports the Open Multiprocessing (OpenMP) application interface, you can generate multithreaded C/C++ functions for some Image Processing Toolbox functions that are included in MATLAB code or in Simulink models with MATLAB Function blocks or MATLAB System blocks. This enhancement improves the function execution speed.

To enable multithreading, select the configuration parameters **Specify custom optimizations** and **Generate parallel for loops**.

The new optimized functions that have multithreading capabilities are:

- `bwlabel`
- `houghpeaks`
- `otsuthresh`
- `bwareaopen`
- `bwboundaries`
- `imbinatfilt`

The code generator generates multithreaded code for the `houghpeaks` function only when the function takes a large Hough transform matrix as an input.

In R2020b, the code generator produced this C code snippet for a MATLAB function containing an image processing function `bwlabel`:

```
...
for (lastRunOnPreviousColumn = 0; lastRunOnPreviousColumn < 1024;
    lastRunOnPreviousColumn++) {
    firstRunOnPreviousColumn = lastRunOnPreviousColumn << 10;
    if (img[firstRunOnPreviousColumn]) {
        numRuns++;
    }
    for (k = 0; k < 1023; k++) {
        runCounter = k + firstRunOnPreviousColumn;
        if (img[runCounter + 1] && (!img[runCounter])) {
            numRuns++;
        }
    }
}}...
```

The loop executed sequentially.

In R2021a, the code generator produces this code snippet:

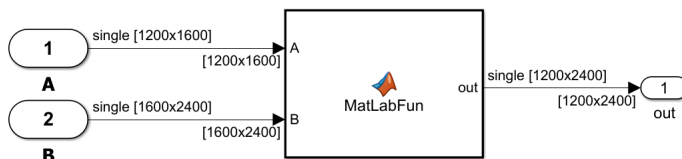
```
...
#pragma omp parallel for \
  num_threads(omp_get_max_threads()) \
  private(rootj,b_c,label,c_tmp,L_tmp,x,c_c,d1,r,i1,i2,exitg1,i3) \
  firstprivate(guard1)
  for (thread = 0; thread < 8; thread++) {
    c_tmp = (thread << 7) + 1;
    L_tmp = (thread + 1) << 7;
    chunksSizeAndLabels[c_tmp - 1] = L_tmp + 1;
    x = (int)ceil(((double)c_tmp - 1.0) * 1024.0 / 2.0);
    label = (double)x + 1.0;
    L_tmp -= c_tmp;
    for (c_c = 0; c_c <= L_tmp; c_c++) {
      b_c = c_tmp + c_c;
      for (r = 0; r < 1024; r++){
... }}}
```

The generated code has the pragma for OpenMP (Open Multiprocessing) before the body of the loop. OpenMP enables shared-memory and multicore platforms to execute loops in parallel. This parallel execution improves the execution speed of the generated code. For more information, see [Speed Up for-Loop Implementation in Code Generated by Using parfor and Algorithm Acceleration Using Parallel for-Loops \(parfor\)](#).

Improved cache performance of generated code containing distributed loop nests

In R2020b, for some modeling patterns, the generated code contained nested for-loops that caused cache misses. In R2021a, when possible, you can generate optimized code containing distributed perfect loop nests that can interchange the order of loop execution. This loop interchange lowers cache misses by accessing data from a single cache block, which can improve the execution speed of the generated code.

Consider the model `matrix_mux` that has a MATLAB Function block.



The MATLAB Function block contains code that performs matrix multiplication of the two input matrices of dimension [1200x1600] and [1600x2400]:

```
function out = MatLabFun(A, B)
out = A * B;
end
```

In R2020b, the code generator produced this code:

```
/* Model step function */
void matrix_mux_step(void)
{
  int32_T i;
```

```

int32_T i_0;
int32_T i_1;
int32_T out_tmp;

/* Output: '<Root>/out' incorporates:
 * Inport: '<Root>/A'
 * Inport: '<Root>/B'
 */
for (i_0 = 0; i_0 < 2400; i_0++) {
    for (i = 0; i < 1200; i++) {
        out_tmp = i + 1200 * i_0;
        rtY.out[out_tmp] = 0.0F;
        for (i_1 = 0; i_1 < 1600; i_1++) {
            rtY.out[out_tmp] += rtU.A[1200 * i_1 + i] * rtU.B_n[1600 * i_0 + i_1];
        }
    }
}

```

The generated nested loop performed matrix multiplication by evaluating the iteration variable `i_1` in the innermost loop of the generated code.

In R2021a, the code generator produces this code:

```

/* Model step function */
void matrix_mux_step(void)
{
    int32_T i;
    int32_T i_0;
    int32_T i_1;
    int32_T out_tmp;

    /* Output: '<Root>/out' incorporates:
     * Inport: '<Root>/A'
     * Inport: '<Root>/B'
     */
    for (i_0 = 0; i_0 < 2400; i_0++) {
        memset(&rtY.out[i_0 * 1200], 0, 1200U * sizeof(real32_T));
    }

    for (i_0 = 0; i_0 < 2400; i_0++) {
        for (i_1 = 0; i_1 < 1600; i_1++) {
            for (i = 0; i < 1200; i++) {
                out_tmp = 1200 * i_0 + i;
                rtY.out[out_tmp] += rtU.A[1200 * i_1 + i] * rtU.B_n[1600 * i_0 + i_1];
            }
        }
    }
}

```

The generated code contains a distributed perfect loop nest. Compared to the previously generated code, the nested loop changes the execution order of the loops. It performs matrix multiplication by evaluating the iteration variable `i` in the innermost loop of the generated code, allowing a sequential access of the memory that reduces cache misses. This optimization can improve the execution speed of the generated code. You can calculate the cache miss rate by dividing the total cache miss number with the total number of memories requested over a time interval and then expressing the ratio in a percentage. In R2021a, when the code generation target hardware is Intel x86-64 (Linux 64), the following optimization of cache-misses for the `matrix_mux` model are observed:

- The level-1 data read miss rate is reduced approximately from 53% to 16%.
- The level-1 data write miss rate is reduced approximately from 41% to 0.01%.

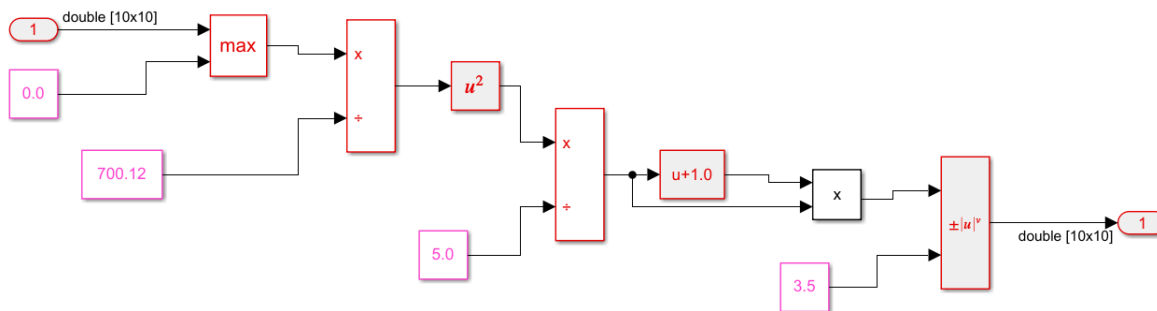
In this observed case, these configurations are used:

- The configuration parameter **Build Configuration** is set to `Faster Runs`.
- The parameter **Array layout** is set to `Column-major`.

Improved expression folding in generated code

In R2021a, for some modeling patterns, you can generate optimized code that folds expressions that reuse input variables of the expressions to hold the computed output results. This optimization reduces data copies in the generated code. The parameter **Eliminate superfluous local variables (expression folding)** enables this optimization.

Consider the model `mSquarePowExprFolding`.



In R2020b, the code generator produced this code:

```
void mSquarePowExprFolding_step(void)
{
    real_T rtb_Product;
    int32_T i;
    for (i = 0; i < 100; i++) {
        rtb_Product = fmax(rtU.In1[i], 0.0) / 700.12;
        rtb_Product = rtb_Product * rtb_Product / 5.0;
        rtb_Product *= rtb_Product + 1.0;
        rtY.Out1[i] = rt_powd_snf(rtb_Product, 3.5);
    }
}
```

The generated code contained a separate expression for the operation where the variable `rtb_Product` was used as an input and output variable.

In R2021a, the code generator produces this code:

```
void mSquarePowExprFolding_step(void)
{
    real_T rtb_Product;
    int32_T i;
    for (i = 0; i < 100; i++) {
        rtb_Product = fmax(rtU.In1[i], 0.0) / 700.12;
        rtb_Product = rtb_Product * rtb_Product / 5.0;
        rtY.Out1[i] = rt_powd_snf((rtb_Product + 1.0) * rtb_Product, 3.5);
    }
}
```

The generated code does not contain the expression. The bolded code line folds the expression, computes the output, and stores it directly in the output structure field `rtY.Out1[i]`. For more information, see [Minimize Computations and Storage for Intermediate Results at Block Outputs](#).

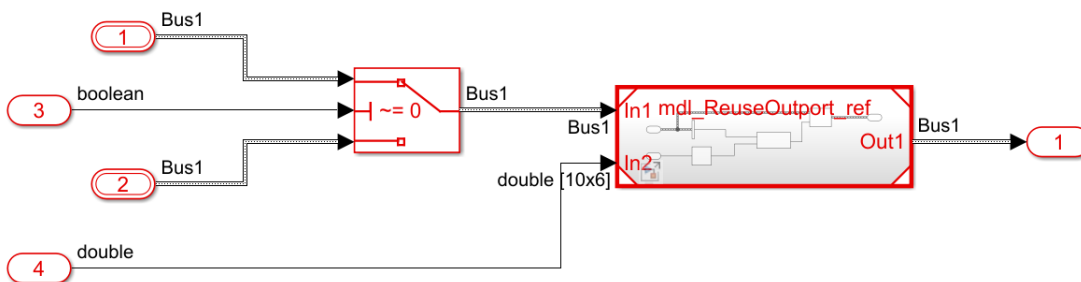
Improved root output port buffer reuse to reduce data copies

In R2021a, the generated code contains fewer data copies for models containing root output blocks that directly connect to one of these blocks:

- Referenced model that is configured to use the same buffer for input ports and output ports. For more information, see [Configure Name and Arguments for Individual Step Functions](#).
- MATLAB Function block with inplace (that is, use the same input and output variable) specification. For more information, see [Specify Buffer Reuse for MATLAB Function Blocks in a Path](#).
- Unit Delay block.

Eliminating redundant data copies conserves RAM and ROM consumption and improves execution speed.

Consider the model mdl_ReuseOutputport that contains a referenced model mdl_ReuseOutputport_ref connected to a root output block.



The referenced model mdl_RootOutputport_ref is configured to use the same buffer for In1 and Out1.

In R2020b, the code generator produced this code:

```

/* Model step function */
void mdl_ReuseOutputport_step(void)
{
    Bus1 rtb_Switch;
    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Inport: '<Root>/In3'
     */
    if (rtU.In3) {
        rtb_Switch = rtU.In1;}
    else {
        rtb_Switch = rtU.In2;
    }
    /* End of Switch: '<Root>/Switch' */
    /* Output: '<Root>/Out1' incorporates:
     * ModelReference: '<Root>/Model'
     * Switch: '<Root>/Switch'
     */
    rtY.Out1 = rtb_Switch;
}

```

```

/* ModelReference: '<Root>/Model' incorporates:
 * Inport: '<Root>/In4'
 * Outport: '<Root>/Out1'
 */
mdl_RootOutport_ref_step(&Model, &rtU.In4[0], &rtY.Out1);
}

```

The generated code contained unnecessary data copy to the local variable `rtb_Switch`.

In R2021a, the code generator produces this code:

```

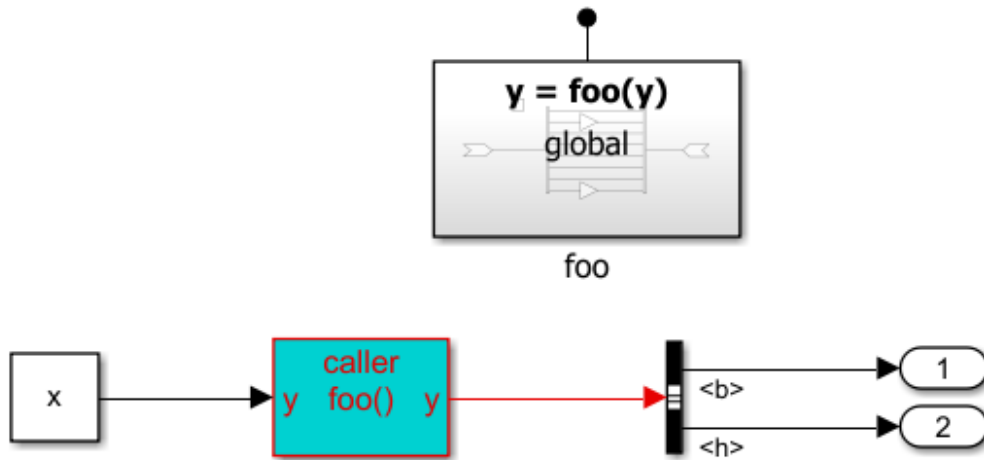
/* Model step function */
void mdl_ReuseOutport_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   * Inport: '<Root>/In3'
  */
  if (rtU.In3) {
    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In1'
    */
    rtY.Out1 = rtU.In1;}
  else {
    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In2'
    */
    rtY.Out1 = rtU.In2;
  }
  /* End of Switch: '<Root>/Switch' */
  /* ModelReference: '<Root>/Model' incorporates:
   * Inport: '<Root>/In4'
   * Outport: '<Root>/Out1'
  */
  mdl_RootOutport_ref_step(&Model, &rtU.In4[0], &rtY.Out1);
}

```

The code generator does not generate the local variable `rtb_Switch` to hold the output of the Switch block. Instead, it stores the Switch block output in the root output structure field `rtY.out1`.

Reduced data copies for blocks with bus inputs and outputs

In R2021a, the generated code contains fewer data copies for some modeling patterns in which a block has a bus data type as an input and output. For example, the model `mBusInSimulinkFunctionBlock` contains a Simulink Function block that takes a bus signal as an input and performs an in-place operation on the bus defined by the function `foo`.



Copyright 2020 The MathWorks, Inc.

R2020b Generated Code	R2021a Generated Code
<pre>void foo(Bus2 *rtuy_y) { Bus2 rtb_BusCreator; rtb_BusCreator = *rtuy_y; rtb_BusCreator.b = rtuy_y->b << 1; rtb_BusCreator.h = (int8_T)(rtuy_y->h << 1); *rtuy_y = rtb_BusCreator; }</pre>	<pre>void foo(Bus2 *rtuy_y) { rtuy_y->b <<= 1; rtuy_y->h <<= 1; }</pre>

In R2020b, the generated code contained data copies to the local `rtb_BusCreator` struct. In R2021a, those data copies are eliminated. The left-shift operation occurs in place reducing RAM consumption.

Verification

PIL target connectivity with debugger

You can use your debugger to provide processor-in-the-loop (PIL) connectivity for target hardware that your debugger supports. Implement a `target.DebugIOTool` debugger abstraction interface that removes the need for custom `rtiostream` functionality. To register the interface with MATLAB, use the `target.create`, `target.get`, and `target.add` functions.

R2021a provides these new classes:

- `target.ApplicationStatus`
- `target.Breakpoint`
- `target.DebugIOTool`
- `target.ExecutionService`
- `target.ExecutionTool`
- `target.MATLABDependencies`

For more information, see [Set Up PIL Connectivity by Using target Package](#).

Unit-tests for generated code from subsystems within code from parent model

If you have a model that contains atomic subsystems, you can use Simulink Test and the SIL/PIL Manager to perform unit tests on code generated from the subsystems. For each atomic subsystem, you can:

- Create a test harness and run back-to-back simulations that test numerical equivalence between the model subsystem and its generated code.
- Export the equivalence test to Simulink Test.
- Analyze code coverage by using Simulink Coverage™.

For more information, see:

- [Choose a SIL or PIL Approach](#)
- [Test Atomic Subsystem Generated Code](#)
- [Atomic Subsystem Workflow Limitations](#)

Code view in SIL/PIL Manager

At the end of a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, the SIL/PIL Manager displays generated code in the Code view, which enables you to:

- Analyze generated code.
- See code metrics.
- Trace between model elements and generated code.

For more information, see:

- Analyze Results and Export Test Cases
- **SIL/PIL Manager**
- Run SIL Simulation That Generates Execution-Time Metrics

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2020b

Version: 7.5

New Features

Bug Fixes

Compatibility Considerations

Model Architecture and Design

Determine programmatically if model or data dictionary contains Embedded Coder Dictionary

In R2020b, you can programmatically determine if a model or Simulink data dictionary contains an Embedded Coder Dictionary by using the new function `coder.dictionary.exist`.

Symbolic dimension inputs for Add, Subtract, Sum of Elements, and Sum blocks

Previously, you could not generate code by using symbolic dimensions as inputs for Add, Subtract, Sum of Elements, and Sum blocks when the block parameter **Sum over** was specified as `Specified dimension`. The code generator produced an error.

In R2020b, you can generate code by using symbolic dimensions as inputs for Add, Subtract, Sum of Elements, and Sum blocks when the block parameter **Sum over** is `Specified dimension`. You can use symbolic dimensions to set constraints for signal dimensions and as block parameters for Add, Subtract, Sum of Elements, and Sum blocks. For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#).

Improved readability for preprocessor conditionals in generated code

You can generate code from models containing one or more variant choices. The generated code contains preprocessor conditionals that control the activation of each variant choice.

In R2020a, the generated code for variant systems and variant subsystems had consecutive preprocessor conditionals with the same condition, resulting in redundant `#if` conditions.

For example, the inner condition `#if isfoo` was executed only when the outer condition `#if isfoo && isbar` was true. The inner condition `#if isfoo` was a redundant condition.

```
real_T rtb_Merge;
real_T rtb_VariantMerge_For_Variant_So;
#if isfoo && isbar
    #if isfoo
        rtb_Merge = mMergeLocalize_P.Constant1_Value;
    #endif
#endif
#if isfoo
    rtb_VariantMerge_For_Variant_So = rtb_Merge;
#endif
```

In R2020b, the generated code for variant systems and variant subsystems does not contain the redundant `#if` conditions that are implied as true. This optimization improves the readability and efficiency of the generated code.

For example:

```
real_T rtb_Merge;
real_T rtb_VariantMerge_For_Variant_So;
#if isfoo && isbar
```

```
    rtb_Merge = mMergeLocalize_P.Constant1_Value;
#endif
#if isfoo
    rtb_VariantMerge_For_Variant_So = rtb_Merge;
#endif
```

The code generator cannot optimize all instances of redundant preprocessor conditionals for better readability and code efficiency.

Memory section configurations for atomic subsystems

Previously, to override a memory section for an atomic subsystem, you had to use the model configuration parameter `MemSecPackage` in the MATLAB Command Window.

In R2020b, you can select the memory sections from built-in packages that you load in the Embedded Coder Dictionary by using these subsystem block parameters:

- **Memory section for initialize/terminate functions**
- **Memory section for execution functions**
- **Memory section for constants**
- **Memory section for internal data**
- **Memory section for parameters**

You cannot select a memory section that you newly define in the Embedded Coder Dictionary. For more information, see [Override Memory Section for Atomic Subsystem](#).

Code Interface Configuration and Integration

Streamlined model data configuration for code generation

R2020b simplifies how you configure individual model data elements, such as block data and signal lines, for code generation. R2018a introduced the Code Mappings editor and an API for specifying default code generation configurations for categories of data elements across a model. Starting in R2020b, without affecting your model design configuration, from the Code Mappings editor, you can configure default settings for categories of data. Then, override those settings for code generation, as needed, for specific data elements. When producing code for data, the code generator uses storage classes that you specify to determine properties, such as whether the data is structured, naming rules for definition and header files, and whether the data is placed in a memory section.

Use the Code Mappings editor to map an individual model data element to:

- **Auto**, which specifies that the code generator use heuristics and model configuration parameter settings (for example, **Default parameter behavior**) to determine how to best represent the data element in the generated code. When possible, the code generator omits data from the code.
- A default storage class to indicate use of the specified default for the corresponding data element category (for example, inports, model parameters, signals, and local data stores).
- Predefined storage classes, such as `ExportedGlobal` and storage classes available in the Simulink package, and storage classes that you define.

When you specify a storage class in the Code Mappings editor, you can view and set relevant storage class properties in the Property Inspector, which also opens in the coder app. For example, for a storage class other than **Auto** that you specify for an individual data element, you specify a value for the **Identifier** property to name an unnamed model data element (required) or override a model name in the generated code for integration purposes.

Through code mappings, you can also associate a model with code configurations for multiple platforms.

Platform	System Target File	Programming Language
C rapid prototyping	GRT-based	C
C production	ERT-based	C
AUTOSAR classic platform	AUTOSAR	C
AUTOSAR adaptive platform	AUTOSAR Adaptive	C++

Starting in R2020b, you can copy code mappings when you convert a subsystem to a referenced model. See [Copy Code Mappings When Converting Subsystems to Referenced Models \(Simulink Coder\)](#).

For more information, see [C Code Generation Configuration for Model Interface Elements](#), **Code Definition and Mapping Limitations and Considerations**, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

For more information on preserving dimensions for individual modeling elements, see the release note “Dimension preservation of multidimensional arrays for individual model elements” on page 5-5.

Migration of Preexisting Models

When you open a model created in a previous release, Simulink migrates data configured for code generation within the blocks and signal lines of a model to the Code Mappings editor. Data configured for code generation within a model includes data represented by:

- Inport blocks
- Outport blocks
- Signal lines
- Block states
- Data stores
- Parameter objects in the model workspace

Simulink does not migrate data that is configured for code generation in external sources, such as the base workspace or a data dictionary.

For information about code mappings, see C Code Generation Configuration for Model Interface Elements, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`. For information on specifying code mappings for AUTOSAR applications, see AUTOSAR Component Configuration (AUTOSAR Blockset).

Compatibility Considerations

The code mappings interfaces for configuring data are compatible with common data configuration scenarios in previous releases of Embedded Coder software.

You can work around many of the incompatibilities by developing MATLAB scripts to run in Embedded Coder. For more information, see “Functionality being removed or changed” on page 5-6 and Migration of Model Data Configurations to Code Mappings.

Dimension preservation of multidimensional arrays for individual model elements

In R2020a, when the model configuration parameter **Array layout** was set to Row-major, you could preserve dimensions for individual model data elements by using the Model Data Editor.

In R2020b, when the model configuration parameter **Array layout** is set to Row-major, you can preserve dimensions of multidimensional arrays for individual model data elements by using the new tabs in the Code Mappings editor such as **Inports**, **Outports**, **Parameters**, **Data Stores**, and **Signals/States**.

You can also preserve dimensions for:

- `mpt.Parameter` objects.
- Signal objects, when you apply a custom package to the signal object and rename the storage class that supports dimension preservation.

For more information, see Preserve Dimensions of Multidimensional Arrays in Generated Code.

Custom data type configuration and modification

For custom data types, such as `Simulink.AliasType` object, that have the **Data scope** specified as **Imported** and that have **Header file** properties, you can configure the generated code to import the type definition from your external code. Previously, if you modified the custom data type and rebuilt the model, the code generator produced an error.

In R2020b, if you modify the custom data type and the data type is not used by shared functions or shared constants, the code generator does not produce an error. You can rebuild the model without deleting the `s\prj` folder. If the modified custom data type is used by shared functions or shared constants, you have to rebuild the model after deleting the `s\prj` folder.

For more information, see [Control File Placement of Custom Data Types](#).

Functionality being removed or changed

The new code mappings interfaces streamline how you configure model data elements for code generation. These interfaces introduce:

- Incompatibilities with uncommon data configuration scenarios from previous releases of Embedded Coder software.
- Changes for the use of other Simulink interfaces for configuring data, such as the Model Data Editor, the Model Explorer, and the Signal Properties dialog box.

Simulink interface changes for data configuration

Still runs

In R2020b, the Code Mappings editor is the primary location to configure model data elements for code generation.

- In the Model Data Editor, the Code view has been removed. The editor does not display a **Code** section in the Property Inspector.
- You can no longer configure code generation properties in the Signal Properties dialog box.
- For `Simulink.Signal` objects in the model workspace, you can no longer configure code generation properties in the Model Explorer or in the property dialog box. To configure these elements for code generation, use the Code Mappings editor or code mappings API.
- For data objects in the model workspace other than `Simulink.Signal` objects, where previously you could configure code generation properties in the Model Explorer or in the property dialog box, links or buttons take you to the Code Mappings editor instead.
 - In the Model Explorer, in the **Contents** pane, click the **Configure** link in the **Storage Class** column.
 - In the Model Explorer **Dialog** pane and in the property dialog box, on the **Code Generation** tab, click **Configure in Coder App**.

For more information see, [C Code Generation Configuration for Model Interface Elements](#), **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

TypeQualifier property for built-in storage classes no longer used for data objects

You can no longer use the `TypeQualifier` property for built-in storage classes, such as `ExportedGlobal` and `ImportedExtern`, with data associated with data objects because more

robust mechanisms are available for achieving the same results. In previous releases, when you specified the property, the code generator added C qualifiers, such as `const` and `volatile`, to the beginning of data declarations and definitions. You might have set this property as:

- `CoderInfo.TypeQualifier` property for data objects in a workspace or data dictionary
- Port parameter `RTWStorageTypeQualifier`
- Block parameter `RTWStateStorageTypeQualifier` for Data Store Memory, Discrete Filter, Discrete State-Space, Discrete-Time Integrator, Discrete Transfer Fcn, Discrete Zero-Pole, and Memory blocks

To address this change in an existing model that uses the `TypeQualifier` property for data objects, open the model in a release before R2020b. Create and run a MATLAB script that loads the data for the model from a workspace or data dictionary, finds data objects that have the `TypeQualifier` property set to a nonempty string value, and changes the storage class setting to a storage class predefined with the required type qualifier (for example, storage class `Const` includes qualifier `const` in data declarations and definitions). For an example, see [Migration of Model Data Configurations to Code Mappings](#).

Starting in R2020b, use the Code Mappings editor or code mappings API to associate data elements with a storage class that specifies a C qualifier (see [Choose Storage Class for Controlling Data Representation in Generated Code](#)). If none of the available storage classes meets your application requirements, define a new storage class by using the Embedded Coder Dictionary (see [Define Storage Classes, Memory Sections, and Function Templates for Software Architecture](#)). Then, use the **Code Mappings Editor** or code mappings API (`coder.mapping.api.CodeMapping`) to associate the model data to the new storage class.

Code configuration for parameter objects initialized in model workspace initialized from external data sources moved to code mappings

Starting in R2020b, the code mappings interface enables you to associate a model with multiple code generation configurations for data. When you load a model created in a previous release of Simulink and the model workspace is initialized from an external data source, such as a MAT-file, Simulink moves the code configuration for the parameter object to the code mappings for that model.

Once the configuration for data elements in a model has been converted to code mappings, use the Code Mappings editor or the code mappings API to get and set parameter code configuration settings. See [C Code Generation Configuration for Model Interface Elements](#), **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

Copies of blocks or signal lines between models no longer include code configuration

Starting in R2020b, when you use the Simulink Editor to copy a block or signal line to another model, Simulink does not copy the code configuration associated with the copied modeling element. The contents of the `Simulink.CoderInfo` object for the copied modeling element is removed. This change:

- Eliminates unnecessary copies of code configuration information for data configured within the model.
- Supports unique code configuration of data elements for a model and its active system target file.
- Promotes reuse of modeling patterns across models that have different code configurations.

To copy the code configuration information associated with a block or signal line, use the code mappings API. For an example, see [Migration of Model Data Configurations to Code Mappings](#). For information about the API, see `coder.mapping.api.CodeMapping`.

Code configuration for data configured within a model removed for some types of models

Starting in R2020b, for a model created in a previous release, Simulink ignores the code configuration for data elements for library models and models configured with an AUTOSAR system target file. For library models, reconfigure code generation for data in the context of models that use the library (see [C Code Generation Configuration for Model Interface Elements](#)). For AUTOSAR models, see [Map AUTOSAR Elements for Code Generation \(AUTOSAR Blockset\)](#).

This change does not apply to data objects saved in the base workspace or a data dictionary.

To avoid losing the code configuration for data, in an earlier release, create and run a MATLAB script that migrates the model to use external data objects. For an example, see [Migration of Model Data Configurations to Code Mappings](#).

Default (Custom) storage class removed

To prevent confusion with the concept of default code configurations that you can set up by using the Code Mappings editor or code mappings API, you can no longer use the `Default (Custom)` storage class for data configured within a model. The storage class is not recommended and will not be in a future release for global data (data configured in the base workspace or a data dictionary).

For models created in R2020a or earlier, the storage class for a data element is set to `Default (Custom)` when these conditions exist:

- The `StorageClass` property for the `Simulink.CoderInfo` object is set to `Custom`.
- The `CustomStorageClass` property for the `Simulink.CoderInfo` object is not modified or is explicitly set to `Default`.

For an `Outport` block, signal line, block state, data store, or model workspace parameter set to `Default (Custom)`, when you load the model, Simulink converts the storage class setting to `ExportedGlobal` and displays a warning about the change. `ExportedGlobal` is equivalent to `Default (Custom)`.

Starting in R2020b, use the Code Mappings Editor or code mappings API to specify default code generation configurations for categories of data elements. See [C Code Generation Configuration for Model Interface Elements](#), [Code Mappings Editor](#), and `coder.mapping.api.CodeMapping`.

Changing between GRT-based and ERT-based system target file

Behavior change

Starting in R2020b, when you change the system target file setting for a model between a GRT-based and ERT-based system target file, Simulink applies an alternative code configuration for each system target file.

A change between system target files can occur if you:

- Alternate between the Simulink Coder and Embedded Coder app.
- Change the active configuration set for a model.

- Change the setting of model configuration parameter **System target file**.

It is a best practice to configure data differently for a model depending on whether you are generating rapid-prototyping (GRT) or production (ERT) code. Simulink associates the code configuration with the system target file so that you can set up multiple code configurations for a model.

To copy code mappings when you switch system target files, create and run a MATLAB script that uses the code mappings API to copy relevant code mappings. For an example, see *Migration of Model Data Configurations to Code Mappings*. For information about the API, see `coder.mapping.api.CodeMapping`.

Simulink.CoderInfo object Alignment property for data configured within a model removed

The `Simulink.CoderInfo` object property `Alignment` for data configured for code generation within a model has been removed, including data represented by:

- Inport blocks
- Outport blocks
- Signal lines
- Block states
- Data stores
- Parameter objects in the model workspace

To use the `Alignment` property, represent data by using data objects outside of the model. For an example, see *Migration of Model Data Configurations to Code Mappings*.

Code configuration for subsystem I/O interface within subsystem

Behavior change

Starting in R2020b, for individual subsystems from which you generate code and executable programs by right-clicking the Subsystem block, to include signal data in the generated subsystem code interface, configure the storage class and storage class properties for the subsystem input and output signals within the subsystem. If you configure the signals outside of the subsystem, the generated code does not include variables for the input and output interface data.

For more information, see *Generate Code and Executables for Individual Subsystems*.

Behavior change of Ignore custom storage classes parameter

Behavior change

In R2020a, when you selected the model configuration parameter **Ignore custom storage classes** and configured predefined storage classes by using default mapping in the Code Mappings editor, the code generator ignored the predefined storage class configuration and applied the `Auto` storage class to the parameters.

In R2020b, when you select the model configuration parameter **Ignore custom storage classes** and configure predefined storage classes by using default mapping in the Code Mappings editor, the code generator does not ignore these storage class configurations:

- `ExportedGlobal`

- ImportedExtern
- ImportedExternPointer

Certain APIs for configuring code interfaces not recommended

Still runs

APIs listed in this table are not recommended.

C Function Interface Control	C++ Class Interface Control	C Function Default Mapping (introduced in R2018a)
RTW.configSubsystemBuild	RTW.configSubsystemBuild	coder.mapping.create
RTW.getFunctionSpecification	RTW.getClassInterfaceSpecification	coder.mapping.defaults.get
RTW.ModelSpecificCPrototype	RTW.ModelCPPArgsClass	coder.mapping.defaults.set
	RTW.ModelCPPClass	
	RTW.ModelCPPDefaultClass	

Starting in R2020b, use the new code mappings API. The new code mappings API:

- Provides one programming interface for configuration of default code generation settings for categories of data and functions and individual data elements and functions.
- Supports multiple configuration mappings for a model.
- Eliminates the need to create data objects to configure model data elements for code generation.

For information about the new code mappings API, see `coder.mapping.api.CodeMapping`.

APIs for controlling data interfaces

Still runs

In R2020b, code generation information for model-owned data objects migrates from data objects to the mapping infrastructure. This change might affect existing scripts that you use to manage the code configuration for these data objects.

- You do not need to update scripts that use existing functions to interact with model-owned data objects. When you get and set code generation information by using one of these functions, data objects now communicate with the mapping to maintain the mapping as the single source for this information. This information includes functions such as `assignin`, `evalin`, `getVariable`, `get_param`, `set_param`, and `isequal`.

For example, these workflows do not require updates:

- Getting the handle of a data object in the model workspace.

```
mws = get_param('modelName', 'modelworkspace');
objHandle = mws.getVariable('Param');
```

- Evaluating an expression in the context of the model workspace.

```
mws.evalin("param=Simulink.Parameter; param.CoderInfo.StorageClass='ExportedGlobal';")
```

- Specifying code generation settings for a signal object stored on a port.

```
portHandles = get_param('blkPath', 'PortHandles');
get_param(portHandles.Output, 'StorageClass');
set_param(portHandles.Output, 'StorageClass', 'ExportedGlobal');
```

- Specifying code generation settings for a root output block.

```
get_param('blkPath', 'StorageClass');
set_param('blkPath', 'StorageClass', 'ExportedGlobal');
```

- Comparing data objects.

```
p1 = Simulink.Parameter;
p1.CoderInfo.StorageClass = 'Custom';
p1.CoderInfo.CustomStorageClass = 'ExportToFile';
p2 = copy(p1);
isequal(p1, p2); % Returns true
p1.CoderInfo.StorageClass = 'ExportedGlobal';
isequal(p1, p2); % Returns false
```

- Previously, when you defined your own storage class to apply to model-owned data, you created it in a data class package. A data object created from that package used only storage classes from that package. In R2020b, these storage classes migrate from packages to the Embedded Coder Dictionary. For existing functions, the storage classes available for model-owned data now come from the coder dictionary and include not only storage classes from the data object package, but storage classes from other packages and built-in storage classes. If you have multiple packages that have been migrated into the coder dictionary, there is a potential for naming conflicts. In this case, storage classes with the same name have the package name added as a suffix. If you use a storage class that has this added suffix, you might need to update your script.
- The function `copy` can no longer copy the code generation properties of model-owned data objects. You can instead use the new `clone` function to create a copy of an object with its code generation properties. You can use the `clone` function to create a copy in the same workspace as the source object.

You can still use existing command-line functions for configuring model-owned data objects, but it is recommended that you use the new code mappings API instead. The code mappings API:

- Provides one programming interface for configuration of default code generation settings for categories of data and individual data elements.
- Supports multiple configuration mappings for a model.
- Eliminates the need to create data objects to configure model data elements for code generation.

For information about the new code mappings API, see `coder.mapping.api.CodeMapping`.

Code Generation

Static code metrics for C99 and C++ libraries

In R2020a, you could generate a static code metrics report when the code generation language was C and the code contained the header files from the language version ISO[®]/IEC 9899:1990. In R2020b, you can generate a static code metrics report when the code contains header files from the language version ISO/IEC 9899:1999 also.

In R2020a, you could generate a static code metrics report when code generation language was C++ and the code contained no references to external header files. In R2020b, you can generate a static code metrics report when the code contains standard header files from the language version ISO/IEC 14882:2011 also.

Code generation using multiple code replacement libraries

In R2020a, you could generate code by using only a single code replacement library. In R2020b, you can generate code that has customizations and capabilities from multiple code replacement libraries. Multiple Code Replacement libraries can be selected in a unified workflow so that the generated code contains optimizations from varied code replacement libraries such as AUTOSAR 4.0 and Intel SSE. You can also select your own custom code replacement libraries along with shipped libraries to further optimize the generated code. For more information, see Optimize Generate Code by Using Multiple Code Replacement Libraries.

Compatibility Considerations

Before R2020b	R2020b
When registering a code replacement library, a comma was permitted in the code replacement library name.	Comma cannot be used in the code replacement library name.
When the <code>CodeReplacementLibrary</code> specifies a standard math library, for instance through its <code>BaseTfl</code> , and if it did not match the <code>TargetLangStandard</code> parameter, the standard math library specified by the <code>CodeReplacementLibrary</code> parameter was used.	If there is a mismatch, the standard math libraries specified by the <code>TargetLangStandard</code> parameter will be used.

Static reusable subsystem functions for C++ class interface

In R2020a and earlier releases, when generating C++ class interface code, the code generator generated reusable subsystem functions as private member functions inside a model class.

In R2020b, the code generator generates reusable subsystem functions that do not access class internals as static private member functions inside a model class when either of the following is true:

- There is no call to another function inside the reusable subsystem functions.
- If there is a call to another function, it must be a call to another reusable subsystem function or a shared utility function in `slprj/ert/_sharedutils`.

Generating reusable subsystem functions as static inside the model class reduces some the MISRA C++ 2008 Rule 9-3-3 violations. For more information, see MISRA C++:2008 Rule 9-3-3.

Name mangling of functions inside MATLAB Function block code

In R2020a and earlier releases, the code generator generated uncompileable code if the model had MATLAB Function block code with the following function names:

- `fclose`
- `feof`
- `ferror`
- `fgetl`
- `fgets`
- `fopen`
- `fprintf`
- `fread`
- `frewind`
- `fscanf`
- `fseek`
- `ftell`
- `fwrite`
- `sprintf`
- `strcmpi`
- `strncmpi`
- `strtok`
- `strcat`

These are also C function names. When MATLAB Function block code used these function names, the generated code clashed with the C function names during compilation and caused an error.

In R2020b, when generating C code from a model that contains a MATLAB Function block and uses the listed function names inside it, the code generator mangles the function names in the generated code so that the code compiles without errors. In name mangling, the code generator adds an extra character to make the identifiers unique so that the identifiers no longer match the C function names.

Generated code enhanced to reduce MISRA C:2012 Rule 13.5 violations

In R2020b, the code generator produces code that reduces some violations of the MISRA C:2012 Rule 13.5. For more information, see MISRA C:2012 Rule 13.5.

Generate static code metrics report programmatically

In R2020b, you can use the new function `coder.report.generateCodeMetrics` to generate a static code metrics report after generating code for a model. You no longer need to generate a code generation report.

Code generation and SIL or PIL simulations for protected models from R2018b and later releases

In R2020b, you can:

- Generate code from protected models created in previous releases (R2018b and later).
- Perform numeric equivalence testing for generated code by running software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

If a user of your protected model upgrades from a previous release to the current release, you do not have to regenerate the protected model for the user.

R2020b supports the new functionality for protected models that specify these system target file types:

- ERT
- ERT-based
- AUTOSAR Classic
- GRT (Embedded Coder required)

For more information, see [Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code](#).

Cross-release code integration for non-finite numbers in shared utility code

For cross-release code integration workflows, R2020b supports the use of non-finite numbers in shared utility code. Previously, if shared utility code contained non-finite numbers, the `sharedCodeUpdate` function produced a warning. For example:

The following files in `folderPath/slprj/ert/_sharedutils` are not allowed in the existing shared code area and will not be copied:

```
rtGetInf.c
rtGetInf.h
rtGetNaN.c
rtGetNaN.h
rt_nonfinite.c
rt_nonfinite.h
```

Continue yes/no:

For more information, see [Cross-Release Shared Utility Code Reuse](#).

Enhanced traceability between variables and modeling elements in Code view

Previously, bidirectional traceability between block output signals and the variables in the generated code was not available. In R2020b, you can obtain bidirectional traceability between block output signals and the variables in the generated code. This traceability is available only in Code view. For example, consider `rtwdemo_comments` model. You can trace the signal `indexed_data` by clicking

the signal in the model. You can then view the highlighted variable in the generated code.

Copyright 1994-2020 The MathWorks, Inc.

```

92  is_data_xor = (((rtu.In1 + rtu.In2) + rtu.In3) + rtu.In4 == 1.0);
93
94  /* Product: '<Root>/Product' incorporates:
95   * Constant: '<Root>/Constant'
96   * Inport: '<Root>/In1'
97   * Inport: '<Root>/In2'
98   * Inport: '<Root>/In3'
99   * Inport: '<Root>/In4'
100  */
101  /* Block description for '<Root>/Product':
102   * This block determines the output index by taking the inner product of
103   * the vector of inputs and [1 2 3 4].
104  */
105  indexed_data = ((rtu.In1 * const_val[0] + rtu.In2 * const_val[1]) + rtu.In3 *
106  const_val[2] + rtu.In4 * const_val[3]);
107
108  /* Chart: '<Root>/Stateflow'
109   *
110   * Block description for '<Root>/Stateflow':
111   * Stateflow block decides the index output by following logic
112   * 1. If one and only one input signal is enabled, the index will
113   * refer to the enabled signal line.
114   * 2. Otherwise, maintain the old value.
115  */

```

- ▶ Simulink Block
- ▶ Stateflow State
- ▶ Stateflow Transition
- ▶ Simulink Signal Object
- ▶ Simulink Parameter Object
- ▶ MPT Signal Object
- ▶ MPT Callback Function
- ▶ Open International Example
- ▶ Open Comments Settings
- ▶ Comments Documentation
- ▶ Customize banners via code templates

Generate Code Using Embedded Coder (double-click)

DOC Text History

For more information, see [Verify Generated Code by Using Code Tracing](#).

Same name error message for Simulink.Bus object and data in C++ code

Previously, during C++ code generation, if the `Simulink.Bus` object and its corresponding data element, such as signal or block, had the same name, the code generator produced a warning. The generated code was not compilable.

In R2020b, during C++ code generation, if the `Simulink.Bus` object and its corresponding data element have the same name, the code generator produces an error.

Standardization of header guards in header files

Previously, for models with shared utilities, header guards in the header files (shared files) were generated as:

```
#ifndef SHARE_MultiWordSignedWrap
#define SHARE_MultiWordSignedWrap
```

In R2020b, for models with shared utilities, header guards in the header files (shared files) are generated as:

```
#ifndef RTW_HEADER_MultiWordSignedWrap_h_
#define RTW_HEADER_MultiWordSignedWrap_h_
```

Deployment

Texas Instruments C2000: Support of UDP and Hardware Interrupt Blocks for F2838x (ARM Cortex-M4) Processor in Embedded Coder Support Package for Texas Instruments C2000 Processors

F2838x (ARM Cortex-M4) processor now supports UDP Send, UDP Receive, and Hardware Interrupt blocks in Embedded Coder Support Package for Texas Instruments C2000 Processors.

Texas Instruments C2000: Support Code Generation for SDFM Module in F2807x, F2837x, F28004x and F2838x Processors for Embedded Coder Support Package for Texas Instruments C2000 Processors

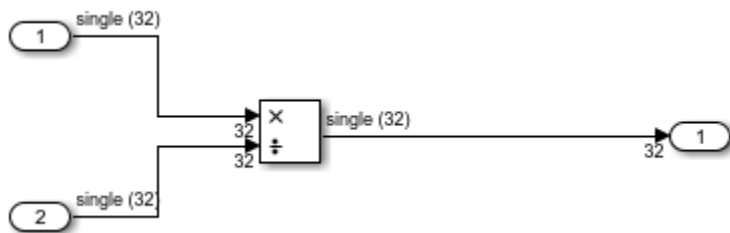
Embedded Coder Support Package for Texas Instruments C2000 Processors supports code generation for Sigma Delta Filter Module (SDFM) in F2837x, F2807x, F28004x, and F2838x processors.

Performance

SIMD code generated using Intel AVX-512 code replacement library

In R2020a, you generated code using the Intel AVX code replacement library, which enables the generated code to process 256 bits of data in parallel on shared-memory, multicore platforms. In R2020b, you can generate code for models and MATLAB code by using the improved Intel AVX-512 code replacement library. The generated code processes 512 bits of data in parallel. The increase in the register size of the data processed, improves execution speed. To generate code, in the Configuration Parameters dialog box, set the **Code replacement library** parameter by clicking **Select** and adding Intel AVX-512(Windows) or Intel AVX-512(Linux) to the **Selected code replacement libraries - prioritized** list pane.

Consider this model mDiv with a Divide block.



In R2020a, when you chose an Intel AVX code replacement library, the `mDiv_step` function contained this code:

```
void mDiv_step(void)
{
    int32_T i;
    for (i = 0; i <= 24; i += 8) {
        _mm256_storeu_ps(&mDiv_Y.Out2[i], _mm256_div_ps(_mm256_loadu_ps
            (&mDiv_U.In1[i]), _mm256_loadu_ps(&mDiv_U.In2[i])));
    }
}
```

The function `_mm256_div_ps` was able to process 256 bits of data in parallel.

In R2020b, when you choose an Intel AVX-512 code replacement library, the `mDiv_step` function contains this code:

```
void mDiv_step(void)
{
    int32_T i;
    for (i = 0; i <= 16; i += 16) {
        _mm512_storeu_ps(&mDiv_Y.Out2[i],
            _mm512_div_ps(_mm512_loadu_ps
                (&mDiv_U.In1[i]), _mm512_loadu_ps(&mDiv_U.In2[i])));
    }
}
```

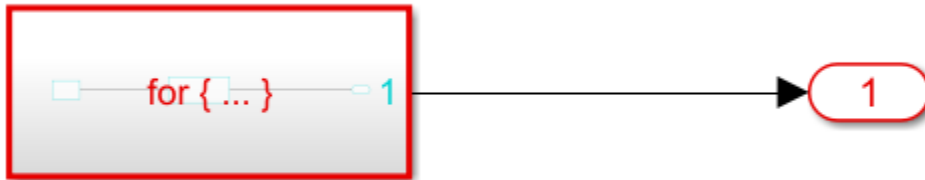
The function `_mm512_div_ps` processes 512 bits of data in parallel. This increase in the number of bits processed in parallel improves the execution speed of the code.

For more information, see [Generate Code Containing Single Instruction Multiple Data for Simulink Models](#) and [Generate Code Containing Single Instruction Multiple Data for MATLAB Code](#).

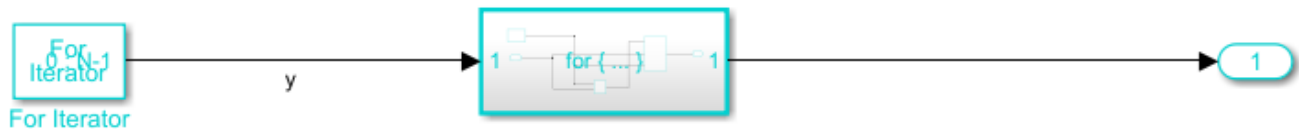
Improved cache performance of generated code that has loop interchange

In R2020a, the generated code contained loops as modelled in Simulink. In R2020b, you can generate optimized code that can interchange the order of execution of loops with constant bounds. This loop interchange avoids cache misses by accessing data from a single cache block. This optimization improves the execution speed of generated code for Intel and ARM targets.

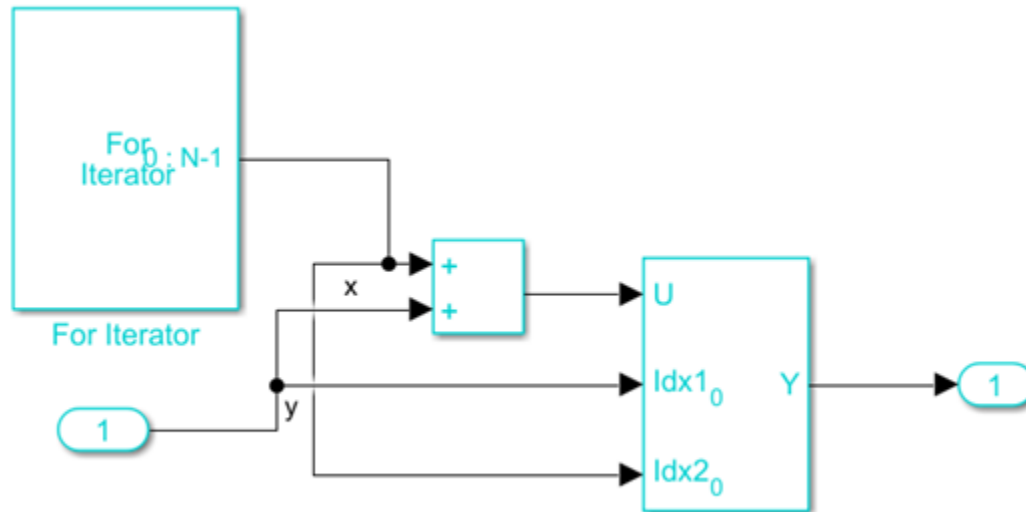
Consider this model mdl_loopinterchange that has a For-iterator Subsystem.



The For-iterator Subsystem shown below has another nested for-iterator subsystem. The top level For-iterator Subsystem has a For-iterator block with an **Iteration limit (N)** of 600.



The nested For-iterator Subsystem shown below has a For-iterator block with an **Iteration limit (N)** of 800.



In R2020a, the code generator produced this code.

```
void mdl_loopinterchange_step(void)
{
    int32_T y;
    int32_T x;

    /* Outputs for Iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
    * ForIterator: '<S1>/For Iterator'
    */
    for (y = 0; y < 600; y++) {
        /* Outputs for Iterator SubSystem: '<S1>/For Iterator Subsystem' incorporates:
        * ForIterator: '<S2>/For Iterator'
        */
        for (x = 0; x < 800; x++) {
            /* Assignment: '<S2>/Assignment' incorporates:
            * Output: '<Root>/Out1'
            * Sum: '<S2>/Add'
            */
            mdl_loopinterchange_Y.Out1[y + 600 * x] = x + y;
        }

        /* End of Outputs for SubSystem: '<S1>/For Iterator Subsystem' */
    }

    /* End of Outputs for SubSystem: '<Root>/For Iterator Subsystem' */
}
```

The signal x was present in the innermost subsystem of the Simulink model. Hence the generated code contained variable x as the innermost loop.

In R2020b, with the loop interchange, the code generator produces this code.

```
void mdl_loopinterchange_step(void)
{
    int32_T y;
    int32_T x;

    /* Outputs for Iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
    * ForIterator: '<S1>/For Iterator'
    */
    /* Outputs for Iterator SubSystem: '<S1>/For Iterator Subsystem' incorporates:
    * ForIterator: '<S2>/For Iterator'
    */
```

```

for (x = 0; x < 800; x++) {
  for (y = 0; y < 600; y++) {
    /* Assignment: '<S2>/Assignment' incorporates:
     * Output: '<Root>/Out1'
     * Sum: '<S2>/Add'
     */
    mdl_loopinterchange_Y.Out1[y + 600 * x] = x + y;
  }
}

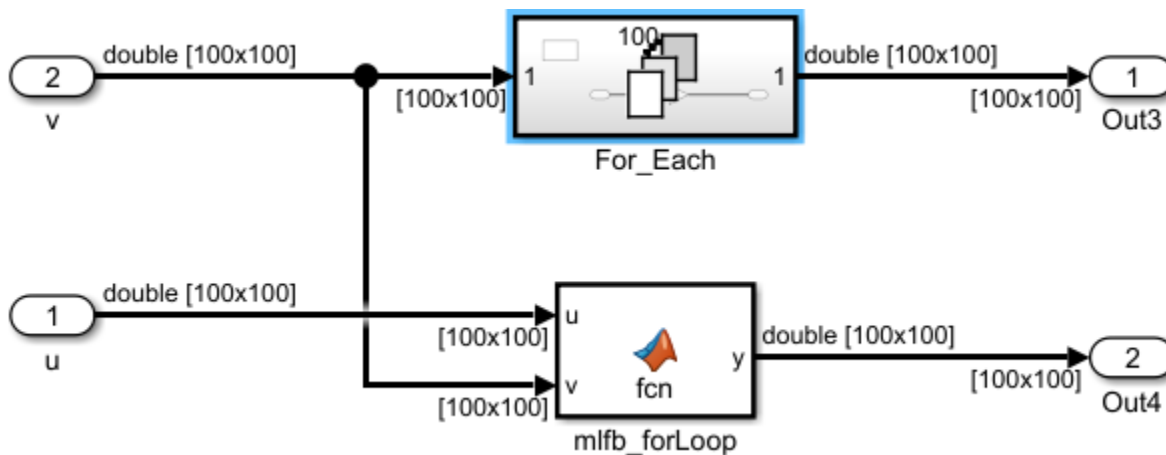
```

The signal x is present in the innermost subsystem of the Simulink model. But the variable y is evaluated in the inner loop of the generated code, allowing a sequential access of the memory that avoids cache misses. This optimization improves the execution speed of the generated code.

SIMD vectorization of loops in Simulink models

In R2020b, the generated code contains SIMD optimizations for MATLAB Function blocks and For-Each Subsystem blocks that contains for-loops. SIMD vectorizations improve speed and efficiency in the generated code.

Consider the model `mLoopVectorization` that has a For Each Subsystem and a MATLAB Function block.



The For Each Subsystem contains a Gain block that executes 100 times. The MATLAB Function block contains code with a for-loops that performs element-wise operations:

```

function y = fcn(u, v)
y = coder.nullcopy(u);

for i = 1:numel(u)
    y(i) = (u(i) .* v(i)) + (u(i) .* v(i));
end

end

```

In R2020a, the code generator produced this C code for the MATLAB Function and the For-Each Subsystem block:

```

for (i = 0; i < 10000; i++) {
    Out4_tmp = mloopVectorization_U.u[i] * mloopVectorization_U.v[i];
    mloopVectorization_Y.Out4[i] = Out4_tmp + Out4_tmp;
}

```



```

for (i = 0; i < 100; i++) {
for (ForEach_itr = 0; ForEach_itr < 100; ForEach_itr++) {
/* Gain: '<S1>/Gain' incorporates:
* ForEachSliceSelector generated from: '<S1>/In1'
*/
Out3_tmp = 100 * i + ForEach_itr;
mloopVectorization_Y.Out3[Out3_tmp] = mloopVectorization_U.v[Out3_tmp] *
57.0;
}
}

```

The loop incremented by one for single, double, and integer data types.

In R2020b, the code generator produces this snippet of vectorized code for the MATLAB Function block and, For-Each Subsystem block when you select the Intel SSE code replacement library:

```

for (i = 0; i <= 9998; i += 2) {
/* Inport: '<Root>/u' incorporates:
* MATLAB Function: '<Root>/mlfb_forLoop'
*/
tmp_0 = _mm_loadu_pd(&mloopVectorization_U.u[i]);

/* Inport: '<Root>/v' incorporates:
* MATLAB Function: '<Root>/mlfb_forLoop'
*/
tmp_1 = _mm_loadu_pd(&mloopVectorization_U.v[i]);

/* MATLAB Function: '<Root>/mlfb_forLoop' */
_mm_storeu_pd(&mloopVectorization_Y.Out4[i], _mm_add_pd(_mm_mul_pd(tmp_0,
tmp_1), _mm_mul_pd(tmp_0, tmp_1)));
}
for (i = 0; i < 100; i++) {
for (ForEach_itr = 0; ForEach_itr <= 98; ForEach_itr += 2) {
/* Gain: '<S1>/Gain' incorporates:
* ForEachSliceSelector generated from: '<S1>/In1'
*/
tmp = 100 * i + ForEach_itr;
_mm_storeu_pd(&mloopVectorization_Y.Out3[tmp], _mm_mul_pd(_mm_loadu_pd
(&mloopVectorization_U.v[tmp]), _mm_set1_pd(57.0)));
}
}
}

```

The loop increments by two because the input data type is `double`. Incrementing by two instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `int64`, the loop increments by two. If the data type is `single` or `int32`, the loop increments by four. This optimization increases the execution speed of the generated code. For more information, see [Generate Code Containing Single Instruction Multiple Data for Simulink Models](#) and [Generate Code Containing Single Instruction Multiple Data for MATLAB Code](#).

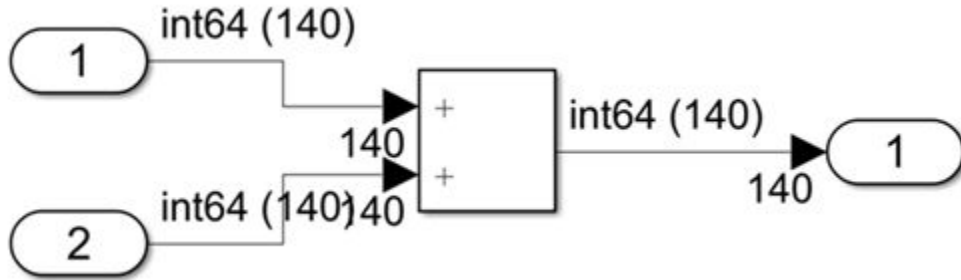
The generated code does not contain SIMD optimization if the **Partition Dimension** parameter of a For Each subsystem is below the **Loop unrolling threshold** configuration parameter.

Generated code optimization through SIMD for integer data type

In R2020a, the generated code contained SIMD optimizations for single precision and double precision data types. In R2020b, for Intel SSE or AVX processors, the generated code for models contains SIMD optimizations for 32 and 64 bit integer data types. The generated code from MATLAB code can contain SIMD optimizations for 8-, 16-, 32- and, 64- bit integer data types.

To generate the code, select an Intel SSE or AVX code replacement library.

Consider this Simulink model `mAdd` that has an Add block:



In R2020a, the code generator produced this C code:

```
void mAdd_step(void)
{
    int32_T i;

    /* Output: '<Root>/Out2' incorporates:
    * Inport: '<Root>/In1'
    * Inport: '<Root>/In2'
    * Sum: '<Root>/Add'
    */
    for (i = 0; i < 140; i++) {
        mAdd_Y.Out2[i] = mAdd_U.In1[i] + mAdd_U.In2[i];
    }

    /* End of Output: '<Root>/Out2' */
}
```

The loop incremented by one for the variable *i*.

In R2020b, the code generator produces this SIMD vectorized code for the Intel SSE code replacement library:

```
void mAdd_step(void)
{
    int32_T i;
    for (i = 0; i <= 138; i += 2) {
        /* Output: '<Root>/Out2' incorporates:
        * Inport: '<Root>/In1'
        * Inport: '<Root>/In2'
        */
        _mm_storeu_si128((__m128i *)&mAdd_Y.Out2[i], _mm_add_epi64(_mm_loadu_si128
            ((__m128i *)&mAdd_U.In1[i]), _mm_loadu_si128((__m128i *)&mAdd_U.In2[i]]));
    }
}
```

The loop increments by two because the input data type is `int64`. Incrementing by two instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `int32`, the loop increments by four. This optimization increases the execution speed of the generated code. For more information, see *Generate Code Containing Single Instruction Multiple Data for Simulink Models* and *Generate Code Containing Single Instruction Multiple Data for MATLAB Code*.

Enhanced Image Processing Toolbox functions in generated code

In R2020b, if possible, you can generate improved C and C++ code from MATLAB code and models containing MATLAB Function and MATLAB System blocks, containing the functions in the Image Processing Toolbox, for embedded targets. The optimizations available are multithreading, data parallelization, and SIMD code generation. These enhancements enable you to improve the speed of function execution.

To enable multithreading, in the Embedded Coder app, select the parameters **Specify custom optimizations** and **Generate parallel for loops**.

To enable SIMD, in the Embedded Coder app, set the **Code replacement library** parameter to Intel SSE or Intel AVX.

The optimized functions are:

- imwarp
- edge,
- medfilt2
- multithresh
- imresize
- regionprops
- imhist
- imopen
- imclose
- imdilate
- imerode
- rgb2ycbcr
- ycbcr2rgb
- houghlines
- hough

The edge function supports SIMD code generation for Intel SSE and AVX processors.

In R2020a, the code generator produced this C code snippet for a MATLAB function containing an image processing function `edge`:

```
...
for (j = 0; j < 1920; j++) {
    memset(&cj[0], 0, 1080U * sizeof(float));
    for (jb = 0; jb < 3; jb++) {
        for (ib = 0; ib < 3; ib++) {
            bij = b[(3 * (2 - jb) - ib) + 2];
            for (i = 0; i < 1080; i++) {
                cj[i] += (float)bij * a[(i + ib) + 1082 * (j + jb)];
            }
        }
    }
    memcpy(&c[j * 1080], &cj[0], 1080U * sizeof(float));
}
...
```

The loop executed sequentially one incremented value at a time.

In R2020b, the code generator produces this snippet of code:

```
...
for (ix = 0; ix < 1920; ix++) {
    for (i = 0; i <= 1064; i += 8) {
        r = _mm256_loadu_ps(&temp[i + 1080 * ix]);
        bx_tmp = (i + 1080 * ix) + 1;
        r1 = _mm256_loadu_ps(&temp[bx_tmp]);
        r2 = _mm256_loadu_ps(&temp[(i + 1080 * ix) + 2]);
```

```

    _mm256_storeu_ps(&by[bx_tmp], _mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(r,
    _mm256_set1_ps(0.465302438F)), _mm256_mul_ps(r1, _mm256_set1_ps(0.0F))),
    _mm256_mul_ps(r2, _mm256_set1_ps(-0.465302438F)));
}
...

```

The loop increments by 8. Incrementing by eight instead of one occurs because the SIMD functions in the loop body process data in parallel. This optimization improves the execution speed of the generated code.

Distribution of execution times for generated code internal functions

If the generated code for your model contains nested functions, you can run software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations that generate pie charts showing the relative execution times of caller and called functions. The pie charts, which show average and maximum execution time distributions, can help you to identify functions that are bottlenecks in code execution.

For more information, see [View and Compare Code Execution Times](#).

Hardware timer for code execution profiling during PIL simulations

For code execution profiling during processor-in-the-loop (PIL) simulations, you can create a timer object by using the target package. For more information, see step 6 in [Set Up PIL Target Connectivity by Using target Package](#).

Caching of array elements to scalar variables reduces computations in generated code

In R2020a, for some modeling patterns, the generated code contained additional computations of an array element's address. These computations occurred for code that read the same array element at multiple locations and did not use a constant value to index into the array.

In R2020b, the code generator is more likely to cache these array elements into scalar variables that can be read at multiple locations in the generated code reducing the number of computations.

This table shows one example.

R2020a	R2020b
<code>return (table[iLeft + 1U] - table[iLeft]) * frac + table[iLeft];</code>	<code>return (table[iLeft + 1U] - yL_0d0) * frac + yL_0d0;</code>

In the R2020a, array elements are not cached into scalar variables. In R2020b, the array element `table[iLeft]` is cached in the variable `yL_0d0` that is read twice.

This table shows another example.

R2020a	R2020b
<pre>for (jy = 0; jy <= iy; jy++) { kBcol = 6 * jy - 1; j_0 = (jy + jj) + 1; if (A[j_0] != 0.0) { for (ix = 0; ix < 6; ix++) { ijA = ix + 1; c = ijA + jj; y[c] -= A[j_0] * y[ijA + kBcol]; } } }</pre>	<pre>for (jy = 0; jy <= iy; jy++) { kBcol = 6 * jy - 1; smax = A[(jy + jj) + 1]; if (smax != 0.0) { for (ix = 0; ix < 6; ix++) { c = (ix + jj) + 1; y[c] -= smax * y[(ix + kBcol) + 1]; } } }</pre>

In R2020a, just the array index value was cached in a variable `j_0`. In R2020b the array element `A[(jy + jj) + 1]` is cached in the variable `smax` that is read twice, including in a nested loop.

Verification

Target connectivity for PIL simulations

You can set up target connectivity for processor-in-the-loop (PIL) simulations by using the `target` Package, which provides new classes for registering connection and communication details:

- `target.Board`
- `target.CommunicationInterface`
- `target.TargetConnection`
- `target.Timer`

The classes enable you to define concisely your target hardware. To register target connectivity, use the `target.create`, `target.get`, and `target.add` classes. You do not need `rtwTargetInfo.m` or `sl_customization.m` files for the registration process.

For more information, see [Set Up PIL Target Connectivity by Using target Package](#).

SIL and PIL testing of reusable library subsystems

Create test harnesses for reusable library subsystems and use the SIL/PIL Manager to test pregenerated code.

- 1 Create a test harness in a library for a unique subsystem and function interface pair.
- 2 Using the SIL/PIL Manager:
 - a Run normal mode and SIL/PIL simulations of the subsystem.
 - b Compare numeric results in the Simulation Data Inspector.
 - c View the Simulink Coverage analysis report.

For more information, see:

- [Test Library Blocks \(Simulink Test\)](#)
- [Library-Based Code Generation for Reusable Library Subsystems](#)

Signal and state logging for SIL and PIL simulations

For top-model and Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations, you can log:

- Signals by setting the `SignalLogging` configuration parameter of the top model to 'on'. Previously, signal logging was supported through the creation of a C API data interface.
- State data by setting the `SaveState` configuration parameter of the top model to 'on'.

In the SIL/PIL Manager, you can test the numerical equivalence between a model and generated code by using logged signals and state data.

For more information, see [Log Signals of a Component](#) and [SIL/PIL Manager Verification Workflow](#).

Removal of top-model SIL and PIL limitations

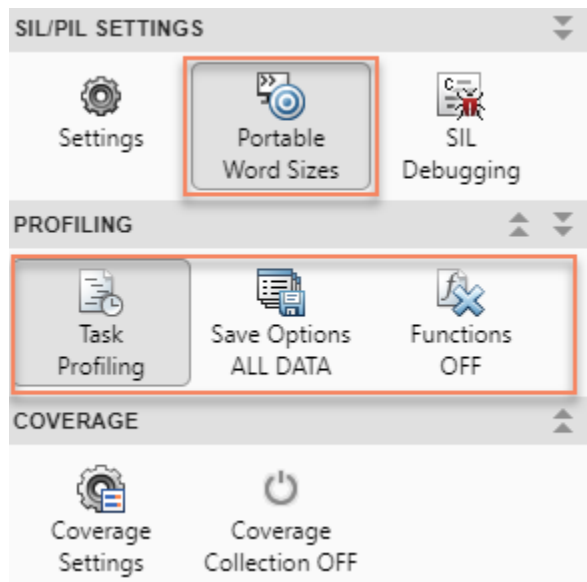
R2020b removes previous limitations of top-model SIL and PIL simulations by providing:

- Support for bus element ports, inline variants, and Dataset logging for export-function models.
- Enhanced support for Initialize Function, Reset Function, and Terminate Function blocks of models.

For more information, see [SIL and PIL Limitations](#).

SIL/PIL Manager settings

The SIL/PIL Manager **Settings** gallery has new buttons.



The **Profile Code** button is removed.

Button	Description
Portable Word Sizes	Toggles PortableWordSizes between 'on' or 'off'.
Task Profiling	Toggles CodeExecutionProfiling between 'on' or 'off'.
Save Options	If you select Task Profiling , clicking this button sets CodeProfilingSaveOptions cyclically to 'SummaryOnly', 'AllData', and 'MetricsOnly'. If you do not select Task Profiling , the button is dimmed.
Functions	Sets CodeProfilingInstrumentation cyclically to 'off', 'coarse', and 'detailed'.

For more information, see [SIL/PIL Manager Verification Workflow and Run SIL Simulation That Generates Execution-Time Metrics](#).

Functionality being removed or changed

silblocktype function produces an error for legacy argument

Errors

In R2020b, running the command `silblocktype('legacy')` produces this error:

The 'legacy' SIL block type is no longer supported.
Use only the 'unified' SIL block type.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2020a

Version: 7.4

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Model Architecture and Design

Function arguments to match graphical block interface for nonreusable subsystems

When generating C/C++ code for nonreusable subsystems, you can specify the function interface in the generated code to use arguments that match the graphical interface of the subsystem block. The arguments represent the input and output ports of the subsystem. This specification generates a predictable interface that can be useful for testing, debugging, and integrating with external code.

To match the function arguments with the graphical interface of the subsystem block, in the Subsystem Block Parameters dialog box, on the **Code Generation** tab, set the **Function packaging** parameter to `Nonreusable function`. The **Function packaging** parameter enables the **Function interface** (Simulink) parameter. Set the **Function interface** parameter to the new `Allow arguments (Match graphical interface)` value. For example, if the subsystem block has four inputs and three outputs, the generated code also has four inputs and three outputs. For more information, see [Generate Predictable Function Interface to Match Graphical Block Interface](#).

To generate an optimized function that has arguments in the generated code, set **Function interface** to `Allow arguments (Optimized)` (previously named `Allow arguments`). The generated function that has arguments might not match the graphical interface of the subsystem block.

External I/O visibility for C++ class interface

In R2019b, when you set the model configuration parameter **Code interface packaging** to `C++ class`, the code generator produced the external input/output type definitions as `public` or `protected` members of the model class. In R2020a, you can configure the visibility of external input/output type definitions by using the new model configuration parameter **External I/O visibility**. Choose from these values:

- `public`
- `protected`
- `private` (default)

The default specification for the **External I/O access** parameter is now `Inlined structure-based method` (previously, `None`).

When you open an existing model saved in R2019b or earlier, the default specification for the **External I/O visibility** parameter is `public` if the **External I/O access** parameter is set to `None`. Otherwise, the default specification is `protected`.

The generated code reduces the MISRA C++ 2008 Rule 11-0-1 violations.

C++ message-based communication provides length argument for service functions

C/C++ message support now generates an additional parameter to specify message payload length in service functions. For more information, see [Generate C++ Code from Top Models for Message-Based Communication By Using External Message Protocols](#).

Zero initialization code model configuration parameters disabled for C++ class interface

Starting in R2020a, members of a C++ model class are initialized in the class constructor. For new and existing models, when the model configuration parameter **Code interface packaging** is set to C++ class, the check boxes of these model configuration parameters are selected and set to 'off' (command line):

- **Remove root level I/O zero initialization** (ZeroExternalMemoryAtStartup)
- **Remove internal data zero initialization** (ZeroInternalMemoryAtStartup)

You cannot change the values of these model configuration parameters.

Code Interface Configuration and Integration

Alias property of Simulink.CoderInfo renamed Identifier

The `Alias` property of the `Simulink.CoderInfo` object, which you use to specify an alternative name for a data object in the generated code, is renamed `Identifier`. For data objects with a non-`Auto` storage class:

- **Identifier** replaces **Alias** in user interfaces.
- At the command line, autocomplete provides `Identifier` instead of `Alias` as a property available for the `Simulink.CoderInfo` object.
- `Identifier` is saved to MATLAB files. `Alias` is still saved elsewhere, including in MAT-files.
- At the command line, you can use `Identifier` and `Alias` properties interchangeably. Setting one property results in the other property having the same value.

For more information on this property, see `Simulink.CoderInfo`.

Model type definitions within class namespace

In R2019b, when generating code for a C++ class interface, the code generator produced model type definitions in the global namespace.

In R2020a, when generating code for a C++ class interface, you can choose to generate the model type definitions within the class namespace. Select the new model configuration parameter **Include model types in model class**. When you open an existing model saved in R2019b or earlier, this parameter is cleared by default. When you create a new model and **Code interface packaging** is set to `C++ class`, this parameter is selected by default.

Model type definitions include:

- Root-level inports and outports
- Block inputs and outputs
- DWork vectors
- Block parameters and constant parameters
- Continuous states
- Real-time model data structure (`rtM`)

The generated code reduces the MISRA 7-3-1 violations.

User-defined type such as `Simulink.Bus` object or type defined in a MATLAB Function block or Stateflow charts is still generated in the global namespace.

Dimension preservation of multidimensional arrays for Data Store Memory blocks, states, and signals

In R2019b, when the model configuration parameter **Array layout** was set to `Row-major`, you could preserve dimensions for root-level Inport and Outport blocks, parameters, and lookup tables.

In R2020a, when the model configuration parameter **Array layout** is set to Row-major, you can preserve the dimensions of multidimensional array data used in Data Store Memory blocks, states, and signals.

From the Code Mappings editor, you can configure the default configurations to preserve dimensions of:

- **Shared local data stores**
- **Global data stores**
- **Internal data**

For these elements, select the **PreserveDimensions** property in the Property Inspector window when **Storage Class** is set to these supported storage classes:

- Volatile
- ExportToFile
- ImportFromFile
- FileScope (not supported for **Global data stores**)
- Localizable

In the Embedded Coder Dictionary, to preserve dimensions when you design your own custom storage class, select the **Preserve array dimensions** property in the Property Inspector. You can apply the supported storage class to these data default categories:

- Shared local data stores
- Global data stores
- Internal data

You can also select the **Preserve array dimensions** property in the data object interface.

For more details, see [Preserve Dimensions of Multidimensional Arrays in Generated Code](#).

Compatibility Considerations

Starting in R2020a, the model configuration parameter **Preserve Stateflow local data array dimensions** is not supported. When you open a model saved in R2019b and earlier releases, a warning is issued if this parameter is selected. In R2020a, select the **Preserve array dimensions** property for the **Internal data** data default category in the Code Mappings editor to preserve dimensions for the Stateflow local data.

Storage class change for model workspace parameter converted to Simulink.Parameter

Previously, in the Model Explorer or Model Data Editor, converting a parameter to a `Simulink.Parameter` object resulted in a parameter object configured with the storage class `Model default`. In R2020a, this conversion results in an object with the storage class set to `Auto`. To prevent code generation optimizations from eliminating storage for the variable, you can manually change the **Storage Class** setting to `Model default` or another storage class.

For more information, see [Choose Storage Class for Controlling Data Representation in Generated Code](#).

Functionality being removed or changed

TypeQualifier property for built-in storage classes is not recommended

Still runs

The property `TypeQualifier` for built-in storage classes, such as `ExportedGlobal` and `ImportedExtern`, is not recommended. When you specify the property, the code generator adds C qualifiers, such as `const` and `volatile`, to the beginning of data declarations and definitions. Instead of using the `TypeQualifier` property, configure data objects by using a storage class that specifies the C qualifier of interest (see [Choose Storage Class for Controlling Data Representation in Generated Code](#)). If none of the available storage classes meets your application data requirements, define a new storage class by using the [Embedded Coder Dictionary](#) (see [Define Storage Classes, Memory Sections, and Function Templates for Software Architecture](#)) or [Custom Storage Class Designer](#) (see [Create Storage Classes by Using the Custom Storage Class Designer](#)).

Simulink.CoderInfo object Alignment property for data configured within a model is not recommended

Still runs

The `Simulink.CoderInfo` object **Alignment** property is not recommended for data configured for code generation within a model. This includes data represented by:

- Outports
- Signal lines
- Block states
- Data stores
- Parameter objects in the model workspace

To use the **Alignment** property, represent data with data objects outside of the model. For example, do the following:

Type of Data Element	Action
Parameter object in the model workspace	Use the Model Explorer to move the object from the model workspace to the base workspace or a data dictionary. Then, set the storage class and alignment properties of the object. Save the model.
Signal line	Open the Signal Properties dialog box, specify a name for the signal and select Must resolve to signal object (storage class must be <code>Auto</code>). Create a <code>Simulink.Signal</code> object in the base workspace or a data dictionary and set the storage class and alignment properties of the object. Save the model.
Block state	Open the Block Parameters dialog box, specify a name for the state and select State must resolve to signal object (storage class must be <code>Auto</code>). Create a <code>Simulink.Signal</code> object in the base workspace or a data dictionary and set the storage class and alignment properties. Save the model.

Code Generation

std::array support in C++ code generation

In C++ 11, `std::array` is a template class that represents fixed-size arrays. When the model configuration parameter **Code interface packaging** is set to `C++ class`, you can generate a C++ class interface that uses an `std::array`, instead of C-style arrays, by specifying the new model configuration parameter **Array container type** as `std::array`.

You can also choose to preserve the C-style arrays by specifying **Array container type** as `C-style array`. This specification is the default.

In R2019b, the generated code contained C-style arrays:

```
real_T const_val[4] = { 1.0, 2.0, 3.0, 4.0 } ;
```

In R2020a, if specified, the generated code can contain an `std::array`:

```
std::array<real_T, 4> const_val = { { 1.0, 2.0, 3.0, 4.0 } };
```

Allow Arguments for non-reusable subsystems with C++

In R2019b, when you generated code for a C++ class interface and set **Function interface** (Simulink) to `Allow arguments`, the code generator produced a function that passed data as global variables. For example:

```
HeadingMode();
```

In R2020a, when you generate code for a C++ class interface and set **Function interface** to `Allow arguments (Optimized)` or `Allow arguments (Match graphical interface)`, the code generator produces a function that uses arguments instead of passing data as global variables. For example, this code is the generated code when **Function interface** is set to `Allow arguments (Optimized)`:

```
HeadingMode(rtU.HDG_Ref, rtU.Psi, rtU.TAS, &rtb_Sum1);
```

For more details on `Allow arguments (Match graphical interface)`, see the release note on “Function arguments to match graphical block interface for nonreusable subsystems” on page 6-3.

\$R token in Memory Sections of Embedded Coder Dictionary

In R2019b, when you created memory sections in Embedded Coder Dictionary, the `$N` token was the only supported token available to use in **Pre Statement** and **Post Statement** properties.

In R2020a, when you create memory sections in Embedded Coder Dictionary, you can use the `$R` token and the `$N` token in **Pre Statement** and **Post Statement** properties. `$R` expands to the model name and `$N` expands to the name of the data element or function. Use only one instance of `$R` and `$N` in a specification. You can use the `$R` token when the **Statements Surround** property is set to `Each variable` or `Group of variables`. For more information, see Embedded Coder Dictionary.

`$R` is not supported for memory sections that you create by using the Custom Storage Class Designer.

Reduction in identifier collisions in model reference hierarchy

In R2019b, when you generated code for a model reference hierarchy, all global variables and types were exported from child models to the top model. This export process sometimes resulted in identifier clashes. For example, the variable name `StorageClass2_mModelRefAutoData` was used in the top model and not the child model, but the code generator inserted name mangling:

```
/* Storage class 'StorageClass2' */
mModelRefAutoDa_StorageClass2_n StorageClass2_mModelRefAutoDa_n; /* '<Root>/Model
```

In R2020a, only used global variables and types are exported from child models to the top model. When you generate code for a model reference hierarchy, you might see fewer identifier collisions and better naming for global variables and types. Also, the generated code is more readable. For example, the variable name `StorageClass2_mModelRefAutoData` is used only in the top model and the code generator does not insert name mangling:

```
/* Storage class 'StorageClass2' */
mModelRefAutoDa_StorageClass2_n StorageClass2_mModelRefAutoData; /* '<Root>/Model
```

Static code metrics in Code view without code generation report

In R2020a, you can generate static code metrics for your generated code without generating a code generation report. To generate static code metrics, select model configuration parameter **Generate static code metrics** and then generate code. Previously, the configuration parameter **Static code metrics** was disabled when you cleared the **Create code generation report** parameter.

In R2020a, the **Static code metrics** parameter is renamed to **Generate static code metrics** and does not depend on the **Create code generation report** parameter. To view the code metrics for a variable or function, place your cursor over the variable or function in the Code view. For more information, see [View Static Code Metrics and Definitions Within the Generated Code](#).

In R2020a, when you generate code by using the Embedded Coder app, the Code view opens by default to display the generated code next to your model. The default values for ERT-based targets have changed from On to Off in these configuration parameters:

- **Create code generation report**
- **Open report automatically**
- **Code-to-model**
- **Model-to-code**
- **Eliminated / virtual blocks**
- **Traceable Simulink blocks**
- **Traceable Stateflow objects**
- **Traceable MATLAB functions**

Compatibility Considerations

Previously, the static code metrics file `codeMetrics.mat` was generated in the `html` subfolder in the build folder. In R2020a, the `codeMetrics.mat` file is generated in the folder `s\prj\target\model\tmwinternal`.

SIL or PIL simulations with protected model AUTOSAR code from R2018b or later

If you have protected models that contain AUTOSAR code generated by using R2018b or a later release and the AUTOSAR code is generated with a Top model code interface, in R2020a, you can run Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations that reference the protected models.

If the AUTOSAR code in the protected model requires shared utility code, you can use `sharedCodeUpdate` to copy the required code to an existing shared utility code folder.

For more information, see [Reference a Protected Model \(Simulink\)](#).

Storage classes on signal lines

Previously, only built-in storage classes and storage classes that you created by using the Custom Storage Class Designer were displayed on signal lines in the block diagram. In R2020a, the block diagram also displays the names of storage classes that you create in an Embedded Coder Dictionary. When you apply the storage class `Model default` to a signal, the signal line displays the name of the default storage class for the internal data category in the form `<name>`. If the default storage class for internal data is unspecified (`Default`), the signal line does not display a storage class name. To display storage class names on signal lines, on the **Debug** tab, click **Information Overlays > Storage Class**. For more information, see [Apply Storage Classes to Individual Signal, State, and Parameter Data Elements](#).

Removal of preprocessor guards in C++ code

In R2019b, when generating C++ code, the code generator included preprocessor guards to check the inclusion of some common header files. For example:

```
#ifndef RTW_HEADER_rtwdemo_comments_h_
#define RTW_HEADER_rtwdemo_comments_h_
#ifdef rtwdemo_comments_COMMON_INCLUDES_
# define rtwdemo_comments_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif
```

In R2020a, when generating C++ code, the code generator removes these preprocessor guards to reduce MISRA 16-2-1 violations in the generated code. For example:

```
#ifndef RTW_HEADER_rtwdemo_comments_h_
#define RTW_HEADER_rtwdemo_comments_h_
#include "rtwtypes.h"
```

Removal of configuration parameter limitations for Simulink string code generation

In R2019b, to generate code with the `std::string` library instead of `C char_T` arrays you had to ensure that:

- You selected the **Use dynamic memory allocation for model initialization** parameter.

- You selected the **Use dynamic memory allocation for model block instantiation** parameter.
- You cleared the **Remove root level I/O zero initialization** parameter in the Configuration Parameters dialog box.
- You cleared the **Remove internal data zero initialization** parameter in the Configuration Parameters dialog box.

In R2020a, you can generate C++ code from model blocks by using the standard C++ string library without configuration parameter restrictions. You can set the parameters to any value. The generated C++ code contains functions and data types from the standard C++ string library.

For more information, see [Generate Code for String Blocks by using the Standard C++ String Library](#).

Deployment

FFT code replacement library (CRL) support for ARM Cortex-A and Cortex-M processors

In R2020a, you can generate optimized code for fast Fourier transform (FFT) algorithms by using the code replacement library (CRL). You get improved code performance for these MATLAB functions:

- `fft`
- `ifft`
- `fft2`
- `ifft2`
- `fftn`
- `ifftn`

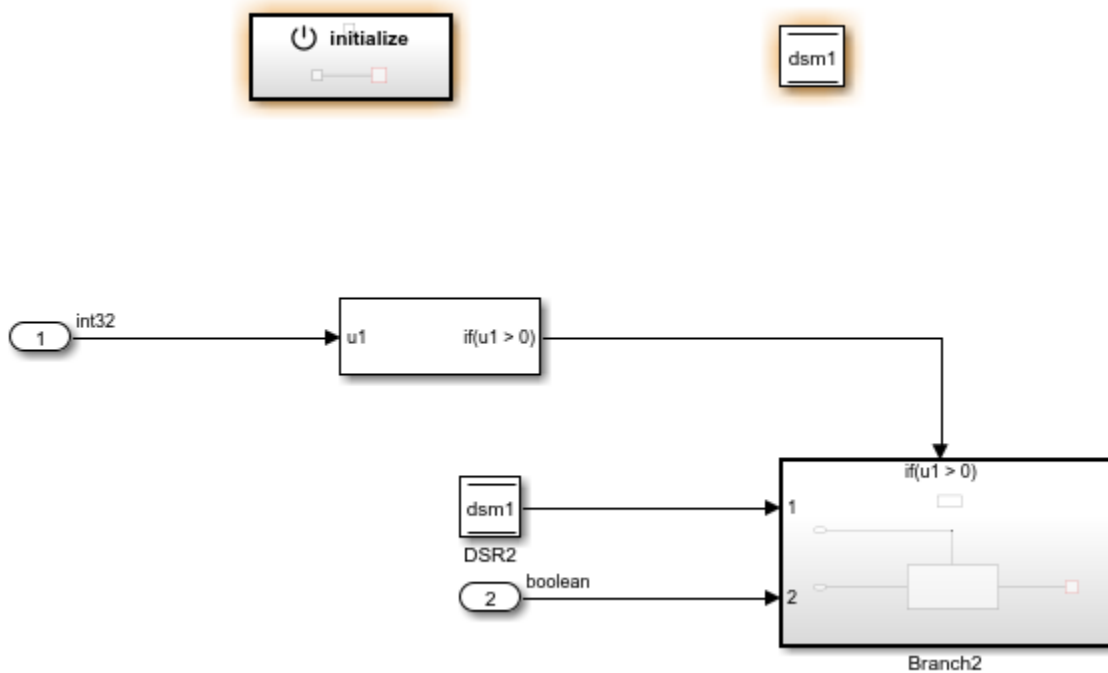
The generated code can now run on ARM Cortex-A and ARM Cortex-M processors. For more information on how to install and get started with these support packages, see [Embedded Coder Support Package for ARM Cortex-A Processors](#) and [Embedded Coder Support Package for ARM Cortex-M Processors](#).

Performance

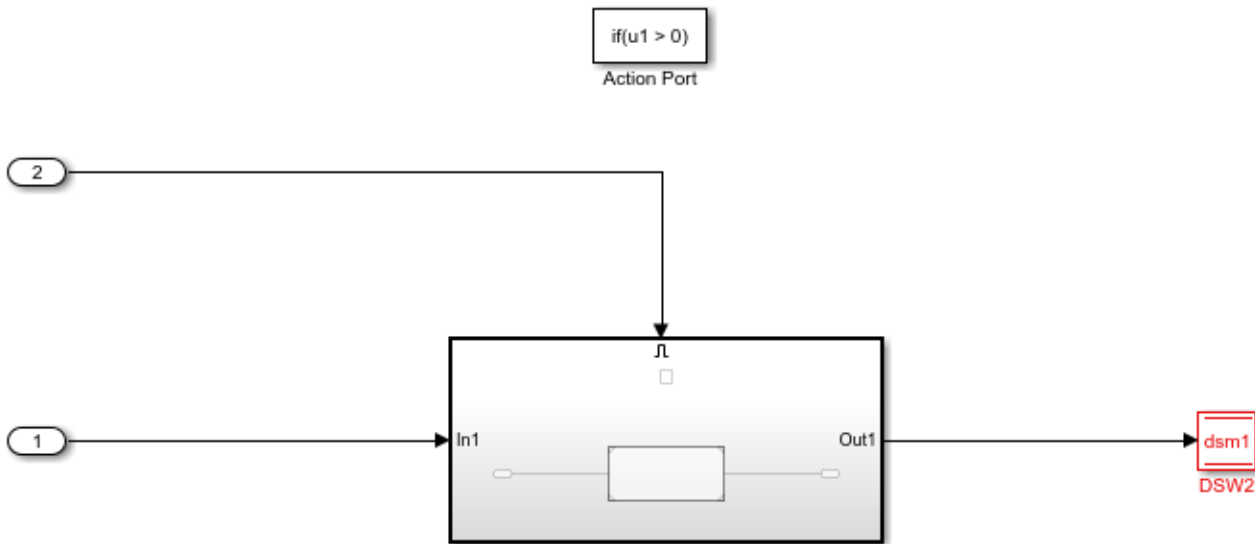
Data Store Memory block reuse to reduce data copies in subsystems

In R2019b, for models that used Data Store Memory blocks to store large bus structures, the generated code contained redundant data copies when the Data Store Memory blocks were read from and written to across the boundaries of subsystems. In R2020a, the code generator can eliminate redundant data copies across subsystem hierarchies when the data store read and write operation happens within subsystems. Eliminating the extra data copies reduces RAM and ROM consumption and improves execution speed.

Consider the model `MultipleDsrDsw` with a Data Store Read block that reads a bus structure from a Data Store Memory block called `dsm1`. The signal is input to a subsystem block `Branch2`.



Inside the subsystem `Branch2`, a bus is output from the Subsystem block and written by a Data Store Write block to the Data Store Memory `dsm1` in top-level model.



In R2019b, the code generator produced this code:

```
void mBusMultipleDsrDsw_step(RT_MODEL *const rtM, int32_T rtU_In1, boolean_T
rtU_In2)
{
    BusType1 rtb_DSR2;
    rtb_DSR2 = rtM->dwork.dsm1;
    if (rtU_In1 > 0) {
        if (rtU_In2) {
            rtM->dwork.mBusDsmBot1 = rtb_DSR2;
            mBusDsmBot_step((&(rtM->mBusDsmBot1)), &(rtM->dwork.mBusDsmBot1));
        }

        rtM->dwork.dsm1 = rtM->dwork.mBusDsmBot1;
    }
}
```

The code contained an unnecessary data copy to the variable `rtb_DSR2`.

In R2020a, the code generator produces this code:

```
void mBusMultipleDsrDsw_step(RT_MODEL *const rtM, int32_T rtU_In1, boolean_T
rtU_In2)
{
    if (rtU_In1 > 0) {
        if (rtU_In2) {
            rtM->dwork.mBusDsmBot1 = rtM->dwork.dsm1;
            mBusDsmBot_step((&(rtM->mBusDsmBot1)), &(rtM->dwork.mBusDsmBot1));
        }

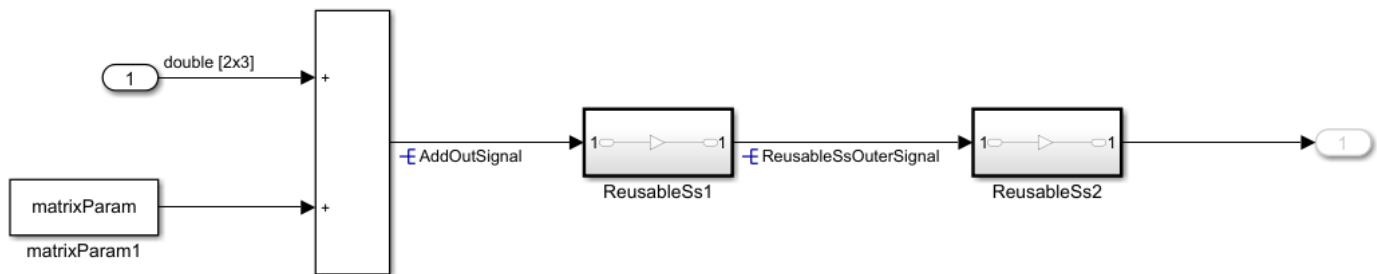
        rtM->dwork.dsm1 = rtM->dwork.mBusDsmBot1;
    }
}
```

The code does not contain the local variable `rtb_DSR2` and the data copy, which improves the efficiency of generated code. For more information, see [Data Copy Reduction for Data Store Read and Data Store Write Blocks](#).

Buffer reuse optimization for multidimensional arrays

You can reuse buffers for multidimensional arrays that you specify to preserve dimensions by selecting the model configuration parameter **Reuse local block outputs** (Simulink Coder) (BufferReuse).

Consider the model NDSignals_ReusableSS that has two reusable subsystems.



When you clear the model configuration parameter **Reuse local block outputs**, this code is generated:

```
void NDSignals_ReusableSS_step(void)
{
    real_T AddOutSignal[2][3];
    real_T ReusableSsOuterSignal[2][3];
    real_T rtb_Gain_e[6];
    int32_T i;
    int32_T i_0;
    for (i_0 = 0; i_0 < 2; i_0++) {
        for (i = 0; i < 3; i++) {
            AddOutSignal[i_0][i] = rtIn1[i_0][i] + matrixParam[i_0][i];
        }
    }

    ReusableSs1((&AddOutSignal[0][0]), rtb_Gain_e);
    for (i_0 = 0; i_0 < 2; i_0++) {
        for (i = 0; i < 3; i++) {
            ReusableSsOuterSignal[i_0][i] = rtb_Gain_e[3 * i_0 + i];
        }
    }

    ReusableSs1((&ReusableSsOuterSignal[0][0]), &rtOut1[0][0]);
}

```

The code generator uses additional buffer for the Gain parameter.

This is the generated code when the model configuration parameter **Reuse local block outputs** is selected:

```
void NDSignals_ReusableSS_step(void)
{
    real_T AddOutSignal[2][3];
    real_T ReusableSsOuterSignal[2][3];
    int32_T i;
    int32_T i_0;
    for (i_0 = 0; i_0 < 2; i_0++) {
        for (i = 0; i < 3; i++) {
            AddOutSignal[i_0][i] = rtIn1[i_0][i] + matrixParam[i_0][i];
        }
    }
}

```

```

    }
  }
  ReusableSs1((&AddOutSignal[0][0]), (&ReusableSsOuterSignal[0][0]));
  ReusableSs1((&ReusableSsOuterSignal[0][0]), &rtOut1[0][0]);
}

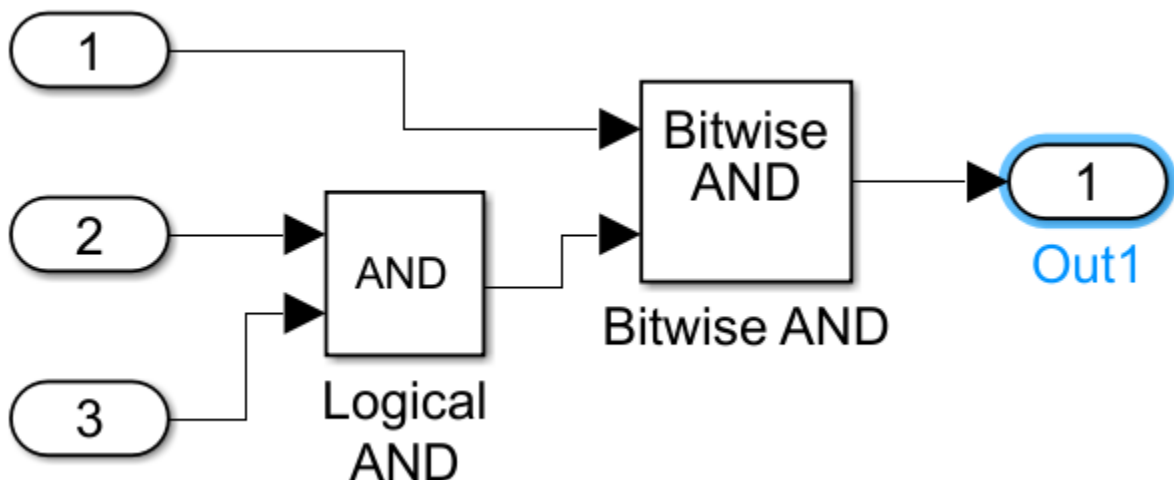
```

The code generator does not generate a separate buffer for the Gain variable and reuses the existing buffers.

Logical operators conversion to bitwise operators in generated code

In R2019b, in the generated code, bitwise operations in the model were represented as logical operators. In R2020a, you can generate code by using either bitwise or logical operators or a combination of both. Certain processors might improve ROM efficiency when the code contains bitwise operators. You can set the new configuration parameter **Operator to represent Bitwise and Logical Operator blocks** to enable this optimization.

Consider the model `logicalandbitwise` with a Logical AND block that connects to a Bitwise AND block.



To generate code, in the Configuration Parameters dialog box, set the **Operator to represent Bitwise and Logical Operator blocks** parameter. Choose from these settings:

- Same as modeled
- Logical operator
- Bitwise operator

In R2019b, the code generator produced this code:

```

void logicalandbitwise_step(void)
{
  /* Output: '<Root>/Out1' incorporates:
   * Inport: '<Root>/Input'
   * Inport: '<Root>/Input1'
   * Inport: '<Root>/Input2'
   * Logic: '<Root>/Logical AND'
   * S-Function (sfix_bitop): '<Root>/Bitwise AND'
   */
}

```

```

    logicalandbitwise_Y.Out1 = (logicalandbitwise_U.Input &&
        (logicalandbitwise_U.Input1 && logicalandbitwise_U.Input2));
}

```

The code contained only logical operators &&, where the Bitwise AND block in the model were cast as logical operators.

In R2020a, when you select the parameter setting `Same as modeled`, the code generator produces this code:

```

void logicalandbitwise_step(void)
{
    /* Output: '<Root>/Out1' incorporates:
    *   Inport: '<Root>/Input'
    *   Inport: '<Root>/Input1'
    *   Inport: '<Root>/Input2'
    *   Logic: '<Root>/Logical AND'
    *   S-Function (sfix_bitop): '<Root>/Bitwise AND'
    */
    logicalandbitwise_Y.Out1 = logicalandbitwise_U.Input &
        (logicalandbitwise_U.Input1 && logicalandbitwise_U.Input2);
}

```

The generated code contains both bitwise and logical operators in the generated code corresponding to the blocks in the models. Presence of bitwise operators might improve ROM efficiency. For more information, see `Control Operator Type in Generated Code`.

Enhanced Image Processing Toolbox functions in generated code

In R2019b, the generated portable C code did not support multi-threading for functions in the Image Processing Toolbox. In R2020a, you can generate code for image processing toolbox functions with multi-threading capabilities. This enhancement enables you to improve the speed of function execution.

The functions with the new multi-threading capability are `edge`, `imwarp`, `imrotate`, `imfilter`, `medfilt2`, `multithresh` and `rgb2gray`.

In R2019b, the code generator produced this C code snippet for a MATLAB function containing an image processing function `imwarp`:

```

void medfilt2(unsigned char b[65536])
{...
    for (j = 0; j < 384; j++) {
        for (i = 0; i < 384; i++) {
            dstXIntrinsic_tmp = i + 384 * j;
            dstXIntrinsic[dstXIntrinsic_tmp] = (((double)j + 1.0) - 1.0) + 1.0;
            dstYIntrinsic[dstXIntrinsic_tmp] = (((double)i + 1.0) - 1.0) + 1.0;
        }
    }...
}

```

The loop executed sequentially.

In R2020a, the code generator produces this snippet of code:

```

void medfilt2(unsigned char b[65536])
{...

#pragma omp parallel for \
    num_threads(omp_get_max_threads()) \
    private(srcWorld_val,srcXWorld_val,rowIdx,srcXIntrinsic_tmp)

    for (colIdx = 0; colIdx < 384; colIdx++) {

```

```

for (rowIdx = 0; rowIdx < 384; rowIdx++) {
    srcXWorld_val = (1.3333333333333333 * (((double)colIdx + 1.0) - 0.5) +
        1.25) + -0.6666666666666663 * (((double)rowIdx + 1.0) - 0.5) + 1.25))
        + -0.3333333333333337;
    srcYWorld_val = (-0.6666666666666663 * (((double)colIdx + 1.0) - 0.5) +
        1.25) + 1.3333333333333333 * (((double)rowIdx + 1.0) - 0.5) + 1.25)) +
        -0.3333333333333331;
    srcXIntrinsic_tmp = rowIdx + 384 * colIdx;
    srcXIntrinsic[srcXIntrinsic_tmp] = (srcXWorld_val - 0.5) + 0.5;
    srcYIntrinsic[srcXIntrinsic_tmp] = (srcYWorld_val - 0.5) + 0.5;
}
}
.....
}

```

The generated code has the pragma for OpenMP (Open Multiprocessing) before the body of the loop. OpenMP enables shared-memory, multicore platforms to execute loops in parallel. This parallel execution improves the execution speed of the generated code. For more information, see [Speed Up for-Loop Implementation in Code Generated by Using parfor](#).

Capture main code execution profiling metrics on target hardware

For code execution profiling, to reduce the communication channel bandwidth usage during a software-in-the-loop simulation, processor-in-the-loop simulation, or XCP external mode simulation, you can capture and store only these profiling metrics on the target hardware:

- Maximum execution time of code section
- Average execution time of code section
- Number of calls to code section

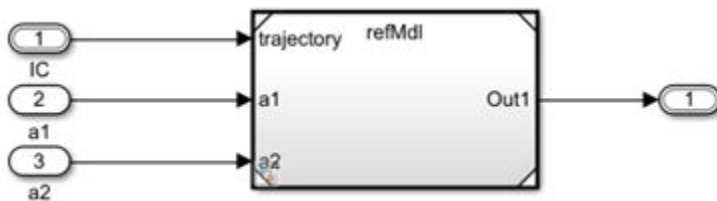
At the end of the simulation, Simulink uploads the metrics from the target hardware to your development computer.

For more information, see [Capture Main Profiling Metrics on Target Hardware](#).

Efficient code for model-reference builds in presence of function prototype control

In R2019b, the generated code for model reference builds contained redundant variable copies or dead code in the presence of the function prototype control. In R2020a, the generated code for a model reference build is optimized to remove dead code and data copies. Eliminating dead code and data copies conserves ROM consumption and improves execution speed.

Consider the model `MdlReferences` with a referenced model `refMdl`.



The step function interface of the referenced model is:

```
void MdlReferences_step([* self], arg_a1, arg_a2, * arg_IC)
```

In R2019b, the code generator for the referenced model produced this code:

```
/* Model step function */
void MdlReferences_step(RT_MODEL_testRef3 * const testRef3_M, real_T arg_a1, real_T
    arg_a2, BusObject *arg_IC)
{
    BusObject arg_IC_0;
    arg_IC_0 = *arg_IC;

    /* ModelReference: '<Root>/Model' */
    refMdl_step((&(testRef3_M->Model)), arg_a1, arg_a2, arg_IC);
}

```

The code contained a variable `arg_IC_0`, which was not used.

In R2020a, the code generator produces this code:

```
/* Model step function */
void MdlReferences_step(real_T arg_a1, real_T arg_a2, BusObject *arg_IC)
{
    /* ModelReference: '<Root>/Model' incorporates:
     * Inport: '<Root>/a1'
     * Inport: '<Root>/a2'
     */
    refMdl_step(&Model, arg_a1, arg_a2, arg_IC);
}

```

The code does not contain the variable `arg_IC_0` or unnecessary data copies, which improves the efficiency of generated code. For more information, see [Override Default C Step Function Interface](#).

Symbolic dimension support for Reshape blocks

In R2020a, you can generate code by using symbolic dimensions as inputs for Reshape blocks. Select the model configuration parameter **Allow symbolic dimension specification**. For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#). You can now use symbolic dimensions to set constraints for signal dimensions and as block parameters for the Reshape block.

Compatibility Considerations

When you specify the input port dimensions of a Reshape block by using symbolic dimensions, code generation behavior changes.

Before R2020a	R2020a
<p>Code generation results in an error. To resolve the error:</p> <ul style="list-style-type: none"> In the Model Configuration Parameters dialog box, clear the Allow symbolic dimension specification check box. 	Code generation is successful.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2019b

Version: 7.3

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Customize C/C++ code file names generated from MATLAB code

The code generator produces file names for your C/C++ code that correspond to your MATLAB code functionality, file names, and the code generation settings. In R2019b, you can generate file names customized with additional characters and tokens. Customize the file names to avoid name clashes when integrating multiple code projects together. See [Customize C/C++ File Names Generated from MATLAB Code](#).

Custom type definitions from external header files

In R2019b, you can import your custom type definitions from external header files. You can then use your own type definitions in the generated C/C++ code.

For more information, see [Import Custom Data Type Definitions from External Header Files](#).

Disable generation of initialize function

By default, the code generator produces an initialize function that initializes the data used by the entry-point functions. However, in R2019b, you can disable the generation of the initialize function while generating standalone code. If the body of your initialize function is empty, you can make this choice to avoid generation of redundant code.

The default behavior of the code generator is to produce the initialize function, even if it is empty. To disable the generation of the initialize function, do one of the following:

- In a `coder.EmbeddedCodeConfig` object, set `IncludeInitializeFcn` to `false`.
- In the MATLAB Coder app, on the **All Settings** tab, set **Initialize function required** to **No**.

Function profiling for SIL and PIL execution

Previously, software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution supported profiling for generated function code only at the entry-point level. In R2019b, SIL and PIL execution also supports profiling for functions that are called inside entry-point functions.

For more information, see:

- [Generate Execution Time Profile](#)
- [View Execution Times](#)
- [Analyze Execution Time Data](#)

Model Architecture and Design

Symbolic dimension support for Stateflow Data

In R2019b, when you select the model configuration parameter **Allow symbolic dimension specification**, Stateflow charts that use C as the action language can propagate the symbolic dimensions of Stateflow data through the model. The symbolic dimensions go into the generated code. For more information, see [Propagate Symbolic Dimensions of Stateflow Data \(Stateflow\)](#) and [Implement Dimension Variants for Array Sizes in Generated Code](#).

Compatibility Considerations

When you specify the size of a Stateflow data object by using a Simulink parameter with a storage class that is not supported for symbolic dimensions, code generation behavior changes.

Before R2019b	R2019b
In the generated code, the symbolic dimensions were replaced by their constant values. No error or warning occurred.	Code generation results in an error. To resolve the error: <ul style="list-style-type: none"> • Change the storage class for the Simulink parameter. • In the Model Configuration Parameters dialog box, clear the Allow symbolic dimension specification check box.

Generate C++ Code for Software Compositions with Message-Based Communication

R2019b introduces C++ and C code generation for message-based communication between Simulink model components using Messages & Events library Send, Receive, and Queue blocks. This release also introduces C++ code generation for message-based communication between Simulink top models and external message protocol services (middleware or operating systems).

For more information about code generation for message-based communication, see [Generate C or C++ Code for Message-Based Communication in Simulink](#) and [Generate C++ Code from Top Models for Message-Based Communication By Using External Message Protocols](#). For more information about the blocks, see [Send, Receive, and Queue block descriptions](#).

Cut, copy, and paste code definitions in Embedded Coder Dictionary

In R2019b, you can cut, copy, and paste code definitions into Embedded Coder Dictionaries. In the Embedded Coder Dictionary dialog box, select code definitions, and click the **Cut** button or the **Copy** button on the quick access toolbar. To add copies of the code definitions, click the **Paste** button in the same dictionary or another dictionary. For more information, see [Embedded Coder Dictionary](#).

Configure Embedded Coder Dictionary programmatically

Previously, you could create and alter code definitions only in the Embedded Coder Dictionary dialog box. In R2019b, you can use the Embedded Coder Dictionary API to configure code definitions

programmatically. To interact with an Embedded Coder Dictionary and its definitions, use these new classes and new functions.

Class	Description
<code>coder.Dictionary</code>	Configure an Embedded Coder Dictionary
<code>coder.dictionary.Section</code>	Access code definitions in one section of an Embedded Coder Dictionary
<code>coder.dictionary.Entry</code>	Interact with one code definition
Function	Description
<code>coder.dictionary.create</code>	Create an Embedded Coder Dictionary
<code>coder.dictionary.open</code>	Access an existing Embedded Coder Dictionary

A `coder.Dictionary` object contains a `coder.dictionary.Section` object for each type of code definition: storage classes, memory sections, and function customization templates. A `coder.dictionary.Section` object contains `coder.dictionary.Entry` objects, which represent the code definitions stored in that section of the Embedded Coder Dictionary.

You can apply the definitions in the dictionary to model elements by configuring settings in the **Code Mappings Editor**. For an example of the feature, see [Configure Code Definitions Programmatically](#).

Data, Function, and File Definition

Generated code calibration and monitoring through XCP and third-party tools

You can generate code that supports parameter tuning and signal monitoring through an ASAM *MCD-1 XCP* communication channel and third-party calibration tools, for example, ETAS® INCA and Vector CANape®. The XCP communication channel supports the *XCP on Ethernet* (TCP/IP) and *XCP on SxI* (SCI) transport layers.

The code generator creates:

- An XCP external mode target application to which you can connect the third-party XCP calibration tools.
- In an ASAP2 file, an IF_DATA XCP block that describes the Simulink Coder XCP slave configuration.

For more information, see *Calibrate Generated Code and Monitor Signals Through XCP and Third-Party Tools*.

Argument specifications not required for Function Caller blocks that invoke scoped Simulink functions

Prior to R2019b, to configure the function prototype for a scoped Simulink Function block that is invoked by a Function Caller block in the parent model, you had to specify input and output arguments for the Function Caller block. Starting in R2019b, this is no longer required.

For more information, see *Customize Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks*.

Implicit validation occurs when configuring C function prototypes

When you configure prototypes for generated C entry-point functions, Simulink implicitly validates the configuration for you. Because validation occurs implicitly, you do not need to call the `runValidation` method. The `runValidation` method is no longer supported.

For more information about configuring C function prototypes, see *Customize Generated C Function Interfaces*.

Map storage classes defined in Embedded Coder Dictionary to nonreusable subsystems with separate data

In R2019b, if your model contains a nonreusable subsystem configured with the subsystem parameter **Function with separate data** selected, you can use the Code Mappings editor to associate the separate internal data for the subsystem with storage classes defined in an Embedded Coder Dictionary. Generating a function with separate data for a nonreusable function can improve the traceability and testability of code because the subsystem data is declared separately from the parent model data structures.

Apply the new storage classes that you create to a model element category in **Code Mappings > Data Defaults**. For more information, see *Configure Default C Code Generation for Categories of Model Data and Functions*.

Code Mappings Editor Changes

These changes were made to the Code Mappings editor:

- **Entry-Point Functions** tab was renamed to **Functions**.
- The tab order, from left to right, was changed to **Data Defaults**, **Function Defaults**, and **Functions**.
- On the **Data Defaults** tab, parameter categories were renamed.
 - **Local parameters** was renamed to **Model parameters**.
 - **Parameter arguments** was renamed to **Model parameter arguments**.
 - **Global parameters** was renamed to **External parameter objects**.

For more information, see **Code Mappings Editor**.

Function `rtw.asap2SetAddress` extracts DWARF debug symbols from binaries compiled using MinGW compiler

If you are generating Executable and Linkable Format (ELF) or Program Database (PDB) files for an embedded target, you can use the `rtw.asap2SetAddress` function to automate replacement of ECU Address placeholder memory address values with actual addresses in a generated ASAP2 file. You specify a call to the function with the name of the generated ASAP2 file and the name of the generated executable ELF, PDB, or DWARF debug information file for the model. Prior to R2019b, if you had an executable program file (.exe) produced with the MinGW compiler, you had to extract DWARF content from that file and pass the file containing the extracted DWARF content to `rtw.asap2SetAddress`. As of R2019b, you can pass the executable program file as produced with the MinGW compiler, without the extraction, to the function.

For more information, see *Automatic ECU Address Replacement for ASAP2 Files (Embedded Coder)*.

Code Generation

Optimized C++ generated code for reusable functions

In R2019a and earlier releases, the code generator packaged reusable subsystem code as functions outside the class definition in the file `model.h` file. In R2019b, during C++ code generation, the code generator generates reusable functions as private methods inside a class, unless reused across models. For example, here is the code in `mBasic.h`

```
class mBasicModelClass {
public:
    ExtU_mBasic_T mBasic_U;
    ExtY_mBasic_T mBasic_Y;
    void initialize();
    void step();
    void terminate();
    mBasicModelClass();
    ~mBasicModelClass();
    RT_MODEL_mBasic_T * getRTM();
private:
    DW_mBasic_T mBasic_DW;
    RT_MODEL_mBasic_T mBasic_M;
    void mBasic_Subsystem(real_T rtu_In, real_T *rty_Out, DW_Subsystem_mBasic_T
                        *localDW);
};
```

The subsystem `mBasic_Subsystem` is a member of the model class `MyClass` and has access to the internals like static parameters and private functions.

The exceptions are when you perform one of these:

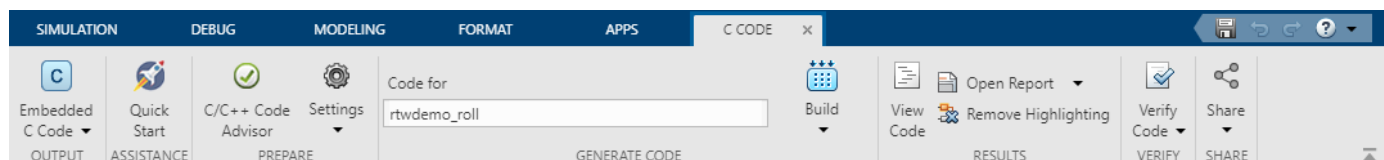
- Code generation of reusable functions linked to a library.
- Library-based code generation of reusable library subsystem.

For more information, see [Generate C++ Class Interface to Model or Subsystem Code](#).

Embedded Coder contextual tabs on the Simulink Toolstrip

To assist you in your code generation workflow, use the Embedded Coder contextual tabs.

To access the **C Code** or the **C++ Code** tab, open the new Embedded Coder app from the **Apps** gallery tab on the Simulink Toolstrip. To support common code generation workflow tasks, the tab provides Embedded Coder functionality corresponding to each task. The Embedded Coder app places the model in the Simulink Editor Code perspective. For information about the Embedded Coder app, see [Embedded Coder](#).



The Embedded Coder app supports models configured with ERT-based system target files. If you have not configured your model or model hierarchy with an ERT-based system target file, Embedded Coder prompts you to either open an app that supports your model's system target file or change your model's system target file to `ert.tlc`.

For more information, see “Simulink Toolstrip: Access and discover Simulink capabilities when you need them”.

Simulink strings through standard C++ string library

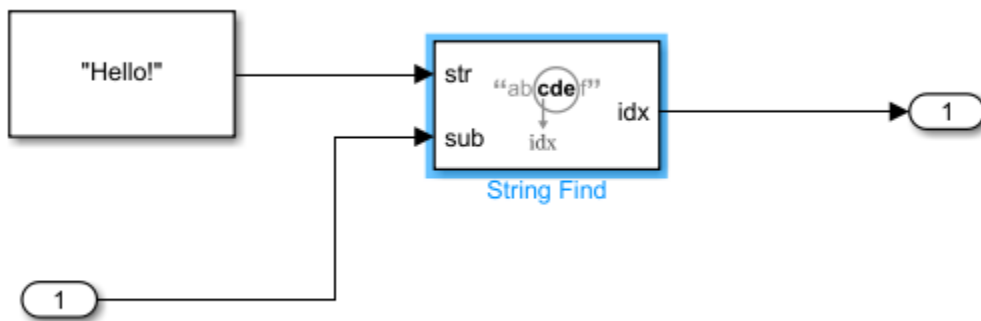
In R2019b, you can generate C++ code from model blocks by using the standard C++ string library. In R2019a, the generated code relied on the character-array-based C string library.

The C++ string library provides consistent C++ code and improves functionality, such as string length retrieval. The C++ string library contains functionality, such as concatenation, string copy, string swapping, string comparison and substring computations. You implement these functionalities in the model by using the corresponding blocks in the string library.

To use the C++ string library in the generated code, set these configuration parameters:

- **Language** parameter to C++.
- **Code interface packaging** parameter to the default C++ class value.
- **Standard math library** parameter to the default C++03 (ISO) value.

Consider the model `mStrfindSubStr` with a string "Hello!" as input and a String Find block with a value of "abcde". The block models a substring find operation.



In R2019a, the code generator produced this code in the header file:

```
typedef struct {
    char_T In1[256];
} ExtU_mStrfindSubStr_T;
```

The code instantiated a character array `In1`.

In R2019a, the code generator produced this code in the C++ source file:

```
// Model step function
void untitled1ModelClass::step()
{
    const char_T *tmp;
    uint16_T tmp_0;
    uint16_T tmp_1;

    // StringFind: '<Root>/String Find' incorporates:
    //   Inport: '<Root>/In1'
    //   StringConstant: '<Root>/String Constant'
```

```

tmp = strstr(&mStrfindSubStr_P.StringConstant_String[0],
            &mStrfindSubStr_U.In1[0]);
if (tmp == NULL) {
    // Output: '<Root>/Out1'
    mStrfindSubStr_Y.Out1 = -1;
} else {
    // Output: '<Root>/Out1'
    mStrfindSubStr_Y.Out1 = (int32_T)(tmp -
    &mStrfindSubStr_P.StringConstant_String[0]) + 1;
}
}

```

The code used multiple array copies to determine if "abcde" was a substring of "Hello!".

In R2019b, the code generator produces this code in the header file:

```

typedef struct {
    std::string In1;
} ExtU_mStrfindSubStr_T;

```

The code instantiates a `std:string` object `In1`.

In R2019b, the code generator produces this code in the C++ source file:

```

// Model step function
void untitled1ModelClass::step()
{
    uint32_T tmpOut;

    // StringFind: '<Root>/String Find' incorporates:
    //   Inport: '<Root>/In1'
    //   StringConstant: '<Root>/String Constant'

    tmpOut = mStrfindSubStr_P.StringConstant_String.find(mStrfindSubStr_U.In1);
    if (tmpOut == (uint32_T)std::string::npos) {
        // Output: '<Root>/Out1'
        mStrfindSubStr_Y.Out1 = -1;
    } else {
        // Output: '<Root>/Out1'
        mStrfindSubStr_Y.Out1 = static_cast<int32_T>(tmpOut) + 1;
    }
}

```

The code uses a clearer method of looking for the first character match, by using the `find` function, which is present in the C++ string library. This code executes faster and is easier to read.

The current implementation has these limitations:

- An array of bus that contains `std:string` is not supported in software-in-loop and processor-in-loop simulations.
- Code generation using `std:string` library does not work and the generated code uses C `char_T` arrays if:
 - **Use dynamic memory allocation for model initialization** parameter is selected.
 - **Use dynamic memory allocation for model block instantiation** parameter is selected.
 - **Remove root level I/O zero initialization** parameter is cleared on the configuration parameters dialog box.
 - **Remove internal data zero initialization** parameter is cleared on the configuration parameters dialog box.

For more information, see “Generate Code for String Blocks by Using the Standard C++ String Library”.

C++ `static_cast` in generated code

In R2019b, when generating C++ code, in the `model.cpp` file, the code generator replaces C-style type casting with `static_cast<>()` syntax. The C-style casts are difficult to locate in the generated code and cannot be checked during compile time. The `static_cast<>()` syntax makes the generated code more readable. The compiler can check the `static_cast<>()` at compile time.

For example, in R2019a, the code generator produced this code:

```
void CPPCodeModelClass::initialize()
{
    // Registration code

    // initialize error status
    rtmSetErrorStatus((&mMemsetCast_M), (NULL));

    // states (dwork)
    (void) memset((void *)&mMemsetCast_DW, 0,
                 sizeof(DW_mMemsetCast_T));

    // external inputs
    (void)memset(&mMemsetCast_U, 0, sizeof(ExtU_mMemsetCast_T));

    // external outputs
    (void) memset((void *)&mMemsetCast_Y, 0,
                 sizeof(ExtY_mMemsetCast_T));
}
```

In R2019b, the code generator produced this code:

```
void CPPCodeModelClass::initialize()
{
    // Registration code

    // states (dwork)
    (void) std::memset(static_cast<void *>(&mMemsetCast_DW), 0,
                      sizeof(DW_mMemsetCast_T));

    // external inputs
    (void)std::memset(&mMemsetCast_U, 0, sizeof(ExtU_mMemsetCast_T));

    // external outputs
    (void) std::memset(static_cast<void *>(&mMemsetCast_Y), 0,
                      sizeof(ExtY_mMemsetCast_T));
}
```

When you enable the **MAT-file logging** model configuration parameter, the code generated in `model.cpp` might still use C-style casts.

Inline traceability for variable and type definitions

R2019b provides line-level traceability coverage for the variable and type definitions in header files. Inline traceability is available with or without comments.

From the code generation report, click a hyperlinked line of code to navigate to corresponding blocks in the model. From the Code view in Code perspective, place your cursor over or click a hyperlinked line of code to navigate to corresponding blocks in the model.

For more information, see [Verify Generated Code by Using Code Tracing](#).

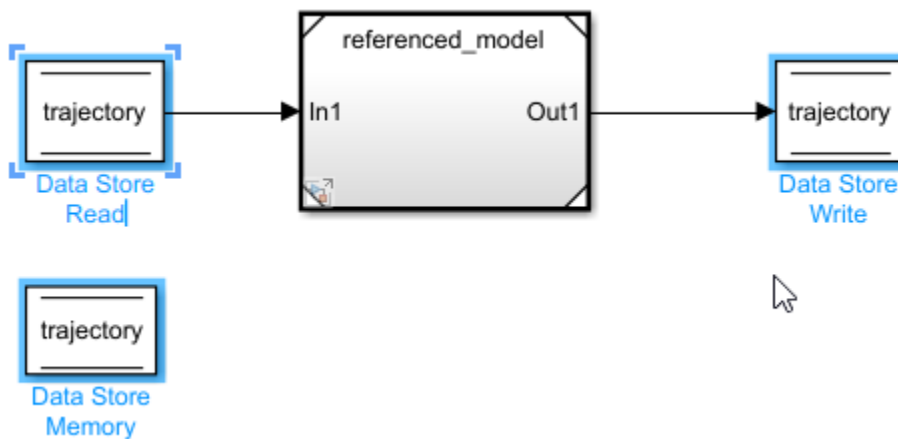
Deployment

Performance

Improved Data Store Memory block reuse to reduce data copies

In R2019b, the generated code contains fewer data copies for models that use Data Store Memory blocks to store large bus structures in a model reference hierarchy. Eliminating these data copies conserves RAM and ROM consumption and improves execution speed.

Consider the model `mDataStore` with a Data Store Read block that reads a bus structure from a Data Store Memory block called `trajectory`. The signal is input to a reference model `refMdl_fpc_1`. The bus is output from the referenced model and written by a Data Store Write block to the Data Store Memory `trajectory`.



In R2019a, the code generator produced this code:

```
/* Model step function */
void mg1963253_step(void)
{
    /* local block i/o variables */
    int32_T rtb_Model2;

    /* DataStoreRead: '<Root>/Data Store Read' */
    rtb_Model2 = mg1963253_DW.trajectory;

    /* ModelReference: '<Root>/Model2' */
    refMdl_fpc_1_step(&Model2, &rtb_Model2);

    /* DataStoreWrite: '<Root>/Data Store Write' */
    mg1963253_DW.trajectory = rtb_Model2;
}

```

The code contained an unnecessary data copy to the variable `rtb_Model2`.

In R2019b, the code generator produces this code:

```
/* Model step function */
void mg1963253_step(void)
{
    /* ModelReference: '<Root>/Model2' incorporates:
     * DataStoreWrite: '<Root>/Data Store Write'
     */
    refMdl_fpc_1_step(&Model2, &mg1963253_DW.trajectory);
}

```

The code does not contain the local variable `rtb_Model2` and data copy, increasing the efficiency of generated code.

SIMD vectorization for loops

In R2019b, for Intel SSE or AVX processors and the ARM® Neon® processors, SIMD intrinsics can vectorize loops and arrays. Vectorized loops process a vector of data as a single instruction, thereby improving execution speed. This vectorization is currently available for MATLAB Coder. The vectorization provides improved speed and efficiency in generated code.

To generate the code, select an Intel SSE or AVX or ARM Neon code replacement library.

Consider the MATLAB function:

```
function [a] = simple(w, x)
    a = w .* 0.0;
    for i = 1:numel(w)
        a(i) = (w(i) + x(i)) .* (w(i) + x(i));
    end
end
```

In R2019a, the code generator produced this C code:

```
for (i = 0; i < 16641; i++) {
    a_tmp = w[i] + x[i];
    a[i] = a_tmp * a_tmp;
}
```

The loop incremented by one for `single` and `double` data types.

In R2019b, the code generator produces this vectorized code for Intel SSE code replacement library:

```
for (i = 0; i <= 16636; i += 4) {
    r = _mm_add_ps(_mm_loadu_ps(&w[i]), _mm_loadu_ps(&x[i]));
    _mm_storeu_ps(&a[i], _mm_mul_ps(r, r));
}
```

The loop increments by 4 because the input data type is `single`. Incrementing by four instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `double`, the loop increments by two. This optimization increases the execution speed of the generated code. For more information, see [What Is Code Replacement? \(MATLAB Coder\)](#).

Optimized code execution speed for Ceiling, Floor, Minimum and Maximum SIMD intrinsic functions

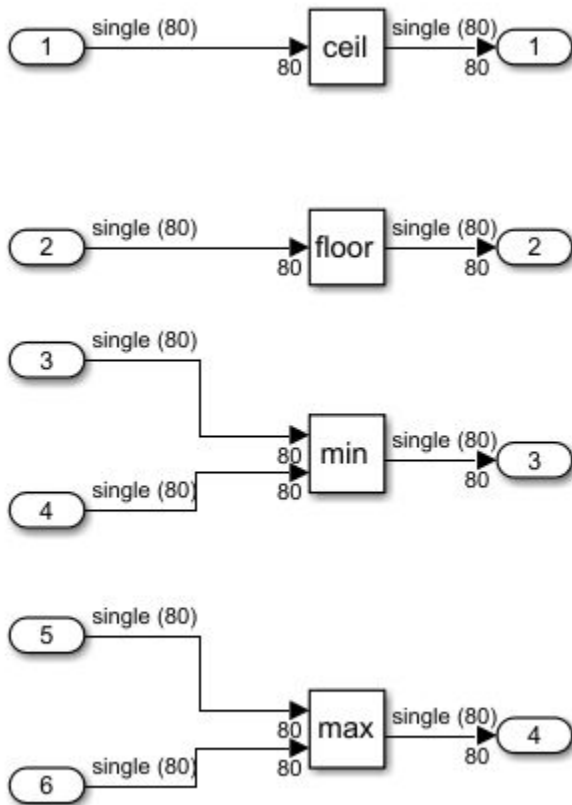
In R2019b, for Intel SSE or AVX processors, you can optimize the Rounding Function and MinMax blocks for models, in parallel in the generated code by using SIMD intrinsics. This optimization results in improved execution speed of the generated code. The optimization of MinMax blocks is also available for the ARM Neon code replacement library. To generate this code, in the Configuration Parameters dialog box, set the **Code replacement library** parameter.

For Intel SSE or AVX processors, you can optimize and compute the `ceil`, `floor`, `max`, `min`, and `sqrt` functions in parallel in the generated code by using SIMD intrinsics. To generate this code, in the MATLAB Coder app, on the **Custom Code** tab, set the **Code replacement library** parameter.

Use the rounding operation for element-wise operations involving `single` and `double` data types. The rounding-off operations are Ceiling and Floor. These operations calculate the rounded-up and the rounded-down whole number values of the input.

The maxima and minima operations require two inputs involving `single` or `double` data types. Calculating the maxima and minima operations yield one element.

Consider the model `mFunctions` that has inputs in `single` data type.



In R2019a, the code generator produced this code:

```
for (i = 0; i < 80; i++) {
/* Output: '<Root>/Output' incorporates:
 * Inport: '<Root>/In1'
 * Rounding: '<Root>/Ceil'
 */
mFunctions_Y.Output[i] = ceilf(mFunctions_U.In1[i]);

/* Output: '<Root>/Output1' incorporates:
 * Inport: '<Root>/In3'
 * Rounding: '<Root>/Floor'
 */
mFunctions_Y.Output1[i] = floorf(mFunctions_U.In3[i]);

/* Output: '<Root>/Output2' incorporates:
 * Inport: '<Root>/In5'
 * Inport: '<Root>/In6'
 * MinMax: '<Root>/Min'
 */
mFunctions_Y.Output2[i] = fminf(mFunctions_U.In5[i], mFunctions_U.In6[i]);

/* Output: '<Root>/Output3' incorporates:
 * Inport: '<Root>/In2'
 * Inport: '<Root>/In4'

```

```

    * MinMax: '<Root>/Max'
    */
    mFunctions_Y.Outputport3[i] = fmaxf(mFunctions_U.In2[i], mFunctions_U.In4[i]);
}

```

The loop incremented by one for single and double data types.

In R2019b, the code generator produces this code:

```

for (idx = 0; idx <= 76; idx += 4) {
/* Inport: '<Root>/In1' */
tmp = _mm_loadu_ps(&mFunctions_U.In1[idx]);

/* Output: '<Root>/Output' */
tmp_0 = _mm_ceil_ps(tmp);
_mm_storeu_ps(&mFunctions_Y.Output[idx], tmp_0);

/* Inport: '<Root>/In3' */
tmp = _mm_loadu_ps(&mFunctions_U.In3[idx]);

/* Output: '<Root>/Output1' */
tmp_0 = _mm_floor_ps(tmp);
_mm_storeu_ps(&mFunctions_Y.Output1[idx], tmp_0);

/* Inport: '<Root>/In5' */
tmp = _mm_loadu_ps(&mFunctions_U.In5[idx]);

/* Inport: '<Root>/In6' */
tmp_0 = _mm_loadu_ps(&mFunctions_U.In6[idx]);

/* Output: '<Root>/Output2' */
tmp_1 = _mm_min_ps(tmp, tmp_0);
_mm_storeu_ps(&mFunctions_Y.Output2[idx], tmp_1);

/* Inport: '<Root>/In2' */
tmp = _mm_loadu_ps(&mFunctions_U.In2[idx]);

/* Inport: '<Root>/In4' */
tmp_0 = _mm_loadu_ps(&mFunctions_U.In4[idx]);

/* Output: '<Root>/Output3' */
tmp_1 = _mm_max_ps(tmp, tmp_0);
_mm_storeu_ps(&mFunctions_Y.Output3[idx], tmp_1);
}

```

The loop increments by four because the input data type is single. Incrementing by four instead of one occurs because the SIMD functions in the loop body process data in parallel. This optimization increases the execution speed of the generated code. If the input data type is double, the loop increments by two. For more information, see Code replacement library (Simulink Coder).

SIMD vectorization for loops without compile-time bounds

In R2019b, for Intel SSE or AVX processors and the ARM Neon processor, SIMD intrinsics can vectorize loops and arrays whose bounds are not set at compile time. Vectorization processes a vector of data as a single instruction, improving execution speed. Vectorization is currently available for MATLAB Coder. This optimization improves speed and efficiency in generated code. Using SIMD on loops also improves the ease of coding because you do not need to specify the bounds for the loops while programming.

To generate the code, select either Intel SSE or AVX or the ARM Neon code replacement library.

Consider the MATLAB function:

```
function C = dynamic(A, B)
    assert(all(size(A) <= [100 100]));
    assert(all(size(B) <= [100 100]));
    assert(isa(A, 'single'));
    assert(isa(B, 'single'));

    C = zeros(size(A), 'like', A);
    for i = 1:numel(A)
        C(i) = (A(i) .* B(i)) + (A(i) .* B(i));
    end
end
```

In R2019a, the code generator produced this C code:

```
void dynamic(const float A_data[], const int A_size[2], const float B_data[],
             const int B_size[2], float C_data[], int C_size[2])
{
    signed char unnamed_idx_0;
    signed char unnamed_idx_1;
    int loop_ub;
    int i;
    float C_data_tmp;
    (void)B_size;
    unnamed_idx_0 = (signed char)A_size[0];
    unnamed_idx_1 = (signed char)A_size[1];
    C_size[0] = unnamed_idx_0;
    C_size[1] = unnamed_idx_1;
    loop_ub = unnamed_idx_0 * unnamed_idx_1;
    if (0 <= loop_ub - 1) {
        memset(&C_data[0], 0, (unsigned int)(loop_ub * (int)sizeof(float)));
    }

    loop_ub = A_size[0] * A_size[1];
    for (i = 0; i < loop_ub; i++) {
        C_data_tmp = A_data[i] * B_data[i];
        C_data[i] = C_data_tmp + C_data_tmp;
    }
}
```

The code sequentially computed the product of sums operation of the array in the loop one iteration at a time. The loop bound was unspecified at compile time and was represented by the variable `loop_ub`.

In R2019b, the code generator produces this C code by using Intel SSE code replacement library:

```
void dynamic(const float A_data[], const int A_size[2], const float B_data[],
             const int B_size[2], float C_data[], int C_size[2])
{
    signed char unnamed_idx_0;
    signed char unnamed_idx_1;
    int loop_ub;
    int scalarLB;
    int vectorUB;
    int i;
    __m128 r;
    float C_data_tmp;
    (void)B_size;
    unnamed_idx_0 = (signed char)A_size[0];
    unnamed_idx_1 = (signed char)A_size[1];
    C_size[0] = unnamed_idx_0;
    C_size[1] = unnamed_idx_1;
    loop_ub = unnamed_idx_0 * unnamed_idx_1;
    if (0 <= loop_ub - 1) {
        memset(&C_data[0], 0, loop_ub * sizeof(float));
    }
}
```

```

loop_ub = A_size[0] * A_size[1];
scalarLB = loop_ub & -4;
vectorUB = scalarLB - 4;
for (i = 0; i <= vectorUB; i += 4) {
    r = _mm_mul_ps(_mm_loadu_ps(&A_data[i]), _mm_loadu_ps(&B_data[i]));
    _mm_storeu_ps(&C_data[i], _mm_add_ps(r, r));
}

for (i = scalarLB; i < loop_ub; i++) {
    C_data_tmp = A_data[i] * B_data[i];
    C_data[i] = C_data_tmp + C_data_tmp;
}
}

```

The loop increments by four because the input data type is `single`. Incrementing by four instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `double`, the loop increments by two. This optimization increases the execution speed of the generated code. For more information, see [What Is Code Replacement? \(MATLAB Coder\)](#).

SIMD for row-major operations

In R2019a, the code generator produced C/C++ SIMD code with column-major array layout. In R2019b, for MATLAB Coder and for Intel SSE or AVX processors and ARM Neon processors, the code generator can produce code with SIMD intrinsics for C/C++ code that uses row-major array layout. See [Row-Major and Column-Major Array Layouts \(MATLAB Coder\)](#).

Generating row-major layout and adding SIMD processing can improve performance for certain algorithms and ease integration with other code that uses row-major layout.

To generate the code, select an Intel SSE or AVX or ARM Neon code replacement library.

Consider the MATLAB function:

```

function C = rowmajor2(A, B)
    assert(all(size(A) == [100 100]));
    assert(all(size(B) == [100 100]));
    assert(isa(A, 'single'));
    assert(isa(B, 'single'));

    C = zeros(size(A), 'like', A);
    for i = 1:100
        for j = 1:100
            C(i,j) = (A(i,j) + B(i,j)) .* (A(i,j) + B(i,j));
        end
    end
end

```

In R2019a, the code generator produced this C code:

```

for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
        b_i = j + 100 * i;
        C_tmp = A[b_i] + B[b_i];
        C[b_i] = C_tmp * C_tmp;
    }
}

```

The vectorized code sequentially computed the product of sums operation of the array in the loop one iteration at a time for the row major iterator `j`.

In R2019b, the code generator produces this C code with Intel SSE code replacement library:

```

for (i = 0; i < 100; i++) {
    for (j = 0; j <= 96; j += 4) {

```



```
    simd_tmp = j + 100 * i;  
    r = _mm_add_ps(_mm_loadu_ps(&A[simd_tmp]), _mm_loadu_ps(&B[simd_tmp]));  
    _mm_storeu_ps(&C[simd_tmp], _mm_mul_ps(r, r));  
  }  
}
```

The vectorized code sequentially computes the product of sums operation of the array in the loop for the row major iterator `j`. The loop increments by four because the input data type is `single`. Incrementing by four instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `double`, the loop increments by two. This optimization increases the execution speed of the generated code. For more information, see [What Is Code Replacement?](#) (MATLAB Coder).

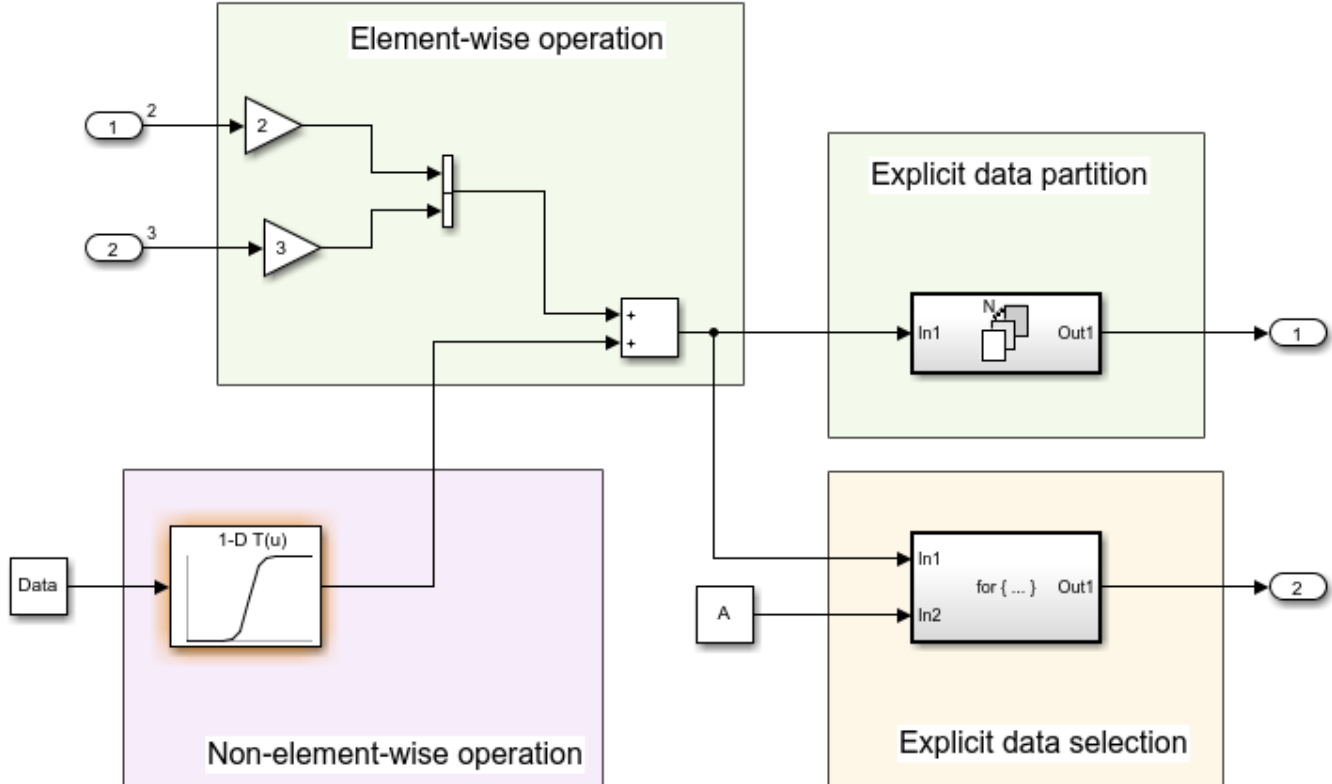
Specification of upper constraint limit for symbolic dimensions

In R2019a, you generated code that checked whether a symbolic dimension was bound by an lower limit. In R2019b, you can generate code that also checks whether it is bound by a upper limit. The code for this check is placed in a header file as a preprocessor directive in the form of a macro.

Symbolic dimensions placed in blocks and data objects help simulate dimension choices without regenerating code for every set as the model passes through simulation and code generation. You can choose to specify these dimensions in between certain positive upper and lower limit values.

Having both the lower and upper limit in the generated code means you can accurately validate the bounds of the parameter values linked to symbolic dimensions before deployment.

Consider the model `rtwdemo_dimension_variants` with multiple modeling patterns. In the model, is an Inport labeled `1` with port dimensions defined by symbol `A`.



In R2019a, the code generator produced this code:

```
#if A <= 1
# error "The preprocessor definition 'A' must be greater than '1'"
#endif
```

The `rtwdemo_dimension_variants.h` file contained data definitions and preprocessor conditionals that defined constraints established between the symbols during simulation. The value of `A` was checked for only a lower constraint, which is 1.

In R2019b, the code generator produces this code:

```
#if A <= 1
# error "The preprocessor definition 'A' must be greater than '1'"
#endif
#if A >= 11
# error "The preprocessor definition 'A' must be less than '11'"
#endif
```

The `rtwdemo_dimension_variants.h` file contains data definitions and preprocessor conditionals that define constraints established between the symbols during simulation. The value of `A` is checked for both lower and upper constraints of 1 and 11. This verification improves the accuracy of the signal dimension during compile time and deployment. For more information, see [Implement Dimension Variants for Array Sizes in Generated Code](#).

Parameter expression saturation

In R2019b, you can control whether the generated code contains saturation code that protects against out-of-range values for tunable parameter expressions. Turning off saturation improves the execution efficiency of the generated code.

To remove saturation from the generated code, in the Configuration Parameters dialog box, select the **Remove code from tunable parameter expressions that saturate out of range values** parameter. The generated code is free of any saturation check bounds and runs more efficiently. For more information, see [Remove Code from Tunable Parameter Expressions That Saturate Against Integer Overflow](#).

You can use the Model Advisor to check the model for configuration parameters that might generate inefficient saturation code. In the Model Advisor, select and run **By Product > Embedded Coder > Check configuration parameters for generation of inefficient saturation code**. For more information, see [Check configuration parameters for generation of inefficient saturation code](#).

Changes to zero initialization code model configuration parameter default settings

In R2019b, the default settings of the Remove root level I/O zero initialization (Simulink Coder) and Remove internal data zero initialization (Simulink Coder) parameters have changed. The **Remove root level I/O zero initialization** and **Remove internal data zero initialization** check boxes are selected by default. At the command line, `ZeroExternalMemoryAtStartup` and `ZeroInternalMemoryAtStartup` are set to 'off' for a model in which the Code interface packaging (Simulink Coder) model configuration parameter is set to `Nonreusable` function. Removing zero initialization code improves the execution efficiency of the generated code and conserves ROM usage.

During startup, standards-compliant C and C++ compilers initialize global data to zero, eliminating the need to include zero initialization code for this data in the generated code. Standards-compliant compilers do not necessarily initialize dynamically allocated data and local variables to zero. Before leaving the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** check boxes selected, confirm:

- If your compiler is not standards-compliant, that it initializes global data to zero.
- If you set the **Code interface packaging** to `Reusable` function or `C++ Class`, that data is either statically allocated or that dynamically allocated data is initialized to zero.

If you set the **Code interface packaging** parameter to `Reusable` function and select the Use dynamic memory allocation for model initialization (Simulink Coder) parameter, the **Remove root level I/O zero initialization** and **Remove internal data zero initialization** check boxes are cleared and `ZeroExternalMemoryAtStartup` and `ZeroInternalMemoryAtStartup` are set to 'on'.

For a model in which the **Code interface packaging** parameter is set to `C++ Class` and the Use dynamic memory allocation for model block instantiation (Simulink Coder) parameter is selected, the **Remove internal data zero initialization** check box is cleared and `ZeroInternalMemoryAtStartup` is set to 'on' and is read-only.

For more information, see [Remove Zero Initialization Code](#).

Compatibility Considerations

- When you load an existing model, the **Remove root level I/O zero initialization** and the **Remove internal data zero initialization** check boxes are cleared and `ZeroExternalMemoryAtStartup` and `ZeroInternalMemoryAtStartup` are set to 'on' for a model in which the **Code interface packaging** parameter is set to `Reusable function` and you select the **Use dynamic memory allocation for model initialization** parameter.
- When you load an existing model, the **Remove internal data zero initialization** checkbox is cleared and the `ZeroInternalMemoryAtStartup` is set to 'on' (command line) and is read-only for a model in which the **Code interface packaging** parameter is set to `C++ Class` and the **Use dynamic memory allocation for model block initialization** check box is selected.
- When you load a model configuration set from a MATLAB script that was created in R2014a or later, and this script sets the **Remove internal data zero initialization** and **Remove root level I/O zero initialization** parameters to values other than their default R2019b values, these parameters have the default R2019b values. These parameters have these values because scripts exported in R2014a or later set these parameters in this order:

```
cs.set_param('ZeroExternalMemoryAtStartup', 'on');
cs.set_param('ZeroInternalMemoryAtStartup', 'on');
...
...
cs.set_param('TargetLang', 'C');
cs.set_param('CodeInterfacePackaging', 'Nonreusable function');
```

In R2019b, to load a configuration set by using this MATLAB script, modify the script so that these parameters are set in the correct order for R2019b. The R2019b order sets the `TargetLang` and `CodeInterfacePackaging` parameters before the `ZeroExternalMemoryAtStartup` and `ZeroInternalMemoryAtStartup` parameters. You must set these parameters at the beginning of the script immediately after the line in which the `SystemTargetFile` parameter is set. The `SystemTargetFile` parameter is set by using the `switchTarget` function in scripts created in R2016a or later.

- When you load a model configuration set from a MATLAB script that was created in R2013b or earlier, change the order in which the parameters are set if the `TargetLang` parameter is set to `C++ (Encapsulated)`. In R2013b or earlier, the script does not contain the `CodeInterfacePackaging` line, so you must move the `TargetLang` line to the beginning of the script immediately after the line in which the `SystemTargetFile` parameter is set.

For more information on saving and loading model configuration sets, see [Manage a Configuration Set \(Simulink\)](#).

Enhanced code execution profiling report

Through the enhanced code execution profiling report, you can:

- Compare execution times of two software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.
- Display the CPU usage of generated tasks.
- Display the execution-time distribution for each profiled function.

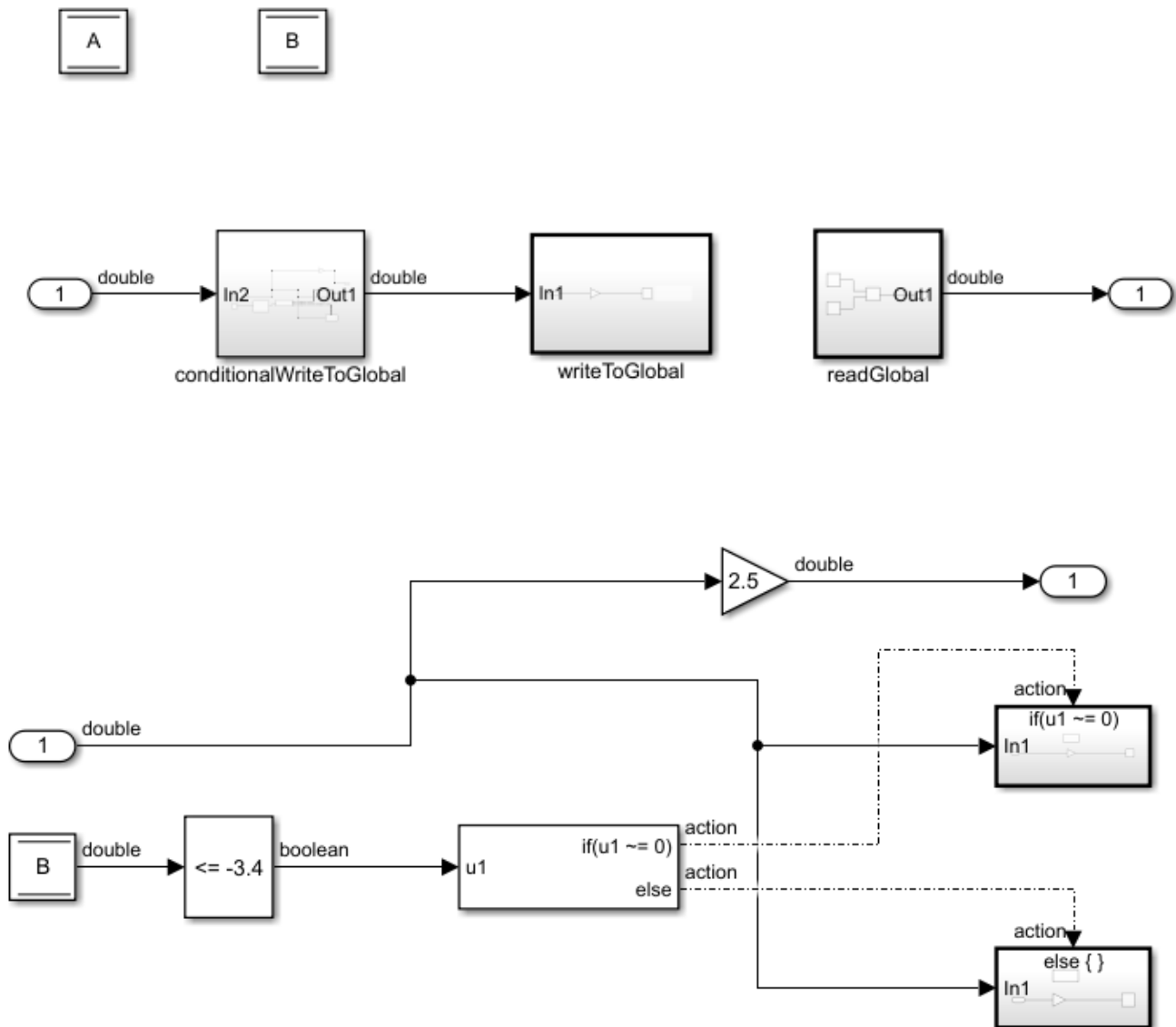
For more information, see:

- View and Compare Code Execution Times
- report

Elimination of unused writes to global variables

In R2019a, the generated code contained some unused write statements to global variables. In R2019b, for some modeling patterns, the code generator eliminates these write statements, which results in less ROM consumption and improved execution speed.

For example, the model `writeInsideFunction` contains a subsystem `conditionalWriteToGlobal` that conditionally writes to the global data store A. The subsystem `writeToGlobal` writes to A. The subsystem `readGlobal` reads from the global data store A.



In R2019a, the code generator produced this code in the `writeInsideFunction.c` file.

```
real_T A;
real_T B;
ExtU rtU;
ExtY rtY;
static void readGlobal(void);
static void writeToGlobal_1(void);
static void readGlobal(void)
{
    rtY.Out1 = A + -5.0;
}

static void writeToGlobal_1(void)
{
    A = 2.0 * rtY.Out1;
}

void writeInsideFunction_step(void)
{
    if (B <= -3.4) {
        A = -4.2 * rtU.In2;
    } else {
        A = -3.0 * rtU.In2;
    }

    rtY.Out1 = 2.5 * rtU.In2;
    writeToGlobal_1();
    readGlobal();
}
```

The generated code contained conditional write statements to the global variable A.

In R2019b, the code generator produces this code in the `writeInsideFunction.c` file.

```
real_T A;
real_T B;
ExtU rtU;
ExtY rtY;
static void readGlobal(void);
static void writeToGlobal_1(void);
static void readGlobal(void)
{
    rtY.Out1 = A + -5.0;
}

static void writeToGlobal_1(void)
{
    A = 2.0 * rtY.Out1;
}

void writeInsideFunction_step(void)
{
    rtY.Out1 = 2.5 * rtU.In2;
    writeToGlobal_1();
    readGlobal();
}
```

```
void writeInsideFunction_initialize(void)
{
    A = 2.0;
    B = 20.0;
}
```

The generated code does not contain the conditional write statements to the global variable A. These write statements are unnecessary because the function `writetoGlobal_1` writes to A immediately after the `if-else` statement.

Verification

SIL/PIL Manager

The SIL/PIL Manager is an app that simplifies verification of code that you generate from a model. You can:

- With one click, test numeric equivalence between the model and generated code by running back-to-back model simulations and software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.
- Configure SIL or PIL simulations to produce code coverage and execution-time profiling metrics.
- Enable your debugger for SIL simulations.
- Export automatically generated test cases for Simulink Test.

The app is in the **SIL/PIL** tab. From the **Apps** tab on the Simulink toolstrip, click **SIL/PIL Manager**. Or, in the **C Code** tab, click **Verify Code > SIL/PIL Manager**.

For more information, see:

- SIL/PIL Manager Verification Workflow
- **SIL/PIL Manager**

Code coverage information in Code view

In R2019b, when you edit your model in the Embedded Coder app, you can view code coverage information in the Code view. To view code coverage information, enable a code coverage tool in the Configuration Parameters dialog box **Verification** pane. Simulate the model in simulation-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode. To display coverage highlighting, on the **Coverage** tab, click **Coverage Highlighting**. The code coverage information is displayed in the Code view. You can still access the code coverage information in the code generation report and in the code coverage report. For more information, see Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode.

Data logging and signal viewer block support for export function models

R2019b adds logging and signal viewer block support for export function models. You can use logging and signal viewer blocks to test export-function models as top or referenced models and in the context of a test harness. For examples, see Test Export-Function Model Simulation Using Function-Call Generators (Simulink), Test Export-Function Model Simulation Using Stateflow Chart (Simulink), and Test Export-Function Model Simulation Using Schedule Editor (Simulink).

SIL/PIL for AUTOSAR Classic Software Components containing referenced models

You can run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of a top-level AUTOSAR Software Component (SWC) that contains referenced models. You can run top-model or Model block (**Code interface** set to Top model) SIL or PIL simulations. In the simulations, the software:

- Before compilation of referenced models, generates AUTOSAR Runtime Environment (RTE) header files.
- Provides the RTE include path for referenced model compilation.

You can also run Model block (**Code interface** set to `Model` reference) SIL or PIL simulations for a referenced model within the top-level AUTOSAR SWC. In this case, before you run a simulation, you must build the parent component to generate the RTE header files. If you do not build the parent component, the SIL or PIL simulation fails.

For more information, see AUTOSAR Runtime Environment.

Traceability for hidden blocks

The code generator sometimes inserts hidden blocks during the code generation process for various reasons, for example, to maintain data integrity. Comments for the hidden blocks are included in the generated code. In R2019b, these comments, and the code generated from the hidden block, trace back to the original block in the model that triggered the insertion of the hidden block. For more information, see Traceability to Hidden 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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2019a

Version: 7.2

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Custom Data Type Replacement: Specify custom data type names for MATLAB data types

Before R2019a, to specify the data type in the generated code, you used either the built-in C data types or MathWorks typedefs.

Starting in R2019a, you can specify custom names for the MATLAB data types in the generated code. This specification improves the readability of the generated code. For more information, see [Customize Data Type Replacement](#)

Model Architecture and Design

Library-based code generation for reusable subsystem function interfaces

In R2019a, for a top-level reusable library subsystem, you can create function interfaces in which you specify subsystem input and output block parameter settings and model configuration parameter settings. Function interfaces are independent models that you save with an accompanying library. Use function interfaces to lock down subsystem behavior so that the library and not the model owns the generated code.

You can create function interfaces from within a library or a model by right-clicking the reusable library subsystem and selecting **C/C++ Function Interfaces > Create Function Interface**. Once you create a function interface, a badge appears at the lower-right corner of the subsystem. To create function interfaces from within a model, enable the code perspective.

From within a library, you can choose from the following methods of creating function interfaces:

- Selecting an existing instance of a reusable library subsystem.
- Configure function interfaces directly in a library.
- Export a function interface. Configure it as a standalone model, and then import it to the library.

After you specify function interfaces for all subsystems in your library, before you generate code for your model, generate code for the library. When you generate code for a model that contains an instance of a reusable library subsystem that can use the pregenerated library code, the model links to the library code. If the model is unable to use the library code, you can specify whether Embedded Coder produces a warning, error, or neither during code generation by setting the new **Behavior when pregenerated library subsystem code is missing** model configuration parameter.

For more information, see Library-Based Code Generation for Reusable Library Subsystems

AUTOSAR Blockset product replaces Embedded Coder Support Package for AUTOSAR Standard

In R2019a, the AUTOSAR Blockset product replaces the Embedded Coder Support Package for AUTOSAR Standard. To generate AUTOSAR-compliant C/C++ code and XML component descriptions for AUTOSAR Classic and Adaptive Platforms, you install AUTOSAR Blockset.

Compatibility Considerations

If you are upgrading to AUTOSAR Blockset from Embedded Coder Support Package for AUTOSAR Standard, review information about compatibility and upgrade issues in the AUTOSAR Blockset release notes.

MISRA C:2012 and Secure Coding checks to improve compliance of generated code

Modifications to existing Model Advisor checks that you use to verify compliance with MISRA C:2012 and Secure Coding standards are outlined in this table.

Model Advisor Check	Description of Change
Check configuration parameters for MISRA C:2012 Check configuration parameters for secure coding standards	Checks now analyze the setting for these configuration parameters: <ul style="list-style-type: none">• Include comments (GenerateComments)• MATLAB user comments (MATLABFcnDesc)
Check for missing error ports for AUTOSAR receiver interfaces	When an error port is missing, the check flags receiver interface ports with these AUTOSAR data access mode types: <ul style="list-style-type: none">• ImplicitReceive• ExplicitReceive• EndToEndRead

Data, Function, and File Definition

Preserve array dimensions for root-level inports and outports in generated code

When configuration parameter **Array layout** is set to Row-major, you can preserve the dimensions of multidimensional array data for:

- Root-level inports and outports
- `Simulink.Signal` objects associated with root-level inport and outport

From the code mapping editor in the code perspective, you can configure default configurations to preserve dimensions of:

- Inports
- Outports
- Global parameters
- Local parameters

In the Embedded Coder Dictionary, to preserve dimensions when you design your own custom storage class, select the **Preserve array dimensions** option in the Property Inspector.

From the Model Data Editor, you can preserve dimensions of these elements by selecting the **Preserve array dimensions** property from the Property Inspector:

- Root-level inport and outport
- `Simulink.Parameter` object
- `Simulink.LookupTable` object

For more details, see [Preserve Dimensions of Multidimensional Arrays in Generated Code](#).

Custom storage class with different code generation settings for single-instance and multi-instance data

In R2019a, you can create custom storage classes that use different settings for single-instance and multi-instance data. For example, the data for a top model is single-instance. If the top model references another model multiple times, the data for the referenced model can be multi-instance. You can create a storage class that uses different storage settings for these two different cases.

In the Embedded Coder Dictionary, define a custom storage class and specify its settings in the **Property Inspector** pane. Select the new property **Use different property settings for single-instance and multi-instance data**. For single-instance data, you can specify the storage type and structure properties. You can separately specify the structure properties for multi-instance data.

When you apply the storage class to a data item, the Embedded Coder Dictionary implements either the single-instance settings or the multi-instance settings depending on the type of data and the context of the model within the model reference hierarchy. For more information, see [Flexible Storage Class for Different Model Hierarchy Contexts](#).

Code generation definitions in multiple packages from Embedded Coder Dictionary

Previously, in an Embedded Coder Dictionary, you could load only one package of code generation definitions at a time. In R2019a, you can load and refer to definitions in multiple packages simultaneously. To load and unload packages from an Embedded Coder Dictionary, click **Manage Packages** and select the package from the drop-down list. For more information, see Refer to Code Generation Definitions in a Package.

Storage classes with get and/or set data access functions in Embedded Coder Dictionary

Starting in R2019a, in the Embedded Coder Dictionary, you can define a storage class for a root-level inports, root-level outports, and local parameters so they can be accessed by customizable `get` and/or `set` functions. Such customization can be useful, for example, to abstract layers of software, gain access to data from an external file, or control access to critical sections of code.

You configure data access customization for a storage class by setting the **Data Access** property to **Function**. You have the option of configuring:

- **Data Scope:** Scope of the access functions, which is currently only imported.
- **Header File:** Naming rule for the header file declaring the access functions.
- **Access Mode:** Whether the access functions return data by value or by pointer.
- **Allowed Access:** Whether to allow read and write access, read-only access, or write-only access to the data.
- **Name of Getter:** Naming rules for the `get` functions.
- **Name of Setter:** Naming rules for the `set` functions.

For more information, see **Embedded Coder Dictionary** and Access Data Through Functions by Using Storage Classes in Embedded Coder Dictionary.

Code definitions from local and shared Embedded Coder Dictionaries

Previously, if a model contained local Embedded Coder Dictionary definitions, the Code Mappings editor did not use definitions from any linked data dictionaries. The editor used either the local dictionary or the shared dictionary, but not both.

In R2019a, you can use definitions from both the local Embedded Coder Dictionary and shared Embedded Coder Dictionaries contained in linked data dictionaries. When you link a shared data dictionary with code definitions to your model, those definitions are read-only in the local Embedded Coder Dictionary. The shared definitions also appear in the Code Mappings editor where you can apply them to model data. For more information, see Deploy Code Generation Definitions to Users.

Also in R2019a, opening the code perspective creates a local dictionary that contains the built-in storage classes in the model.

Compatibility Considerations

Previously, you used the function `coder.dictionary.remove` with `sourceName` set to the name of the model to remove the local Embedded Coder Dictionary from a model. In R2019a, when you use

the function `coder.dictionary.remove` on a model, the function does not remove the local dictionary. Instead, the function removes the local definitions from packages that you have loaded and local custom definitions. The model still contains the local dictionary with definitions from the `SimulinkBuiltIn` package.

Code packaging support for model arguments

Previously, you could not configure the packaging of model arguments in the generated code. In R2019a, you can configure the packaging for these parameters in the model where they are defined. In the Model Data Editor, set the storage class for a model argument to `Model default`. With this setting, the parameters acquire the default code generation settings that you specify for **Parameter arguments**. The **Model default** must be a structured storage class. Parameter values must be finite.

For more information, see [Configure Packaging of Parameter Arguments in Generated Code](#).

Model argument support for top models

In R2019a, you can generate code for a top model that promotes Model block or model workspace parameters as model arguments.

- For Model block parameters, specify that the parameter is an argument.
- For parameters in the model workspace, specify that the parameter is an argument and specify the storage class as `Model default`.

The parameter becomes part of a data structure in the generated code. If you generate code that uses the reusable code format, access the data structure through a pointer in the real-time model for the top model instance.

For more information about model arguments in the generated code, see [Configure Packaging of Parameter Arguments in Generated Code](#).

Compatibility Considerations

Previously, you accessed parameter data in the top model through a pointer in the real-time model. If you generated code that used the reusable code format, you could define different parameter values for each instance of a top model. You defined these different values by setting the pointer to different memory with different sets of parameter values. In R2019a, for multi-instance top models and top models that use a malloc memory model, parameter data in the top model is declared as a standalone global variable. For C++ code generation, the parameter data is a static member of the model class. To define different parameter values for each instance of a top model, you must configure model arguments in the same way that you configure model arguments for referenced models. Define the parameter in the model workspace then, in the Model Data Editor, Property Inspector, or Model Explorer, select the **Argument** check box.

C entry-point function prototype preview and customization in the Code Mapping Editor

In R2018b, you could preview and customize names of entry-point functions in the Code Mapping Editor column, **Function Name**. You could also customize the arguments of Simulink functions and step functions by selecting the vertical dots in the column to open a configuration dialog box.

In R2019a, you can preview and customize prototypes of entry-point functions in a new Code Mapping Editor column, **Function Preview**. To customize entry-point function names, and the arguments of Simulink functions and step functions, you can select the prototype hyperlink to open a configuration dialog box.

For more information, see [Override Default Naming for Individual C Entry-Point Functions](#) and [Override Default C Step Function Interface](#).

Code Generation

Code metrics information in code view

Code metrics information was previously available only in the code generation report. In R2019a, you can also view code metrics directly in the code view.

To view the code metrics for a variable or function, place your cursor over the variable or function in the code view. You can still access the metrics in the code generation report. For more information, see [Static Code Metrics](#).

Cross-release code import without opening previous release

Previously, you needed the `crossReleaseExport` function for the cross-release code integration workflow. The function opened a previous release and generated a cross-release artifact in a folder within the current working folder. In R2019a, the `crossReleaseExport` function is not required. To import generated code from a previous release, when you run either the `crossReleaseImport` or `sharedCodeUpdate` functions, specify the location of the build folder.

For more information, see:

- [Cross-Release Code Integration](#)
- [Integrate Generated Code by Using Cross-Release Workflow](#)

The `crossReleaseExport` function will be removed in a future release.

Compatibility Considerations

In R2019a, the `crossReleaseExport` function does not:

- Open a previous release.
- Generate cross-release artifacts.

You can continue to use scripts from previous releases that:

- Run the `crossReleaseExport` function.
- Specify cross-release artifact folders as arguments for the `crossReleaseImport` and `sharedCodeUpdate` functions.

This table lists examples of how the functions behave.

Scenario	Behavior
Script runs <code>crossReleaseExport</code> : <pre>artifactLocation = crossReleaseExport(pathToBuildFolder);</pre>	Warning produced. <code>crossReleaseExport</code> returns value of <code>pathToBuildFolder</code> .

Scenario	Behavior
Location of artifact folder from previous release passed to <code>crossReleaseImport</code> : <pre>blockHandles = crossReleaseImport(pathToArtifact, configSet, ... 'Simulationmode', 'SIL');</pre>	Warning produced. From the artifact, <code>crossReleaseImport</code> extracts path to build folder and then imports code from the build folder.
Value of 'CodeLocation' passed to <code>crossReleaseImport</code> : <pre>blockHandles = crossReleaseImport(pathToArtifact, configSet, ... 'Simulationmode', 'SIL', ... 'CodeLocation', pathToAnchorFolder);</pre>	Warning produced. From the artifact, <code>crossReleaseImport</code> extracts path to build folder relative to an anchor, and then imports code from the build folder.
Location of artifact folder from previous release passed to <code>sharedCodeUpdate</code> : <pre>blockHandles = sharedCodeUpdate(pathToArtifact, pathToExistingSharedCode);</pre>	Warning produced. From the artifact, <code>sharedCodeUpdate</code> extracts path to shared code folder and then imports code from the shared code folder.
Value of 'CodeLocation' passed to <code>sharedCodeUpdate</code> : <pre>sharedCodeUpdate(pathToArtifact, pathToExistingSharedCode, ... 'CodeLocation', pathToAnchorFolder);</pre>	Warning produced. From the artifact, <code>sharedCodeUpdate</code> extracts path to shared code folder relative to an anchor, and then imports code from the shared code folder.

Import of code from previous release for code generation-only workflow

The `crossReleaseImport` function supports a new value for `SimulationMode`. If you specify the name-value pair, `SimulationMode`, 'none', the function creates a Cross-Release Code Integration block that:

- Supports generation of code that calls the imported code.
- Does not support simulation.

The function does not compile the imported code. You can use the block, for example, in workflows where compilation occurs on a different computer.

Maximum line width for generated code

You can specify the maximum line width for wrapping generated code. Select the **Configurations Parameters > Code Generation > Code Style > Maximum line width** parameter. The default value is 80. You can specify any integer in the range of 50–1000.

If the comments exceed the maximum line width specified, the tail comments are generated on a new line with right justification. These are not affected:

- `#define` tail comments
- Simulink block comments
- Stateflow object comments
- Banner comments

Symbolic dimension support for `%roll` directive

You can now write an S-function that has symbolic dimensions by using the `%roll` directive.

Before R2019a, you wrote a separate code path and checked every block property for symbolic dimensions, for example:

```
%function Outputs(block, system) Output
    %assign outputHasSymbolicWidth = LibBlockOutputHasSymbolicWidth(0)
    %if outputHasSymbolicWidth || LibBlockInputHasSymbolicWidth(0)
        %assign symbolicWidth = outputHasSymbolicWidth ...
            ? LibBlockOutputSignalSymbolicWidth(0) ...
            : LibBlockInputSignalSymbolicWidth(0)
        {
            int_T i0;
            for (i0 = 0; i0 < (%<symbolicWidth>); i0++) {
                %assign u0 = LibBlockInputSignal(0, "", "i0", 0)
                %assign k0 = LibBlockParameter(Gain, "", "i0", 0)
                %assign rhs = "%<k0> * %<u0>"
                %<LibBlockAssignOutputSignal(0, "", "i0", 0, rhs)>
            }
        }
    %else
        %assign rollVars = ["U", "Y", "P"]
        %roll sigIdx = RollRegions, lcv = 5, block, "Roller", rollVars
        %assign u0 = LibBlockInputSignal(0, "", lcv, sigIdx)
        %assign k0 = LibBlockParameter(Gain, "", lcv, sigIdx)
        %assign rhs = "%<k0> * %<u0>"
        %<LibBlockAssignOutputSignal(0, "", lcv, sigIdx, rhs)>
    %endroll
%endif
%endfunction
```

In R2019a, you do not have to maintain the separate code path, for example:

```
%function Outputs(block, system) Output
    %assign rollVars = ["U", "Y", "P"]
    %roll sigIdx = RollRegions, lcv = 5, block, "Roller", rollVars
    %assign u0 = LibBlockInputSignal(0, "", lcv, sigIdx)
    %assign k0 = LibBlockParameter(Gain, "", lcv, sigIdx)
    %assign rhs = "%<k0> * %<u0>"
    %<LibBlockAssignOutputSignal(0, "", lcv, sigIdx, rhs)>
%endroll
%endfunction
```

For details about configuring symbolic dimensions for S-functions, see [Configure Dimension Variants for S-Function Blocks](#).

Embedded Coder contextual tabs on the Simulink Toolstrip Tech Preview

In R2019a, you have the option to turn on the Simulink Toolstrip. For more information, see “Simulink Toolstrip Tech Preview replaces menus and toolbars in the Simulink Desktop”.

The Simulink Toolstrip includes contextual tabs, which appear only when you need them. The Embedded Coder contextual tabs include options for completing actions that apply only to Embedded Coder.

- To access the **C Code** tab, open the Embedded Coder app from the App gallery tab on the Simulink Toolstrip. If the **C++ Code** tab opens, select C code from the **Output** section of the gallery.
- To access the **C++ Code** tab, open the Embedded Coder app from the App gallery tab on the Simulink Toolstrip. If the **C Code** tab opens, select C++ code from the **Output** section of the gallery.
- To access the **SIL/PIL** tab, open the SIL/PIL Manager app from the App gallery tab on the Simulink toolstrip. Or, click **Verify > SIL/PIL Simulation** in the **C Code** tab.
- To access the **Hardware** tab, in the **C Code** tab, select **Verify > Run on Custom Hardware**.

Documentation does not reflect the addition of the Embedded Coder contextual tabs.

Deployment

Embedded Coder Support Package for PX4 Autopilots: Generate, build and deploy Simulink models on Pixhawk flight controllers

The Embedded Coder Support Package for PX4[®] Autopilots is available from release R2019a onwards. You can use the support package to generate, build, and deploy Simulink models on Pixhawk[®] series flight controllers.

The support package uses the PX4 toolchain, and includes a library of Simulink blocks that help you to develop PX4 autopilot models that use the various uORB topics, sensors, and PWM-based actuators.

DSP System Toolbox Support Packages for ARM Cortex -A and ARM Cortex -M Processors will be removed

Starting in R2019a, the DSP System Toolbox Support Package for ARM Cortex[®]-A Processors and DSP System Toolbox Support Package for ARM Cortex-M Processors will no longer be available for download. This functionality has been moved to Embedded Coder Support Package for ARM Cortex-A Processors and Embedded Coder Support Package for ARM Cortex-M Processors, respectively. For more details on installing and getting started with these support packages, see [Setup and Configuration \(Embedded Coder Support Package for ARM Cortex-A Processors\)](#) and [Setup and Configuration \(Embedded Coder Support Package for ARM Cortex-M Processors\)](#).

Performance

Reusable custom storage classes across referenced models

In R2019a, you can configure the code generation for models that have Reusable custom storage classes and referenced models by using the configuration parameter Detect non-reused custom storage classes (Simulink) on the **Data Validity** pane. This configuration reduces the number of global variables in the code and RAM usage.

The default behavior of the parameter settings varies with the presence of Reusable custom storage classes and referenced models.

When there are Reusable custom storage classes and referenced models present, the parameter settings are:

None

Simulink software generates a message for you to set the parameter to Error.

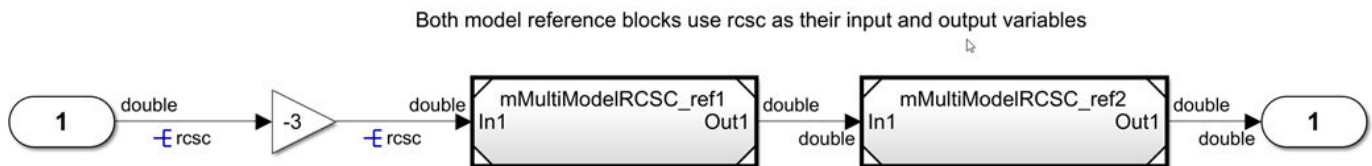
Warning

Simulink software generates a message for you to set the parameter to Error

Error

If Reusable custom storage classes can be combined Simulink software generates code. If not, it generates an error.

For example, the model MultiModelRCSC contains a Reusable custom storage class created with a void-void interface and a model reference.



This model reference reuses the custom storage class.



The code generated from the combined Reusable custom storage class is:

```
#include "mMultiModelRCSC.h"

/* Exported data definition */

/* Definition for custom storage class: Reusable */
real_T rcsc;
```



```

/* Model step function */
void mMultiModelRCSC_step(void)
{
    /* Gain: '<Root>/Gain' */
    rcsc = -3.0 * rcsc;

    /* ModelReference: '<Root>/Model' */
    mMultiModelRCSC_ref1();

    /* ModelReference: '<Root>/Model1' */
    mMultiModelRCSC_ref2();
}

```

The variable `rcsc` is reused with a gain of `-3.0` without additional global variable.

For models without a Reusable custom storage class shared among referenced models, the parameter functions the same way as it has in earlier releases.

Parallelization of execution of for-loops

In R2019a, you can compute `for`-loops in parallel by using multithreading to improve the speed of code execution of MATLAB Function, MATLAB System, and For Each blocks.

Consider a model like `parForExample` with a MATLAB Function block.



To turn on the parallel loop computation, in the Configuration Parameters dialog box on the **Optimization** pane, select the **Maximize execution speed** option from the **Priority** drop-down list. The parameter **Generate parallel for loops** is automatically selected. The parameter enables the compiler to compute loops in parallel.

The function inside the MATLAB Function block contains this code with the statement `parfor` for looping.

```

function y = access3a(u) %#codegen

% Copyright 2019 The MathWorks, Inc.

persistent pA;
if isempty(pA)
    pA = 0;
end
A = ones(20,50);
t = 0;

parfor (i = 1:10,4) % SIV - trivial
    A(i,1) = A(i,1) + 1;

```

end

$y = A(1,4) + u + t + pA;$

In R2019a, the code generator produces this code:

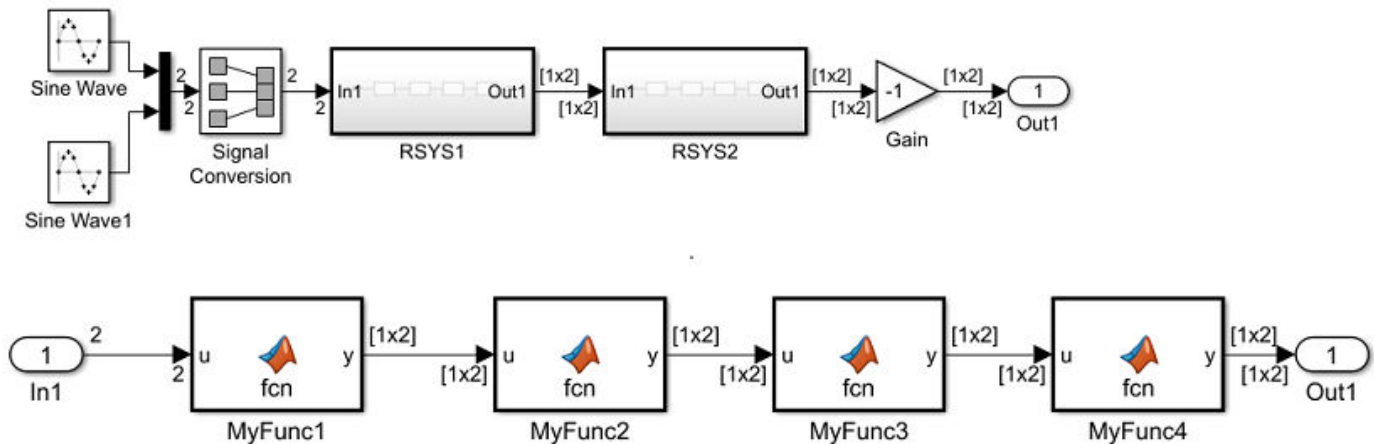
```
#pragma omp parallel for num_threads(4 >
    omp_get_max_threads() ? omp_get_max_threads() : 4)
```

In the generated code, the pragma instructs the compiler to execute the looping in parallel. This parallel execution results in an increase of the speed of execution of the generated code. For more information, see [Speed Up for-Loop Implementation in Code Generated by Using parfor](#).

Subsystem output with internal signals for buffer reduction

In R2019a, for more modeling patterns, the code generator can reuse variables for subsystem output signals and signals internal to the system. Reusing these variables conserves RAM consumption.

For example, the model `subsystem_out_reuse` contains the reusable subsystem `RSYS1`. This subsystem contains a series of connected MATLAB blocks.



In R2018b, the code generator produced this code:

```
/* Output and update for atomic system: '<Root>/RSYS1' */
void subsystem_out_reuse_RSYS1(const real_T rtu_In1[2], real_T rty_Out1[2])
{
    real_T rtb_y_d[2];
    real_T rtb_y_a[2];

    /* MATLAB Function: '<S1>/MyFunc1' */
    subsystem_out_reuse_MyFunc1(rtu_In1, rtb_y_d);

    /* MATLAB Function: '<S1>/MyFunc2' */
    subsystem_out_reuse_MyFunc2(rtb_y_d, rtb_y_a);

    /* MATLAB Function: '<S1>/MyFunc3' */
    subsystem_out_reuse_MyFunc2(rtb_y_a, rtb_y_d);

    /* MATLAB Function: '<S1>/MyFunc4' */
    subsystem_out_reuse_MyFunc2(rtb_y_d, rty_Out1);
}
```

The generated code contained two local buffers for holding values between the inputs and the outputs of each MATLAB Function block.

In R2019a, the code generator produces this code:

```
/* Output and update for atomic system: '<Root>/RSYS1' */
void subsystem_out_reuse_RSYS1(const real_T rtu_In1[2], real_T rty_Out1[2])
{
    real_T rtb_y_d[2];

    /* MATLAB Function: '<S1>/MyFunc1' */
    subsystem_out_reuse_MyFunc1(rtu_In1, rtb_y_d);

    /* MATLAB Function: '<S1>/MyFunc2' */
    subsystem_out_reuse_MyFunc2(rtb_y_d, rty_Out1);

    /* MATLAB Function: '<S1>/MyFunc3' */
    subsystem_out_reuse_MyFunc2(rty_Out1, rtb_y_d);

    /* MATLAB Function: '<S1>/MyFunc4' */
    subsystem_out_reuse_MyFunc2(rtb_y_d, rty_Out1);
}
```

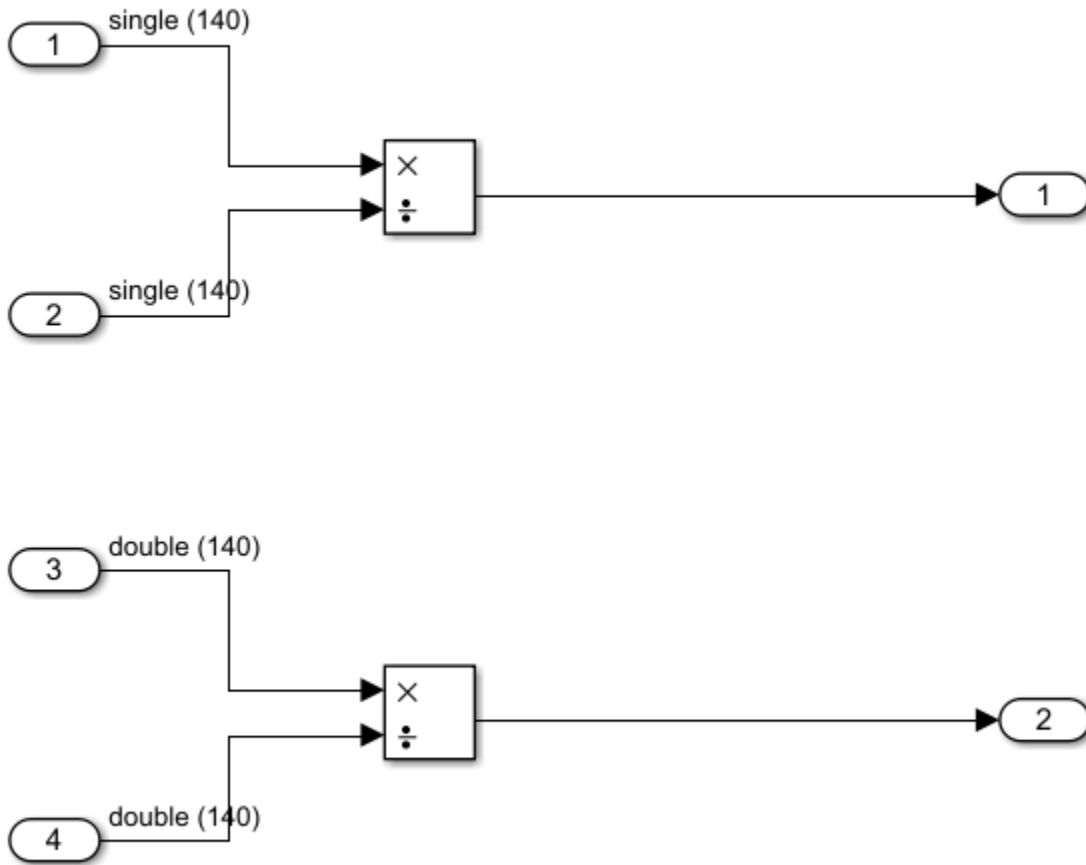
The generated code contains one local buffer for holding intermediate values between the inputs and outputs of each MATLAB Function block. For more information, see “Enable and Reuse Local Block Outputs in Generated Code”.

Optimized code execution speed for Single Instruction, Multiple Data (SIMD) intrinsic division operation

In R2019a, for Intel processors with SSE or AVX support, you can optimize and compute the division operation for models in parallel in the generated code by using SIMD intrinsics. To generate the code, in the Configuration Parameters dialog box, for the **Code replacement library** parameter, choose an Intel SSE or Intel AVX code replacement library.

The division operation is for element-wise arithmetic operations involving single and double data types.

Consider the model MDiv with inputs in single and double data type.



In R2018b, the code generator produced this code:

```
void mDiv_step(void)
{
    int32_T i;
    for (i = 0; i < 140; i++) {
        /* Output: '<Root>/Out2' incorporates:
        * Inport: '<Root>/In1'
        * Inport: '<Root>/In2'
        * Product: '<Root>/Divide'
        */
        mDiv_Y.Out2[i] = mDiv_U.In1[i] / mDiv_U.In2[i];

        /* Output: '<Root>/Out3' incorporates:
        * Inport: '<Root>/In5'
        * Inport: '<Root>/In6'
        * Product: '<Root>/Divide2'
        */
    }
}
```

```

    mDiv_Y.Out3[i] = mDiv_U.In5[i] / mDiv_U.In6[i];
  }
}

```

The code sequentially computes the for-loop 4 to 8 byte values at a time depending on whether the data type is single or double.

In R2019a, the code generator produces this code:

```

void mDiv_step(void)
{
  int32_T idx;
  __m128 tmp;
  __m128 tmp_0;
  __m128 tmp_1;
  __m128d tmp_2;
  __m128d tmp_3;
  __m128d tmp_4;
  for (idx = 0; idx <= 136; idx += 4) {
    /* Inport: '<Root>/In1' */
    tmp = _mm_loadu_ps(&mDiv_U.In1[idx]);

    /* Inport: '<Root>/In2' */
    tmp_0 = _mm_loadu_ps(&mDiv_U.In2[idx]);

    /* Outport: '<Root>/Out2' */
    tmp_1 = _mm_div_ps(tmp, tmp_0);
    _mm_storeu_ps(&mDiv_Y.Out2[idx], tmp_1);
  }

  for (idx = 0; idx <= 138; idx += 2) {
    /* Inport: '<Root>/In5' */
    tmp_2 = _mm_loadu_pd(&mDiv_U.In5[idx]);

    /* Inport: '<Root>/In6' */
    tmp_3 = _mm_loadu_pd(&mDiv_U.In6[idx]);

    /* Outport: '<Root>/Out3' */
    tmp_4 = _mm_div_pd(tmp_2, tmp_3);
    _mm_storeu_pd(&mDiv_Y.Out3[idx], tmp_4);
  }
}

```

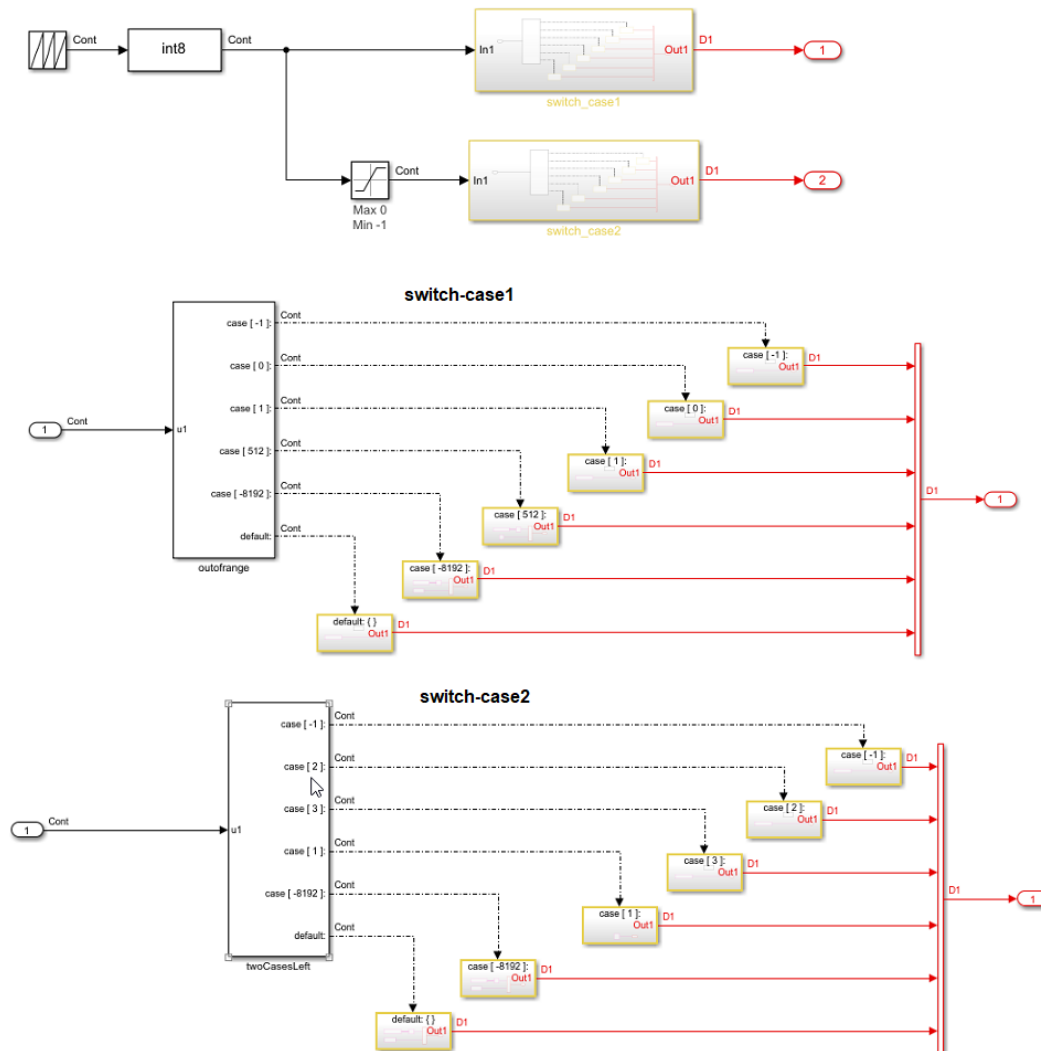
The code computes the division operation at 16 byte values at a time for single or double data types. The parallel computing increases the execution speed of the generated code. For more information, see Code replacement library (Simulink Coder).

Optimized code for Switch Case blocks

In R2018b, for Switch Case blocks, the generated code might have contained case statements that did not execute because the control expression in the switch statement did not equal the constant expression in the case statement. This unused code increased ROM consumption and reduced execution speed.

In R2019a, where possible, the code generator eliminates these unreachable case statements reducing ROM consumption and increasing execution speed.

For example, the model `switch_case_optimization` contains a Repeating Sequence block whose values range from -2 to 3. The output signal from this block directly connects to the subsystem `switch_case1`. This signal contains a branch to a Saturation block with a **Max** value of 0 and a **Min** value of -1. This signal connects to the subsystem `switch_case2`. These subsystems each contain a Switch-Case block that connects to Switch Case Action Subsystem blocks.



In R2018b, the code generator produced this code for the `outrange` and `twoCasesLeft` Switch Case blocks.

```

/* SwitchCase: '<S2>/outrange' */
switch (rtb_Saturation4) {
  case -1:
    rtAction = 0;
    break;

  case 0:
    rtAction = 1;
    break;

```

```

case 1:
    rtAction = 2;
    break;

case 512:
    rtAction = 3;
    break;

case -8192:
    rtAction = 4;
    break;

default:
    rtAction = 5;
    break;
}
...
/* SwitchCase: '<S3>/twoCasesLeft' */
switch (rtb_Saturation4) {
case -1:
    rtAction = 0;
    break;

case 2:
    rtAction = 1;
    break;

case 3:
    rtAction = 2;
    break;

case 1:
    rtAction = 3;
    break;

case -8192:
    rtAction = 4;
    break;

default:
    rtAction = 5;
    break;
}

```

For `outofrange`, the generated code contained six case statements even though cases 3 and -8192 did not execute because they are outside the input range of -2 through 3. For `twoCasesLeft`, the generated code contained six case statements even though only case -1: is within the input range of 0 through -1.

In R2019a, the code generator produces this code for the `outofrange` and `twoCasesLeft` blocks.

```

...
/* SwitchCase: '<S2>/outofrange' */
switch (rtb_Saturation4) {
case -1:
    rtAction = 0;
    break;

```

```
    case 0:
        rtAction = 1;
        break;

    case 1:
        rtAction = 2;
        break;

    default:
        rtAction = 5;
        break;
}
...
/* SwitchCase: '<S3>/twoCasesLeft' */
if (rtb_Saturation4 == -1) {
    rtAction = 0;
} else {
    rtAction = 5;
}
```

In R2019a, the generated code does not contain the `case` statements that do not execute. If only one case and the `default` statement remain, the generated code contains an `if-else` statement. For more information, see [Switch](#).

Removal of instrumentation overhead from execution-time profiling

R2019a provides improved execution-time profiling of generated code that is run on deterministic hardware.

- You can run a processor-in-the-loop (PIL) simulation that automatically discards the time overhead that the code instrumentation introduces.
- You can use the target hardware to estimate the average overhead value or you can specify the value manually.
- The software filters the execution times of AUTOSAR Runtime Environment (RTE) code and functions that run within Simulink, for example, Function Caller blocks.

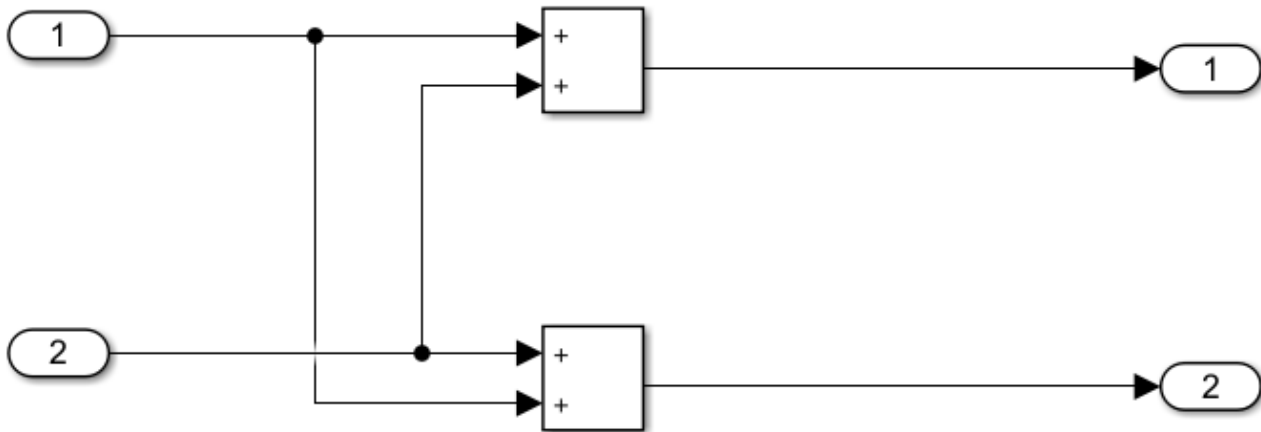
For more information, see:

- [Remove Instrumentation Overheads from Execution Time Measurements](#)
- `rtw.connectivity.Config`

Improvement in execution speed through common subexpression elimination

Previously, for models that contained redundant subexpressions that were numerically and logically equivalent, and commutative such as addition and multiplication, the generated code repeatedly calculated the value of the subexpression. In R2019a, the generated code contains a temporary variable that holds the value of these subexpressions and eliminates the redundant calculations. This optimization improves the execution speed of the generated code. The parameter **Eliminate superfluous local variables (expression folding)** enables this optimization.

Consider a model like `addExpr`.



In R2018b, the code generator produced this code:

```
void addExpr_step(void)
{
  /* Outputport: '<Root>/Out1' incorporates:
   * Inport: '<Root>/In1'
   * Inport: '<Root>/In2'
   * Sum: '<Root>/Add'
   */
  rtY.Out1 = rtU.In1 + rtU.In2;

  /* Outputport: '<Root>/Out2' incorporates:
   * Inport: '<Root>/In1'
   * Inport: '<Root>/In2'
   * Sum: '<Root>/Add1'
   */
  rtY.Out2 = rtU.In2 + rtU.In1;
}
```

The same addition operation occurred repeatedly and the values were stored in `rtY.Out1` and `rtY.Out2`.

In R2019a, the code generator produces this code:

```
void addExpr_step(void)
{
  real_T Out1_tmp;

  /* Sum: '<Root>/Add' incorporates:
   * Inport: '<Root>/In1'
   * Inport: '<Root>/In2'
   * Sum: '<Root>/Add1'
   */
  Out1_tmp = rtU.In1 + rtU.In2;

  /* Outputport: '<Root>/Out1' incorporates:
   * Sum: '<Root>/Add'
   */
  rtY.Out1 = Out1_tmp;

  /* Outputport: '<Root>/Out2' */
  rtY.Out2 = Out1_tmp;
}
```

```
}
```

The generated code contains the temporary variable `Out1_tmp` for holding the result of the addition operation thereby eliminating the redundancy. For more information, see [Eliminate superfluous local variables \(Expression folding\) \(Simulink Coder\)](#).

Data copy reduction in function calls

In R2019a, the code generator avoids large temporary buffers and redundant data copies of structures that are passed to MATLAB functions. Some of the coding patterns that you can optimize include passing a structure or an array as the first argument, and then an element of the structure or the array as the second argument of a function.

Consider a sample MATLAB code snippet:

```
#callee function

sfunction b = func1( a, b )
%#codegen
coder.inline('never');
b.result = single(0.0);
len = int32(length(b.array));
if a.start > int32(0) && a.start <= len && a.stop > int32(0) && a.stop <= len
    for n = a.start : a.stop
        b.result = b.result + (a.a1 + a.a2 + a.a3 + a.a4 + a.a5
                               * abs(b.array(n) * a.a6(n)));
    end;
else
    b.result = single(0);
end;
end

#caller function
function [ handle ] = topFunc( handle )
%#codegen
[handle.struct(1)] = func1(handle, handle.struct(1));
[handle.struct(2)] = func1(handle, handle.struct(2));
end
```

A structure `handle` and an element of the structure `handle.struct(1)` are passed to a function `func1`.

In R2018b, the code generator produced this C code for the preceding MATLAB script.

```
void topFunc(struct0_T *handle)
{
    static float t0_a6[1000000];
    memcpy(&t0_a6[0], &handle->a6[0], 1000000U * sizeof(float));
    func1(handle->start, handle->stop, handle->a1, handle->a2, handle->a3,
          handle->a4, handle->a5, t0_a6, &handle->b_struct[0]);
    memcpy(&t0_a6[0], &handle->a6[0], 1000000U * sizeof(float));
    func1(handle->start, handle->stop, handle->a1, handle->a2, handle->a3,
          handle->a4, handle->a5, t0_a6, &handle->b_struct[1]);
}
```

There are multiple data copies of the array `t0_a6`.

In R2019a, the code generator produces this C code instead:

```
void topFunc(struct0_T *handle)
{
    func1(handle, &handle->b_struct[0]);
    func1(handle, &handle->b_struct[1]);
}
```

The generated code does not contain multiple data copies, which improves the efficiency of the generated code.

Code generation for lookup table optimization

In R2019a, if you have a Simulink Check™ license, you can generate more efficient code for your model by using refactored Lookup Table blocks. For more information, see “Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks” (Simulink Check).

Verification

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2018b

Version: 7.1

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Column Limit in Generated Code: Generate more readable code by controlling line wrapping

In R2018b, to improve the readability of the generated code, you can specify the maximum number of columns before a line break.

In an Embedded Coder configuration object, set the `ColumnLimit` property to the maximum number of columns. For example:

```
cfg = coder.config('lib','ecoder',true);  
cfg.ColumnLimit = 120;
```

The default `ColumnLimit` value is 80.

The equivalent MATLAB Coder app setting is **Column limit** on the **All Settings** tab.

Other rules for placement of the line break can take precedence over the column limit that you specify.

Static Code Metrics On Demand: Run static code metrics analysis when needed after code generation

In previous releases, you had to request a static code metrics report before code generation. If you requested the static code metrics report, the static code metrics analysis ran at code generation time. In R2018b, you can request a static code metrics report, if and when you need it, after code generation. Code generation can be faster because the static code metrics analysis does not run at code generation time.

To produce a static code metrics report, you must generate standalone code by using Embedded Coder and enable production of a code generation report.

In previous releases, you requested a static code metrics report by using one of these methods:

- Specifying `-report` with `codegen`.
- In a configuration object, setting `GenerateCodeMetricsReport` to `true`.
- In the MATLAB Coder app, setting **Static code metrics** to **Yes**.

In R2018b, by default, in a configuration object, `GenerateCodeMetricsReport` is `false`. By default, in the app, **Static code metrics** is **No**. If you generate code with the default setting for static code metrics, static code metrics analysis does not run at code generation time. Instead, you can run the analysis and produce the code metrics report later by clicking **Code Metrics** on the **Summary** tab of a code generation report. For product and platform considerations for running the analysis after code generation, see [Generating a Static Code Metrics Report for Code Generated from MATLAB Code](#).

Compatibility Considerations

If you want to run static code metrics analysis at code generation time as in previous releases:

- In a configuration object, set `GenerateCodeMetricsReport` to `true`.
- In the MATLAB Coder app, set **Static code metrics** to Yes.

In previous releases, if you generated standalone code by using `codegen` with `-report`, static code metrics analysis always ran at code generation time, regardless of the value of `GenerateCodeMetricsReport`. In R2018b, if you generate standalone code by using `codegen` with `-report`, the analysis runs at code generation time only if `GenerateCodeMetricsReport` is `true`. If `GenerateCodeMetricsReport` is `false`, you can run the analysis later by clicking **Code Metrics** in the code generation report.

Single Instruction, Multiple Data (SIMD) Support: Generate Intel SSE/AVX intrinsic in MATLAB Coder

In R2018b, for element-wise arithmetic operations involving single and double data types, you can generate more efficient code that contains SIMD intrinsics. The code contains less data copies and no wrapper functions for the SIMD intrinsics. To generate this code, select the code replacement library to use for code generation in a project. On the **Custom Code** tab, set the **Code replacement library** parameter to one of these new SIMD code replacement libraries:

- Intel SSE (Windows)
- Intel AVX (Windows)
- Intel SSE (Linux)
- Intel AVX (Linux)

Alternatively, in a code configuration object, set the `CodeReplacementLibrary` parameter. Note that the code generator does not generate SIMD code for division operations.

Model Architecture and Design

Multi-Instance Code Generation: Generate multi-instance code for top and referenced models that are based on rates, exported functions, or rates and exported functions

R2018b expands the modeling styles from which the code generator produces code. Previously, you could not generate multi-instance (reentrant) code from models that used discrete and asynchronous sample rates. Now you can generate multi-instance code for top and referenced models that use:

- Discrete sample rates
- Asynchronous sample rates (for example, with exported function models)
- Discrete and asynchronous sample rates

For more information, see *Design Models for Generated Embedded Code Deployment*.

Code Preview in Embedded Coder Dictionary: Verify pseudocode preview as you select data, function, and memory section properties

R2018b adds a code preview capability to the Embedded Coder Dictionary for code definition verification. As you configure property settings of a code definition for data, a function, or a memory section, the code preview displays pseudocode that you can use to verify configuration results.

For more information, see:

- **Embedded Coder Dictionary**
- Define Storage Classes, Memory Sections, and Function Templates for Software Architecture
- Configure Default C Code Generation for Categories of Model Data and Functions

Embedded Coder Dictionary Mapping Control: Define storage classes that restrict mappings to parameters or signals

In R2018b, when defining a storage class in the Embedded Coder Dictionary, you can specify whether users can map the storage class to parameters, signals, or parameters and signals.

See *Constrain Use of Storage Class Code Mappings*.

Embedded Coder Dictionary Version Handling: Use and export code definitions saved in previous releases with models created in later releases

R2018b introduces Simulink data dictionary version handling, which includes version handling for the Embedded Coder Dictionary. You can:

- Link a model to a data dictionary that includes code definitions saved in a previous version of Simulink—for example, you can link a model that you develop with R2018b with a dictionary saved in R2018a.

- Continue developing a model, which has a local Embedded Coder Dictionary, that was created with a version of Simulink that is older than the current version---for example, if you started developing a model that uses a local Embedded Coder Dictionary in R2018a, you can open and continue development of that model in R2018b).
- Export (save) an Embedded CoderDictionary for use in models created with a previous release of the code generator.

For more information, see Dictionary Usage for Models Created with Different Versions of Simulink (Simulink) and **Embedded Coder Dictionary**.

AUTOSAR Run-Time Calibration: Map internal signals, states, and model workspace parameters to AUTOSAR component memory and internal parameters for calibration

Map internal signals and states to AUTOSAR per-instance and static memory for calibration

In R2018b, in an AUTOSAR model, you can map internal signals and states to AUTOSAR `ArTypedPerInstanceMemory` and `StaticMemory` for run-time calibration. Code Mapping Editor adds **Signals** and **States** tabs for mapping individual internal signals and states and configuring their attributes. For more information, see Map Block Signals and States to AUTOSAR Variables.

For referenced models within an AUTOSAR component model, Embedded Coder automatically maps internal signal and states for model reference code generation. Internal signals and states automatically map to AUTOSAR `ArTypedPerInstanceMemory` for multi-instance model reference, or AUTOSAR `StaticMemory` for single-instance model reference.

In the AUTOSAR Runtime Environment (RTE), calibration and measurement tools can access `ArTypedPerInstanceMemory` and `StaticMemory` generated from internal signals and states in the AUTOSAR top model and referenced models.

Map model workspace parameters to AUTOSAR component internal parameters for calibration

In R2018b, in an AUTOSAR model, you can map model workspace parameters to AUTOSAR component internal `SharedParameters` and `ConstantMemory` for run-time calibration. Code Mapping Editor adds a **Parameters** tab for mapping individual model workspace parameters, including lookup table and breakpoint parameters, and configuring their attributes. For more information, see Map Model Workspace Parameters to AUTOSAR Component Internal Parameters.

In the AUTOSAR Runtime Environment (RTE), calibration and measurement tools can access component internal `SharedParameters` and `ConstantMemory` generated from model workspace parameters in the AUTOSAR model.

Specify C type qualifiers for AUTOSAR static and constant memory

For an AUTOSAR component, you can configure C type qualifiers to customize generated AUTOSAR-compliant C code for AUTOSAR static memory and AUTOSAR constant memory. For example, you can apply C type qualifiers such as `const` or `volatile` to control compiler optimizations.

In R2018b, you can:

- Import C type qualifiers from `arxml` files into an AUTOSAR component model.

- In an AUTOSAR model, use Code Mapping Editor to configure C type qualifiers for model signals, states, and parameters that are mapped to AUTOSAR StaticMemory or AUTOSAR ConstantMemory.
- Build the model to export type qualifiers to arxml files and generate AUTOSAR-compliant C code that uses the type qualifiers.

For more information, see Specify C Type Qualifiers for AUTOSAR Static and Constant Memory.

AUTOSAR Memory Sections: Use SwAddrMethods to control memory placement of AUTOSAR runnable functions and internal data

AUTOSAR software components use software address methods (SwAddrMethods) to group data and function definitions in memory, primarily for efficiency, performance, and data access by run-time calibration tools. R2018b extends SwAddrMethod support to allow you to control the memory placement of runnable (entry-point) functions and runnable internal data, including signals, states, and parameters. In R2018b, you can:

- Add, delete, and modify SwAddrMethods in Simulink.
- Specify SwAddrMethods for runnable functions.
- Specify SwAddrMethods for runnable internal data.

When you build the AUTOSAR model, exported arxml descriptions for runnables reflect their SwAddrMethod associations. Generated AUTOSAR-compatible C code groups runnable functions and internal data together into memory sections based on the SwAddrMethod associations you configured.

For more information, see Configure AUTOSAR SwAddrMethods, Map Entry-Point Functions to AUTOSAR Runnables, and Configure SwAddrMethod.

AUTOSAR XML Import and Export: Round trip imported arxml file structure and control packaging of new elements

R2018b improves the round trip of imported AUTOSAR XML file structure and content.

- When you import arxml files for an AUTOSAR component into Simulink, Embedded Coder preserves the arxml file structure for export.
- AUTOSAR elements that you create in Simulink export to one or more *modelName*.arxml* files, which are separate from the imported files. As before, you control the file packaging of new elements by using AUTOSAR Dictionary parameter **Exported XML file packaging**.
- In the round-tripped arxml file content, import preserves AUTOSAR element UUIDs. If an imported element does not have a UUID, none is created.

With the improvements, you can more easily compare pre-import and post-export arxml files and track changes. Round-tripped arxml files are clearly differentiated from new arxml files. For example, if you import 20 arxml files into a model, and then use Simulink to define additional AUTOSAR interfaces and data, code generation exports the 20 imported files, along with *modelName*.arxml* files containing the new AUTOSAR interface and data elements.


For more information, see Round-Trip Preservation of AUTOSAR XML File Structure and Element Information.

AUTOSAR XML Import: Changes to ArTypedPerInstanceMemory and StaticMemory import behavior

In R2018b, to support AUTOSAR run-time measurement and calibration, you can map internal signals and states to AUTOSAR component memory and map model workspace parameters to AUTOSAR internal calibration parameters. R2018b enhances the `arxml` importer to support the new component data mapping, reduce clutter in the base workspace, and provide modeling flexibility.

- In general, importing AUTOSAR `ArTypedPerInstanceMemory`, `StaticMemory`, `SharedParameter`, or `ConstantMemory` elements no longer creates data objects in the Simulink base workspace or data dictionary. Scoping data to the AUTOSAR component better encapsulates component data for participation in system-level modeling.
- In general, importing AUTOSAR `ArTypedPerInstanceMemory` or `StaticMemory` elements no longer adds Data Store Memory blocks to the model. The software imports the elements more flexibly, without adding Data Store Memory blocks. You can decide whether to model the imported `ArTypedPerInstanceMemory` and `StaticMemory` elements by using signals, states, or data store memory.
- In one case, an `ArTypedPerInstanceMemory` element with a Service Dependency, the importer adds a Data Store Memory block and a corresponding `AUTOSAR.Signal` object to the model.

To use an imported `ArTypedPerInstanceMemory` or `StaticMemory` element in your AUTOSAR model, reference the SHORT-NAME of the imported element in the name of a block signal or block state.

Then update Simulink-to-AUTOSAR mapping by using the **Update** button  or the AUTOSAR function `autosar.api.syncModel`. You can use Code Mapping Editor, **Signals** or **States** tab, to view and configure the mapping of Simulink internal signals and states to AUTOSAR component memory.

For imported `ArTypedPerInstanceMemory` and `StaticMemory` elements, the importer creates Simulink signal objects in the model workspace. The signal objects are not required to model AUTOSAR component memory. You can use the signal objects in your model or remove them. Optionally, you can inspect the signal objects for their design properties, such as data type or min and max values.

Compatibility Considerations

If existing AUTOSAR infrastructure expects the importer to add Data Store Memory blocks for all imported `ArTypedPerInstanceMemory` and `StaticMemory` elements, update the infrastructure to reflect the new importer behavior. Alternatively, after the import completes, manually add Data Store Memory blocks and corresponding `AUTOSAR.Signal` objects to the AUTOSAR model.

Obsolete AUTOSAR signal and state map functions removed

In R2018b, you can individually map Simulink block signals and states to AUTOSAR `ArTypedPerInstanceMemory` or `StaticMemory` for AUTOSAR run-time calibration. This replaces an older mechanism in which you set the storage class of named signals and states to `Model default` to generate `ArTypedPerInstanceMemory`.

In R2018b, you can no longer access the following `autosar.api.getSimulinkMapping` functions, which are associated with the older signal and state mapping mechanism. The functions have been removed from MATLAB help.

<code>getDataDefaults</code>	Get AUTOSAR memory type for Simulink signals or discrete states
<code>mapDataDefaults</code>	Map Simulink signals or discrete states to AUTOSAR memory type

Compatibility Considerations

If an AUTOSAR script relies on the older signal and state map functions `getDataDefaults` and `mapDataDefaults`, update the script to use the new signal and state map functions, `getSignal`, `getState`, `mapSignal`, and `mapState`.

MISRA C:2012 and Secure Coding Standards: Improve compliance of generated code by using updated Model Advisor checks

Modifications to existing Model Advisor checks that you use to verify compliance with MISRA C:2012 and Secure Coding standards are outlined in this table.

Model Advisor Check	Description of Change
Check configuration parameters for MISRA C:2012 Check configuration parameters for secure coding standards	Checks now analyze the setting for these configuration parameters: <ul style="list-style-type: none"> • External mode • Undirected event broadcasts • Compile-time recursion limit for MATLAB functions • Enable run-time recursion for MATLAB functions
Check for blocks not recommended for MISRA C:2012 Check for blocks not recommended for secure coding standards	Checks now flag the usage of these blocks in a model or subsystem: <ul style="list-style-type: none"> • Compose String • Scan String • String to Double • String to Single • To String

Data, Function, and File Definition

Individual Function Mappings in Code Mapping Editor: Override default function mappings with individual function mappings

R2018b simplifies configuration of functions for code generation. From the Code Mapping Editor in the code perspective, you can configure default configurations for categories of functions. Then, from the same interface, you can:

- Review a list of the entry-point functions that you can configure for a model.
- Override the default configuration for a category of functions with configurations for individual functions.

You can configure each model entry-point function with a unique:

- Function customization template
- Function name
- Memory section

You can customize the interface arguments for the base-rate (first) step function for a model by opening the Configure C Step Function Interface dialog box directly from the Code Mapping Editor.

For example, for a model that has three step functions, one for each of three rates, you can configure the interface for each function differently.

For more information, see [Customize Generated C Function Interfaces](#).

Compatibility Considerations

Previously:

- You would open the dialog box for customizing the C step entry-point function interface by using the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Model Configuration Parameters dialog box. In R2018b, you open the dialog box from the **Code Mapping Editor**. See [Override Default C Step Function Interface](#).
- You would change the name of the C initialize entry-point function in the Configure C/C++ Function Interface dialog box. You opened that dialog box with the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Model Configuration Parameters dialog box. In R2018b, you change the names of entry-point functions, including the initialize function, in the **Code Mapping Editor**, on the **Entry-Point Functions** tab. See [Override Default Naming for Individual C Entry-Point Functions](#).

Function Interface Control: Access Configure C Step Function Interface dialog box from Code Mapping Editor in code perspective

For C code generation, configuration of interface arguments for the base rate step function of a model is more streamlined. You can open the Configure C Step Function Interface dialog box from the Code Mapping Editor in the code perspective.

For more information, see [Override Default C Step Function Interface](#).

Function Interface Control: Configure step functions for multi-instance, rate-grouped, single-tasking models

As of R2018b, for C code generation, you can configure the interface for the step function of a multi-instance, rate-grouped, single-tasking model. A multi-instance model is a top model with model configuration parameter **Code interface packaging** set to `Reusable` function and a referenced model with parameter **Total number of instances allowed per top model** set to `Multiple`. You configure the interface in the `Configure C Step Function Interface` dialog box, which is accessible from the `Code Mapping Editor` in the code perspective.

For more information, see `Generate Reentrant Code from Top Models and Override Default C Step Function Interface`.

Shared Default Code Configurations for Data and Functions: Share default code configuration settings between models

In R2018b, the code generator provides more flexibility for creating and managing Embedded Coder Dictionaries that models and modeling teams can share. You can set up an Embedded Coder Dictionary to configure default code definitions for categories of data and functions. If an Embedded Coder dictionary is shared between models in a Simulink data dictionary, all models that are linked to the data dictionary and have data or function categories mapped to storage class or function customization template setting `Dictionary Default` use the same coder dictionary defaults. If you make a change to the default settings in the shared Embedded Coder dictionary, the code generator applies the updated default settings to all models that meet both of these conditions:

- Are linked to the shared data dictionary
- Have data or functions configured to use coder dictionary defaults.

For more information, see `Configure Default Code Mapping in a Shared Dictionary`.

Storage Class on Root-Level I/O: Access global data and functions in multi-instance models

Currently, you cannot use a global storage class for signals and states in a multi-instance model. In R2018b, you can generate code for model reference and multi-instance top models that enable each model instance to access the same global data on its root-level I/O. Specify a global storage class on the root-level I/O of a multi-instance model to:

- Read from global data and access functions for root inputs.
- Write to global data and access functions for root outputs.

For more information, see `Use Storage Classes in Reentrant, Multi-Instance Models and Components`.

Code Generation

Code View in Code Perspective: View generated code directly in Code Perspective

In the Code Perspective, you can view your generated code alongside your model by using the Code view. This integration of the code and model in the Code Perspective helps you:

- Quickly navigate to locations in the model and the code.
- Understand the relationship between model elements and the code.
- Customize your generated code and check that the results are correct.

After generating code, open the Code Perspective. In the Code Perspective, select the **Code** tab in the bottom-right corner. If the **Code** tab is not available, from the editor menu, select **View > Code**. While customizing your code, the view enables you to:

- Search across all code files for function, variables, type definitions, and other code elements. Based on the text in the search field, the search tool can provide suggested searches.
- Trace from the model to the code. Highlight the code related to a model element by selecting the element in the model editor.
- Trace from the code to the model. Highlight the model element related to code by placing your cursor over or clicking a code element hyperlink.
- Navigate within the code. Locate where a function or variable is defined by placing your cursor over the code element. To go to the definition code, click the hyperlink in the information dialog box.
- Highlight lines of code that have changed since you last generated code.
- View the storage class mapping for model elements such as root inports and outports. Place your cursor over the corresponding variable declarations in the header file.

For an example of how to use the Code view during code customization, see [Override Default C Step Function Interface](#).

Data Coherency: Generate one variable for each Data Store read and write operation

In R2018b, for models containing read and write operations for Data Store Memory blocks, the generated code contains a single variable to hold the value for each Data Store Read and Write operation. Generating one variable instead of multiple variables improves data access coherency. To enable this feature, in the Configuration Parameters dialog box, on the **Interface** pane, select the **Implement each data store block as a unique access point** parameter. This parameter is new in R2018b. Its default setting is `off`. For further information see, [Implement each data store block as a unique access point \(Simulink Coder\)](#) and the example [Improve Data Coherency in Generated Code](#).

AUTOSAR Code Generation: Automatically generate AUTOSAR platform data types in C code

In R2018b, code generation for AUTOSAR models automatically generates AUTOSAR platform data types. For example, generated AUTOSAR-compliant C code uses AUTOSAR data types `sint8`, `uint8`,

sint16, uint16, sint32, uint32, float32, and float64 instead of Simulink code generation data types int8_T, uint8_T, int16_T, uint16_T, int32_T, uint32_T, real32_T, and real64_T.

Automatic AUTOSAR type generation allows you to generate AUTOSAR platform data types for top models, referenced models, and shared utilities without configuring Simulink data type replacement. For more information, see Automatic AUTOSAR Data Type Generation.

Data Type Replacement: Specify replacement types for 64-bit integers

You can specify replacement types for data types uint64 and int64. Create a Simulink.NumericType object to specify character vectors for the code generator to use as names for the data types. In the Configuration Parameters dialog box, on the **Code Generation > Data Type Replacement** pane, use **uint64** and **int64**. For more information, see Control Names of 64-Bit Integers.

You can control the data type limit identifiers in the generated code by using these fields under **Advanced Parameters**.

Data Type Limit Identifier Setting	Default Identifier	Command-Line Parameter
64-bit integer maximum identifier	MAX_int64_T	MaxIdInt64
64-bit unsigned integer maximum identifier	MAX_uint64_T	MaxIdUInt64
64-bit integer minimum identifier	MIN_int64_T	MinIdInt64

For more information, see Specify Boolean and Data Type Limit Identifiers.

Multi-Dimensional Arrays: Preserve array dimensions for parameters and lookup tables in generated code

By default, the code generator generates one-dimensional arrays in the C/C++ code for multi-dimensional model data. In R2018b, if the array layout of your model data is row-major, you can preserve dimensions of multidimensional arrays used in parameters and lookup tables in the generated code. Preserving array dimensions in generated code enhances integration with external code.

For example, consider matrix A.

$$A = \begin{matrix} & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{matrix}$$

Before R2018b, this is the generated code:

```
matrixParam[6] = {1, 4, 2, 5, 3, 6};
```

In R2018b, when you set the model configuration parameter **Array layout** (Simulink Coder) to Row-major, you can preserve dimensions in the generated code, which looks like:

```
matrixParam[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

For more information, see Dimension Preservation of Multidimensional Arrays and Preserve Dimensions of Multidimensional Arrays in Generated Code.

For details about row-major code generation, see Row-Major Array Layout: Simplify integration with external C/C++ code for Lookup Table and other blocks (Simulink Coder).

Custom Storage Classes

Preserve array dimensions of parameters for these custom storage classes:

- Const
- Volatile
- ConstVolatile
- ExportToFile
- ImportFromFile
- FileScope

For `Simulink.Parameter` and `Simulink.LookupTable`, you can enable the property **Preserve Array Dimensions** to preserve array dimensions.

Programmatically, to preserve array dimensions for a custom storage class, use these commands at the command prompt:

```
temp = Simulink.Parameter;
temp.CoderInfo.StorageClass = 'Custom';
temp.CoderInfo.CustomStorageClass = 'ExportToFile';
temp.CoderInfo.CustomAttributes.PreserveDimensions = 'Yes';
```

To preserve dimensions when you design your own custom storage class, use the new **Preserve Array Dimensions** property in the Custom Storage Class Designer. **Preserve Array Dimensions** has these options:

- **No**: Flattens the multi-dimensional array to one dimension in the generated code. This is the default option.
- **Yes**: Preserves array dimensions for all parameters with the specified custom storage class.
- **Instance Specific**: If you want to preserve array dimensions for each instance of the custom storage class. You can enable the **Preserve Array Dimensions** property on the parameter object. For Simulink package, this property is set to **Instance Specific** by default.

This property is not available in Embedded Coder Dictionary.

Stateflow Local Data

To preserve array dimensions for Stateflow local data, enable the model configuration parameter **Preserve Stateflow local data array dimensions** on the **Code Generation > Interface** pane. For more details, see Select Array Layout for Matrices in Generated Code (Stateflow)

AUTOSAR

For AUTOSAR target, if you set **Array layout** to **Row-major**, you can preserve dimensions of AUTOSAR parameters, lookup tables and Stateflow local data in the generated code.

Hardware Implementation Parameters: ProdHWDeviceType and TargetHWDeviceType are case-insensitive

In R2018b, the values for the ProdHWDeviceType and TargetHWDeviceType command-line parameters are case-insensitive. For example, these commands specify the same value for ProdHWDeviceType:

- `set_param(modelOrConfigurationSet, 'ProdHWDeviceType', 'atmel->avr')`
- `set_param(modelOrConfigurationSet, 'ProdHWDeviceType', 'Atmel->AVR')`

Enumerated Types: Optimizations in generated code

In R2018b, the generated code for enumerated types may contain these optimizations:

Appearance of casts

Depending on the size of integers that your current hardware supports, enumerated constants might contain type casting. For example:

Before R2018b	After R2018b
<pre>#ifndef DEFINED_TYPEDEF_FOR_enum_colors_int32 #define DEFINED_TYPEDEF_FOR_enum_colors_int32 typedef int32_T enum_colors_int32; #define red ((enum_colors_int32)1) #define blue ((enum_colors_int32)2) #define green ((enum_colors_int32)3) #define yellow ((enum_colors_int32)4) #endif</pre>	<pre>#ifndef DEFINED_TYPEDEF_FOR_enum_colors_int32_ #define DEFINED_TYPEDEF_FOR_enum_colors_int32_ typedef int32_T enum_colors_int32; /* enum enum_colors_int32 */ #define red (1) /* Default value */ #define blue (2) #define green (3) #define yellow (4) #endif</pre>

Appearance of unsigned integer

If the base type of an enumerated constant is an unsigned integer, the enumerated constant value might contain the appendix U. For example:

Before R2018b	After R2018b
<pre>#ifndef DEFINED_TYPEDEF_FOR_enum_motorspeed_uint8 #define DEFINED_TYPEDEF_FOR_enum_motorspeed_uint8 typedef uint8_T enum_motorspeed_uint8; #define off ((enum_motorspeed_uint8)1) #define slow ((enum_motorspeed_uint8)2) #define medium ((enum_motorspeed_uint8)3) #define fast ((enum_motorspeed_uint8)4) #endif</pre>	<pre>#ifndef DEFINED_TYPEDEF_FOR_enum_motorspeed_uint8_ #define DEFINED_TYPEDEF_FOR_enum_motorspeed_uint8_ typedef uint8_T enum_motorspeed_uint8; /* enum enum_motorspeed_uint8 */ #define off ((enum_motorspeed_uint8)1U) #define slow ((enum_motorspeed_uint8)2U) #define medium ((enum_motorspeed_uint8)3U) #define fast ((enum_motorspeed_uint8)4U) #endif</pre>

Enumerated comments

Comments for enumerated elements appear before the typedef. For example:

Before R2018b	After R2018b
<pre> #ifndef DEFINED_TYPEDEF_FOR_dCodeGenEnum1_ #define DEFINED_TYPEDEF_FOR_dCodeGenEnum1_ typedef enum { a = 0 /* Default value */ b, c } dCodeGenEnum1; /* First enumerated data type */ #endif </pre>	<pre> #ifndef DEFINED_TYPEDEF_FOR_dCodeGenEnum1_ #define DEFINED_TYPEDEF_FOR_dCodeGenEnum1_ /* First enumerated data type */ typedef enum { a = 0 /* Default value */ b, c } dCodeGenEnum1; #endif </pre>

For more information on enumerated data types, see Enumeration.

Deployment

Texas Instruments C2000: Use DMA and CAN blocks for all supported C28x devices with the addition of DMA for F28x7x/F28004x and CAN for F28004x

Texas Instruments C2000 F28x7x and F28004x processors support direct memory access (DMA). The F28004x processor also supports the Controller Area Network (CAN) protocol. For more information, see C28x-DMA_ch# (Embedded Coder Support Package for Texas Instruments C2000 Processors), C28x CAN Calibration Protocol, C28x eCAN Receive, and C28x eCAN Transmit.

Code Generation Assumptions: Use standalone workflow to run checks

With the new `buildStandaloneCoderAssumptions` function, you can use a standalone workflow to check code generation assumptions with reference to your target hardware. Previously, to perform the checks, you ran a processor-in-the-loop (PIL) simulation. With the new workflow, you can perform the checks before PIL target connectivity is available.

The code generation report displays the list of assumptions that you can check, which includes assumptions about flush-to-zero (FTZ) and denormals-are-zero (DAZ) subnormal numbers.

For more information, see [Check Code Generation Assumptions](#).

Build Process: Library and header files for model reference hierarchy are not copied

Previously, the build process copied:

- Model reference library files to the build folder for the parent model
- Model reference header files to the `referenced_model_includes` subfolder of the build folder for the parent model.

In R2018b, the build process does not copy model reference library or header files. The build process creates a response file for the header file paths.

If you want the build process to copy model reference header files to the `.../parentModel/referenced_model_includes` subfolder, set these new custom toolchain attributes to `true`:

- `NoCompilerCommandFile`
- `CopyReferencedModelHeaders`

For more information, see [addAttribute](#).

The build argument `MODELREF_LINK_LIBS` is not supported. For example, the `getBuildArgs` function does not extract the `MODELREF_LINK_LIBS` argument identifier and value from a build information object.

The `MODELREF_LINK_LIBS` template makefile (TMF) token is still supported.

If you run a MATLAB script that uses the `getBuildArgs` function to extract an argument identifier and value for `MODELREF_LINK_LIBS`, the script might fail.

Build Process: MATLAB_INCLUDES is not required in custom template makefiles

The MATLAB_INCLUDES macro is not required in custom template makefiles. In R2018b, the build process extracts the required include paths from a build information object. You do not have to remove the macro from existing template makefiles.

STM32F7 Tuning and Monitoring: Perform external mode simulation on STM32F7 for parameter tuning and signal monitoring by using XCP over TCP/IP or UART (Serial)

The Embedded Coder Support Package for STMicroelectronics Discovery Boards (STM32F746G and STM32F769I) supports external mode simulation for parameter tuning and signal monitoring using Universal Measurement and Calibration Protocol (XCP) over TCP/IP or UART as the transport layer. XCP-based external mode enables signal monitoring using Simulation Data Inspector and Dashboard blocks.

Performance

Execution-Time Profiling: Specify profiling granularity through model-wide and block-specific controls

In a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, you can control the granularity of execution-time profiling with these parameters:

- `CodeProfilingInstrumentation` -- This modified configuration parameter provides model-wide control with three options:
 - `'off'` -- No function-level instrumentation, so execution times for functions in generated code are not collected.
 - `'coarse'` -- Measure execution times only for function code generated from referenced models and atomic subsystems.
 - `'detailed'` -- Measure execution times for all functions in generated code.
- `CodeProfilingOverride` -- This new block parameter provides control at the block-level with three options:
 - `'off'` -- Disable profiling for block.
 - `'on'` -- Enable profiling for the block if profiling is enabled for the parent model.
 - `'inherit'` (default) -- Apply profiling settings of parent block.

Changing the block profiling configuration does not cause the regeneration of production code.

For more information, see [Code Execution Profiling with SIL and PIL](#).

Compatibility Considerations

Previously, the `CodeProfilingInstrumentation` configuration parameter supported only two options, `'on'` and `'off'`. When you load a model that you created in a previous release, R2018b updates the value of `CodeProfilingInstrumentation`.

Value in Previous Release	Value in R2018b
<code>'off'</code>	<code>'off'</code>
<code>'on'</code>	<code>'detailed'</code>

In R2018b, if you run `set_param(modelName, 'CodeProfilingInstrumentation', 'on')`, the function sets `'CodeProfilingInstrumentation'` to `'detailed'` and produces a warning.

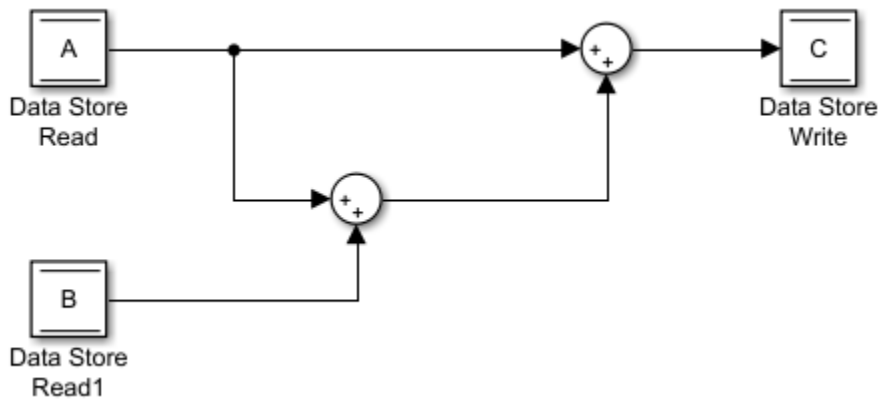
`CodeProfilingOverride` replaces the **Tag** block property value, `DoNotProfile`. In R2018b, `DoNotProfile` is still supported, but the SIL or PIL simulation produces a warning. In a future release, `DoNotProfile` will not be supported.

Global Variable Caching: Reduce access for global variable arrays with custom storage classes

In R2018a, when you set the **Optimize global data access** parameter to `Minimize Global Data Access`, the code generator reduced scalar global variable accesses by generating local variables. In

R2018b, the code generator can generate local variables to reduce access for global variable arrays with custom storage classes. This optimization improves execution speed because the code for local variable references has a smaller overhead than the code for global variable references.

For example, the `catching_example` model contains two read operations from the global data store A. A is a Simulink.Parameter with a custom storage class.



In R2018a, when you generate code with the **Optimize global data access** parameter set to **Minimize global data access**, you get this code in the `catching_example.c` file.

```

static int16_T A[2];
static int16_T B;
static int16_T C[2];
RT_MODEL_catching_example_T catching_example_M;
RT_MODEL_catching_example_T *const catching_example_M = &catching_example_M;
void catching_example_step(void)
{
    int32_T i;
    for (i = 0; i < 2; i++) {
        C[i] = (int16_T)((int16_T)(A[i] + A[i]) + B);
    }
}

```

For accessing global data, the generated code contains the global array A. The reads occur to this global variable.

In R2018b, when you generate code with the **Optimize global data access** parameter set to **Minimize global data access**, you get this code in the `catching_example.c` file.

```

static int16_T A[2];
static int16_T B;
static int16_T C[2];
RT_MODEL_catching_example_T catching_example_M;
RT_MODEL_catching_example_T *const catching_example_M = &catching_example_M;
void catching_example_step(void)
{
    int32_T i;
    int16_T A_0;
    for (i = 0; i < 2; i++) {
        A_0 = A[i];
        C[i] = (int16_T)((int16_T)(A_0 + A_0) + B);
    }
}

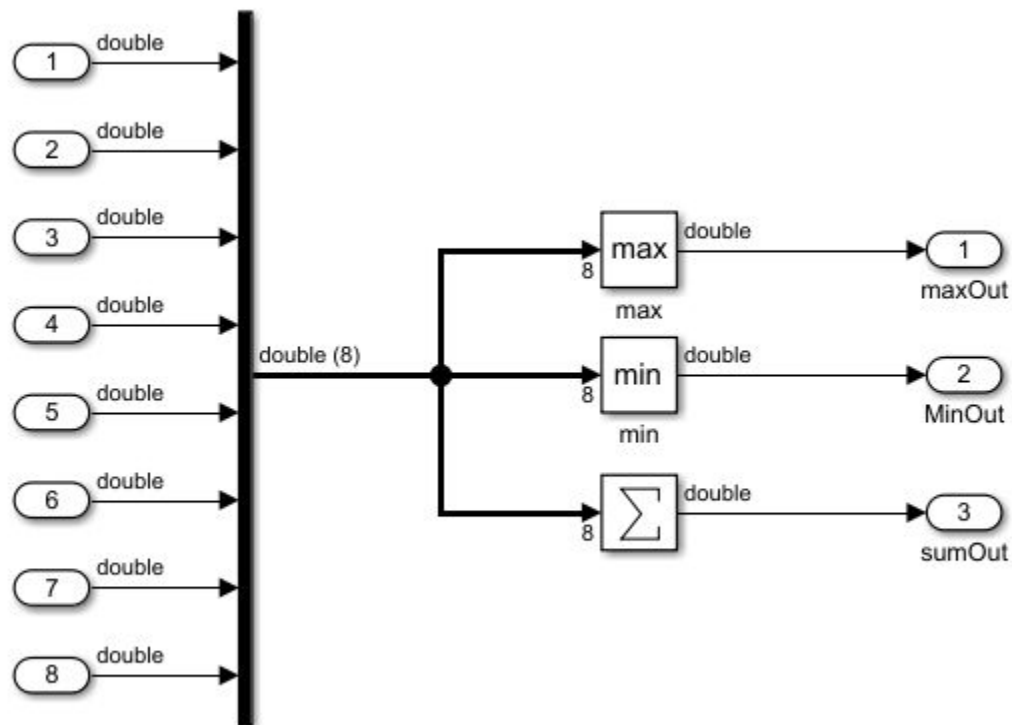
```

The data reads are to the local scalar variable `A_0` instead of the global array `A`. For more information, see [Optimize global data access \(Simulink Coder\)](#) and [Optimize Global Variable Usage](#).

Data Copy Reduction: Eliminate unnecessary data copies for Mux blocks

Previously, for models that contained Mux blocks followed by branching, extra data copies were in the generated code. In R2018b, the code generator improves execution speed by eliminating these data copies.

For example, the model `multiplex_ex` contains eight signals feeding into a Mux block. The result is a 1-D vector signal with a width of eight.



In R2018a, the code generator produced this code:

```
void multiplex_ex_step(void)
{
    real_T tmpForInput[8];
    int32_T i;
    real_T tmp;
    tmpForInput[0] = rtU.In1;
    tmpForInput[1] = rtU.In2;
    tmpForInput[2] = rtU.In3;
    tmpForInput[3] = rtU.In4;
    tmpForInput[4] = rtU.In5;
    tmpForInput[5] = rtU.In6;
    tmpForInput[6] = rtU.In7;
    tmpForInput[7] = rtU.In8;
    tmp = rtU.In1;
    for (i = 0; i < 7; i++) {
```

```

        if (!(tmp > tmpForInput[i + 1])) {
            tmp = tmpForInput[i + 1];
        }
    }

    rtY.maxOut = tmp;
    tmpForInput[0] = rtU.In1;
    tmpForInput[1] = rtU.In2;
    tmpForInput[2] = rtU.In3;
    tmpForInput[3] = rtU.In4;
    tmpForInput[4] = rtU.In5;
    tmpForInput[5] = rtU.In6;
    tmpForInput[6] = rtU.In7;
    tmpForInput[7] = rtU.In8;
    tmp = rtU.In1;
    for (i = 0; i < 7; i++) {
        if (!(tmp < tmpForInput[i + 1])) {
            tmp = tmpForInput[i + 1];
        }
    }

    rtY.sumOut = tmp;
    rtY.MinOut = tmp;
    tmpForInput[0] = rtU.In1;
    tmpForInput[1] = rtU.In2;
    tmpForInput[2] = rtU.In3;
    tmpForInput[3] = rtU.In4;
    tmpForInput[4] = rtU.In5;
    tmpForInput[5] = rtU.In6;
    tmpForInput[6] = rtU.In7;
    tmpForInput[7] = rtU.In8;
    tmp = -0.0;
    for (i = 0; i < 8; i++) {
        tmp += tmpForInput[i];
    }

    rtY.sumOut = tmp;
}

```

The code contained three data copies to each element of the temporary array `tmpForInput`.

In R2018b, the code generator produces this code:

```

#include "multiplex_ex.h"

ExtU rtU;
ExtY rtY;
void multiplex_ex_step(void)
{
    int32_T i;
    real_T tmp;
    real_T tmpForInput_tmp[8];
    real_T u1_tmp;
    tmpForInput_tmp[0] = rtU.In1;
    tmpForInput_tmp[1] = rtU.In2;
    tmpForInput_tmp[2] = rtU.In3;
    tmpForInput_tmp[3] = rtU.In4;
    tmpForInput_tmp[4] = rtU.In5;

```

```
tmpForInput_tmp[5] = rtU.In6;
tmpForInput_tmp[6] = rtU.In7;
tmpForInput_tmp[7] = rtU.In8;
tmp = rtU.In1;
for (i = 0; i < 7; i++) {
    u1_tmp = tmpForInput_tmp[i + 1];
    if (tmp <= u1_tmp) {
        tmp = u1_tmp;
    }
}

rtY.maxOut = tmp;
tmp = rtU.In1;
for (i = 0; i < 7; i++) {
    u1_tmp = tmpForInput_tmp[i + 1];
    if (tmp >= u1_tmp) {
        tmp = u1_tmp;
    }
}

rtY.sumOut = tmp;
rtY.MinOut = tmp;
tmp = -0.0;
for (i = 0; i < 8; i++) {
    tmp += tmpForInput_tmp[i];
}

rtY.sumOut = tmp;
}
```

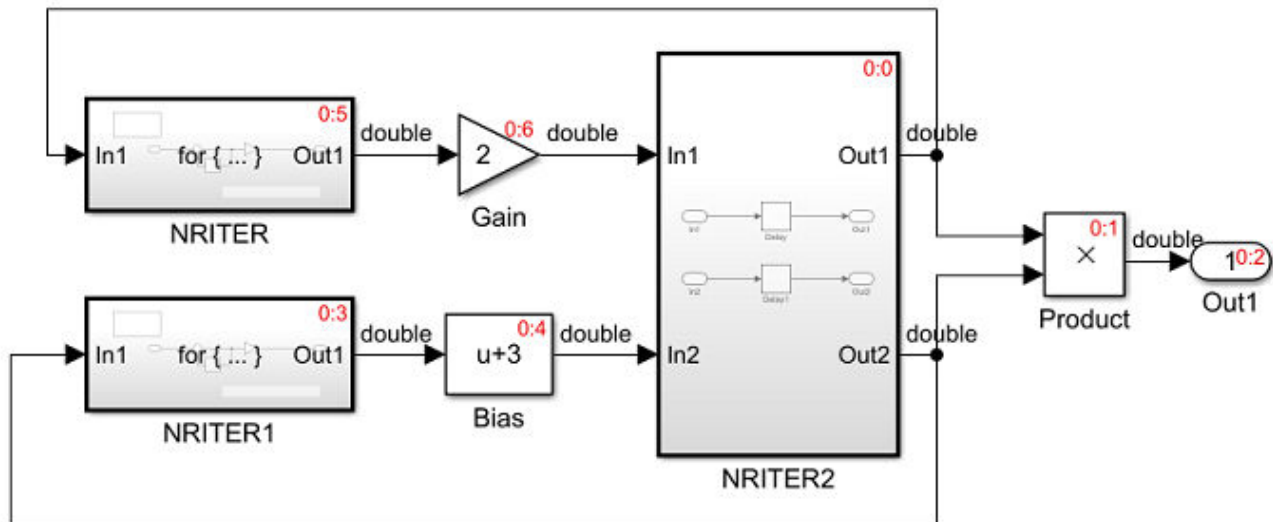
The code contains one data copy instead of three data copies to each element of the temporary array tmpForInput_tmp.

Enhanced Buffer Reuse: Buffer reuse across the boundary of an Iterator subsystem

In R2018a, for global signals, the code generator could not reuse an Iterator subsystem output with signals outside of the Iterator subsystem. In R2018b, for global and local Iterator subsystem outputs, the code generator can reuse these variables. Reusing these variables reduces memory usage.

For example, the model foriterator_ex contains three subsystems. In the subsystem block parameters dialog box, the **Function packaging** parameter is set to **Nonreusable function** and

the **Function interface** parameter is set to void-void.



In R2018a, the generated code was the following:

```
static void NRITER(void)
{
    int32_T s1_iter;
    real_T rtb_Sum;
    for (s1_iter = 0; s1_iter < 5; s1_iter++) {
        rtb_Sum = rtDW.Delay + rtDW.Delay_DSTATE_i[0];
        rtDW.IterOut_c = -rtb_Sum;
        rtDW.Delay_DSTATE_i[0] = rtDW.Delay_DSTATE_i[1];
        rtDW.Delay_DSTATE_i[1] = rtb_Sum;
    }
}

static void NRITER1(void)
{
    int32_T s2_iter;
    real_T rtb_Sum;
    for (s2_iter = 0; s2_iter < 5; s2_iter++) {
        rtb_Sum = rtDW.Delay1 + rtDW.Delay_DSTATE_h[0];
        rtDW.IterOut = -rtb_Sum;
        rtDW.Delay_DSTATE_h[0] = rtDW.Delay_DSTATE_h[1];
        rtDW.Delay_DSTATE_h[1] = rtb_Sum;
    }
}

...
void foriterator_ex_step(void)
{
    NRITER2();
    rtY.Out1 = rtDW.Delay * rtDW.Delay1;
    NRITER1();
    rtDW.Bias = rtDW.IterOut + 3.0;
    NRITER();
}
```

```

    rtDW.Gain = 2.0 * rtDW.IterOut_c;
    NRITER2_Update();
}

```

The generated code contains two different global variables to hold the NRITER and NRITER1 subsystem outputs.

In R2018b, the generated code is the following:

```

static void NRITER(void)
{
    int32_T s1_iter;
    real_T rtb_Sum;
    for (s1_iter = 0; s1_iter < 5; s1_iter++) {
        rtb_Sum = rtDW.Delay + rtDW.Delay_DSTATE_i[0];
        rtDW.IterOut = -rtb_Sum;
        rtDW.Delay_DSTATE_i[0] = rtDW.Delay_DSTATE_i[1];
        rtDW.Delay_DSTATE_i[1] = rtb_Sum;
    }
}

static void NRITER1(void)
{
    int32_T s2_iter;
    real_T rtb_Sum;
    for (s2_iter = 0; s2_iter < 5; s2_iter++) {
        rtb_Sum = rtDW.Delay1 + rtDW.Delay_DSTATE_h[0];
        rtDW.IterOut = -rtb_Sum;
        rtDW.Delay_DSTATE_h[0] = rtDW.Delay_DSTATE_h[1];
        rtDW.Delay_DSTATE_h[1] = rtb_Sum;
    }
}
...
void foriterator_ex_step(void)
{
    NRITER2();
    rtY.Out1 = rtDW.Delay * rtDW.Delay1;
    NRITER1();
    rtDW.Bias = rtDW.IterOut + 3.0;
    NRITER();
    rtDW.Gain = 2.0 * rtDW.IterOut;
    NRITER2_Update();
}

```

The generated code contains the same variable `rtDW.IterOut` to hold the two subsystem outputs. As a result, in R2018b, the DW global structure contains one less global variable than in R2018a.

Code Replacement: Optimize generated code with SIMD and row-major order support and code replacement enhancements

R2018b includes these code replacement enhancements:

- Single Instruction, Multiple Data (SIMD) support for the MATLAB environment and enhanced for the Simulink environment to include multidimensional signals, square root operations, and operations between scalars and vectors.

- New `Inlined ARM NEON Intrinsics` code replacement library for floating-point arithmetic operations. Use the library to inline code that you generate for ARM Cortex-A processors.
- For the Simulink environment, there is now support for row-major order:
 - **Array layout supported by entry** menu in the Code Replacement Tool for creating row-major code replacement table entries. The menu appears when you set **Argument type** to `Matrix`. You can set **Array layout supported by entry** to `Column-major` (default), `Row-major`, or `Column-and-Row`.
 - Code replacement table entry property `ArrayLayout` for specifying row-major order programmatically. You can set the property to `COLUMN_MAJOR`, `ROW_MAJOR`, or `COLUMN_AND_ROW`.
 - For lookup table and interpolation functions, you can get and set the algorithm parameter `UseRowMajorAlgorithm` with calls to `getAlgorithmParameters` and `setAlgorithmParameters`.

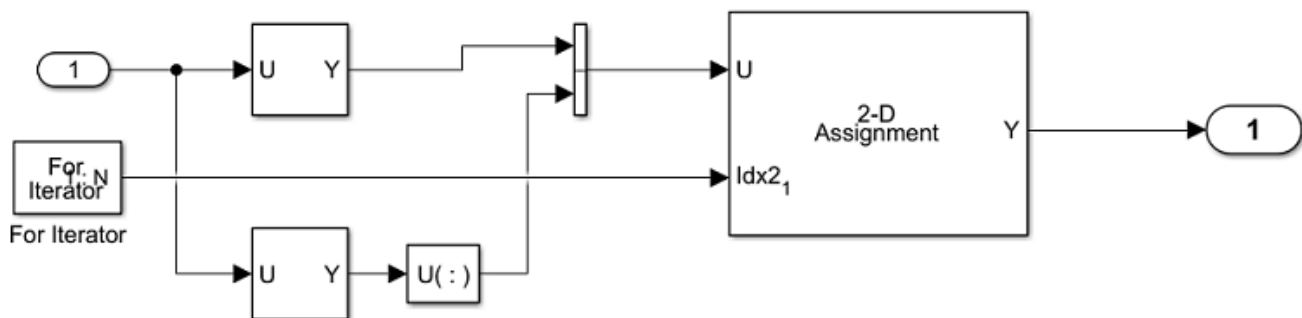
See `Define Code Replacement Mappings`, `Array Layout and Code Replacement`, and `Lookup Table Function Code Replacement`.

- Code replacement for shape-agnostic matrix addition, subtraction, and element-wise multiplication. When you enable this option, if the total number of matrix elements matches the code replacement library match criteria, code replacement occurs for shape-agnostic matrix operations. For more information, see `Code Replacement Match and Replacement Process`.

Inplace Optimization for Assignment Blocks: Reduce data copies for Assignment blocks

Previously, for Assignment blocks, there was an extra temporary variable and associated data copy in the generated code. In R2018b, when you select the `Perform inplace updates for Bus Assignment blocks (Simulink Coder)` parameter, the code generator can remove this data copy. This optimization increases code execution speed and conserves RAM consumption.

For example, the model `assign_model` contains an Assignment block.



In R2018a, the code generator produced this code in the `assign_model_step` function.

```
void assign_model_step(void)
{
    int32_T s1_iter;
    real_T rtb_VectorConcatenate[6];
    static const int8_T tmp[3] = { 0, 2, 3 };

```

```

int32_T i;
for (s1_iter = 0; s1_iter < 6; s1_iter++) {
    for (i = 0; i < 3; i++) {
        rtb_VectorConcatenate[i] = rtU.Inl[8 + tmp[i]];
        rtb_VectorConcatenate[3 + i] = rtU.Inl[((i << 1) + 1) << 2];
    }

    for (i = 0; i < 6; i++) {
        rtY.Outl[i + 6 * s1_iter] = rtb_VectorConcatenate[i];
    }
}
}

```

The code contains an extra variable `rtb_VectorConcatenate` for holding intermediate values.

In R2018b, the code generator produces this code in the `assign_model_step` function.

```

void assign_model_step(void)
{
    int32_T s1_iter;
    static const int8_T tmp[3] = { 0, 2, 3 };

    int32_T s1_iter_0;
    int32_T i;
    for (s1_iter = 0; s1_iter < 6; s1_iter++) {
        s1_iter_0 = s1_iter * 6;
        for (i = 0; i < 3; i++) {
            rtY.Outl[i + s1_iter_0] = rtU.Inl[8 + tmp[i]];
        }

        s1_iter_0 = s1_iter * 6 + 3;
        for (i = 0; i < 3; i++) {
            rtY.Outl[i + s1_iter_0] = rtU.Inl[((i << 1) + 1) << 2];
        }
    }
}

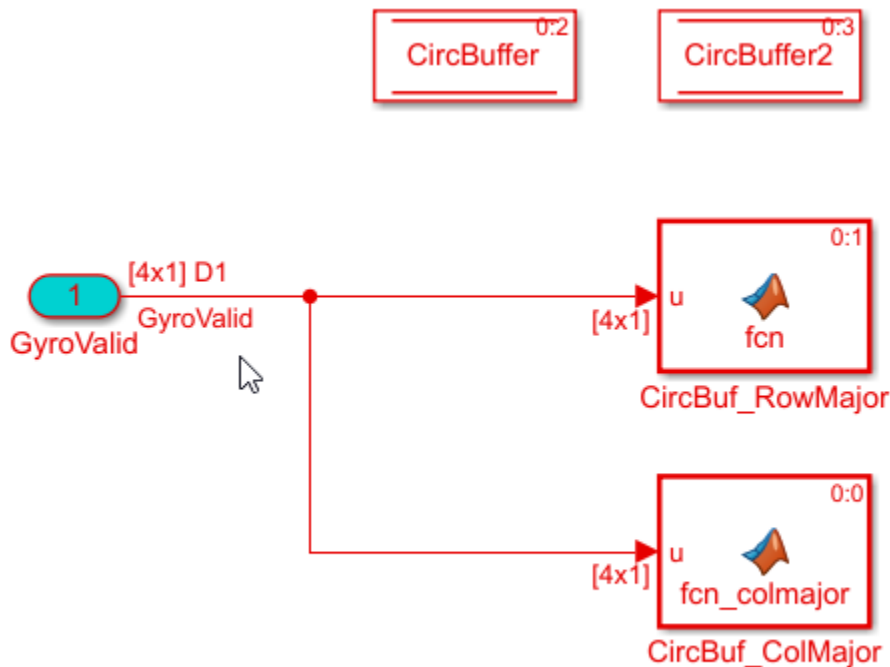
```

The `rtb_VectorConcatenate` variable and the associated data copy is not in the generated code.

Execution Speed: Eliminate redundant subexpressions

Previously, for models that contained redundant subexpressions consisting of left shift (<<) and right shift (>>) operators, the generated code repeatedly calculated the value of the subexpression. In R2018b, the generated code contains a temporary variable that holds the value of these subexpressions. This optimization improves the execution speed of the generated code because it eliminates redundant calculations. The parameter **Eliminate superfluous local variables (expression folding)** enables this optimization.

For example, the model `inplace_exp` contains two MATLAB functions, which result in the generated code containing left shift operators in the array index.



In R2018a, the `inplace_exp_step` function contained this code:

```

RT_MODEL_inplace_exp_T inplace_exp_M;
RT_MODEL_inplace_exp_T *const inplace_exp_M = &inplace_exp_M;
real_T CircBuffer[480];
real_T CircBuffer2[480];
real_T GyroValid[4];
void inplace_exp_step(void)
{
    int32_T i;
    CircBuffer2[0] = GyroValid[0];
    CircBuffer2[1] = GyroValid[1];
    CircBuffer2[2] = GyroValid[2];
    CircBuffer2[3] = GyroValid[3];
    CircBuffer[0] = GyroValid[0];
    CircBuffer[120] = GyroValid[1];
    CircBuffer[240] = GyroValid[2];
    CircBuffer[360] = GyroValid[3];
    for (i = 0; i < 119; i++) {
        CircBuffer2[(i + 1) << 2] = CircBuffer2[i << 2];
        CircBuffer2[1 + ((i + 1) << 2)] = CircBuffer2[(i << 2) + 1];
        CircBuffer2[2 + ((i + 1) << 2)] = CircBuffer2[(i << 2) + 2];
        CircBuffer2[3 + ((i + 1) << 2)] = CircBuffer2[(i << 2) + 3];
        CircBuffer[i + 1] = CircBuffer[i];
        CircBuffer[i + 121] = CircBuffer[120 + i];
        CircBuffer[i + 241] = CircBuffer[240 + i];
        CircBuffer[i + 361] = CircBuffer[360 + i];
    }
}

```

The same left shift operation occurred repeatedly.

In R2018b, the code generator contains this code:

```
void inplace_exp_step(void)
{
    int32_T i;
    int32_T CircBuffer2_tmp;
    int32_T CircBuffer2_tmp_0;
    CircBuffer2[0] = GyroValid[0];
    CircBuffer2[1] = GyroValid[1];
    CircBuffer2[2] = GyroValid[2];
    CircBuffer2[3] = GyroValid[3];
    CircBuffer[0] = GyroValid[0];
    CircBuffer[120] = GyroValid[1];
    CircBuffer[240] = GyroValid[2];
    CircBuffer[360] = GyroValid[3];
    for (i = 0; i < 119; i++) {
        CircBuffer2_tmp = i << 2;
        CircBuffer2_tmp_0 = (i + 1) << 2;
        CircBuffer2[CircBuffer2_tmp_0] = CircBuffer2[CircBuffer2_tmp];
        CircBuffer2[1 + CircBuffer2_tmp_0] = CircBuffer2[CircBuffer2_tmp + 1];
        CircBuffer2[2 + CircBuffer2_tmp_0] = CircBuffer2[CircBuffer2_tmp + 2];
        CircBuffer2[3 + CircBuffer2_tmp_0] = CircBuffer2[CircBuffer2_tmp + 3];
        CircBuffer[i + 1] = CircBuffer[i];
        CircBuffer[i + 121] = CircBuffer[120 + i];
        CircBuffer[i + 241] = CircBuffer[240 + i];
        CircBuffer[i + 361] = CircBuffer[360 + i];
    }
}
```

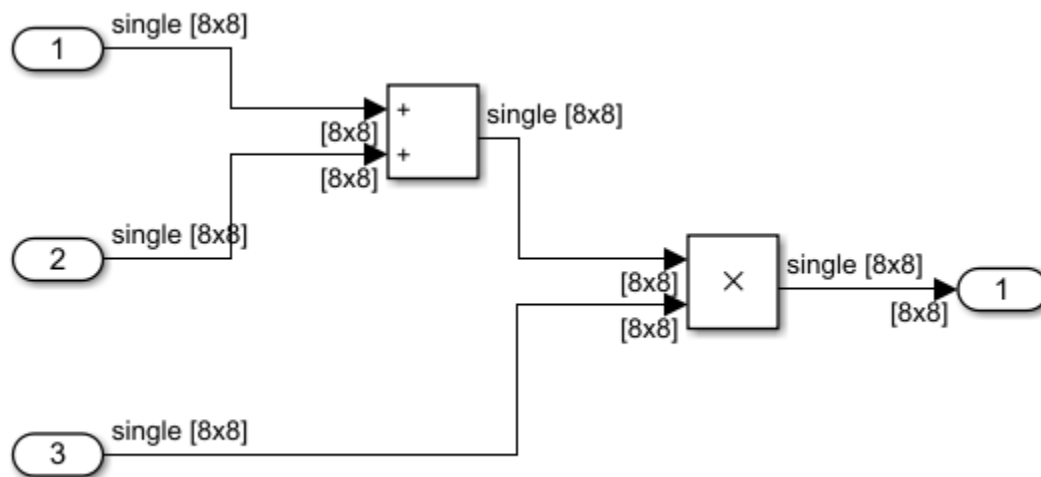
The generated code contains the temporary variables `CircBuffer2_tmp` and `CircBuffer2_tmp_0` for holding the result of the left shift operation. For more information, see [Eliminate superfluous local variables \(Expression folding\) \(Simulink Coder\)](#).

Single Instruction, Multiple Data (SIMD) Intrinsics: Generate code with optimized load and store operations for multidimensional signals and square root operations

In R2018a, for Intel processors with SSE support, you generated code with optimized load and store functions that utilized SIMD instructions. In the Configuration Parameters dialog box, for the **Code replacement library** parameter, you generated this code by choosing an Intel SSE or Intel AVX code replacement library. The optimized load and store functions were for element-wise arithmetic operations involving single and double data types. For more information, see [Single Instruction, Multiple Data \(SIMD\) Intrinsics: Generate code with optimized load and store operations for use with Intel processors with SSE/AVX support](#).

In R2018b, for models containing multidimensional signals, square root operations, and operations between scalars and vectors, you can generate code containing optimized load and store functions.

For example, the model `mMultiDimAddMultiply` performs addition and multiplication operations on multidimensional signals.



In R2017b, when you chose an Intel IPP/SSE code replacement library, the `mMultiDimAdd_step` function contained this code:

```
void mMultiDimAddMultiply_step(void)
{
    __declspec(align(16)) real32_T rtb_Add2[64];
    mw_gcc_sse_mm_add_f32x4(mMultiDimAddMultiply_U.In1, 8, 8,
        mMultiDimAddMultiply_U.In2, rtb_Add2);
    mw_gcc_sse_mm_add_f32x4(rtb_Add2, 8, 8, mMultiDimAddMultiply_U.In3,
        mMultiDimAddMultiply_Y.Out1);
}
```

The `mw_gcc_sse_mm_add_f32x4` function is a wrapper function for the load and store functions. The code also contains the buffer `rtb_Add2`. In R2018a, when you chose an Intel SSE or Intel AVX library, the code generator did not generate the wrapper functions or the optimized load and store functions for models containing multidimensional signals and square root operations.

In R2018b, when you choose an Intel SSE or Intel AVX code replacement library, the `mMultiDimAdd_step` function contains this code:

```
void mMultiDimAddMultiply_step(void)
{
    int32_T idx;
    __m128 tmp;
    __m128 tmp_0;
    __m128 tmp_1;
    for (idx = 0; idx <= 60; idx += 4) {
        tmp = _mm_loadu_ps(&mMultiDimAddMultiply_U.In1[idx]);
        tmp_0 = _mm_loadu_ps(&mMultiDimAddMultiply_U.In2[idx]);
        tmp_1 = _mm_add_ps(tmp, tmp_0);
        tmp = _mm_loadu_ps(&mMultiDimAddMultiply_U.In3[idx]);
        tmp_0 = _mm_mul_ps(tmp_1, tmp);
        _mm_storeu_ps(&mMultiDimAddMultiply_Y.Out1[idx], tmp_0);
    }
}
```

In R2018b, there is no function wrapper for the SIMD intrinsics. The `mMultiDimAdd_step` function contains the load and store functions. The buffer `rtb_Add2` is not in the generated code. For more information on Code Replacement Libraries, see [What Is Code Replacement?](#)

Code Generation Report: Generate static code metrics reports faster

In R2018b, for some models, when you create code generation reports and select the **Static code metrics** model configuration parameter, the code generator creates reports faster than in R2018a. This improvement is particularly noticeable for models that contain approximately 80 or more referenced models.

Functionality Being Removed or Changed

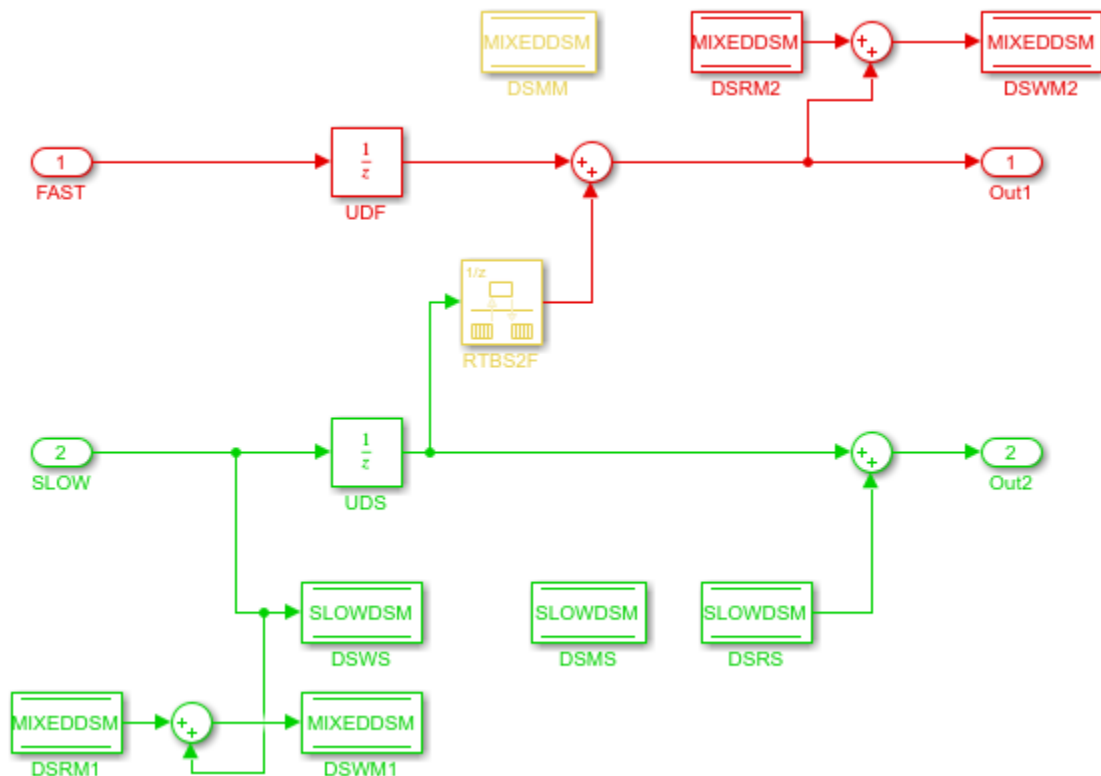
In R2018b, the configuration parameter **Parameter structure** is being removed. **Parameter structure** or `InlinedParameterPlacement` controlled how parameter data was generated for reusable subsystems.

Starting in R2018b, Embedded Coder generates a single, flat parameter data structure. Subsystem parameters are defined as fields within the structure. This type of nonhierarchical data structure can reduce compiler padding between word boundaries in memory, producing more efficient compiled code. For more information, see `Parameter Data Types in the Generated Code`.

Cache Efficiency: Store global block signal and state data operating at the same rate in one data structure

In R2018b, for models operating at multiple rates, you can store global block signal data (block I/O) and global state data (DWork vectors) operating at the same rate in one data structure. Storing this data in one structure improves cache efficiency when deploying a multirate model to multiple cores. To enable this feature, select the **Generate separate internal data per entry-point function** parameter, which is new in R2018b. Selecting the **Combine signal/state structures** parameter enables the **Generate separate internal data per entry-point function** parameter.

For example, the `multirate_demo` model operates at two different rates as the colors red and green in the diagram indicate. The color yellow is hybrid rate. The **Treat each discrete rate as a separate task** parameter is set to on.



With the **Generate separate internal data per entry-point function** parameter set to off, the `multirate_demo.h` file contains this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
    real_T RTBS2F; /* '<Root>/RTBS2F' */
    real_T UDS; /* '<Root>/UDS' */
    real_T Sum3; /* '<Root>/Sum3' */
    real_T Sum1; /* '<Root>/Sum1' */
    real_T UDF_DSTATE; /* '<Root>/UDF' */
    real_T UDS_DSTATE; /* '<Root>/UDS' */
    real_T RTBS2F_Buffer0; /* '<Root>/RTBS2F' */
    real_T MIXEDDSM; /* '<Root>/DSMM' */
    real_T SLOWDSM; /* '<Root>/DSMS' */
} DW_multirate_demo_T;
```

The data for each task is in one DWorks (DW_) structure.

With the **Generate separate internal data per entry-point function** set to on, the `multirate_demo.h` file contains this code:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
    real_T RTBS2F_Buffer0; /* '<Root>/RTBS2F' */
    real_T MIXEDDSM; /* '<Root>/DSMM' */
} DW_multirate_demo_T;
```

```
/* Internal Data Grouped For Same Function, for system '<Root>' */
```

```
typedef struct {
    real_T RTBS2F;          /* '<Root>/RTBS2F' */
    real_T Sum3;           /* '<Root>/Sum3' */
    real_T UDF_DSTATE;     /* '<Root>/UDF' */
} FuncInternalData0_multirate_demo_T;

/* Internal Data Grouped For Same Function, for system '<Root>' */
typedef struct {
    real_T UDS;            /* '<Root>/UDS' */
    real_T Sum1;          /* '<Root>/Sum1' */
    real_T UDS_DSTATE;    /* '<Root>/UDS' */
    real_T SLOWDSM;       /* '<Root>/DSMS' */
} FuncInternalData1_multirate_demo_T;
```

For each task, the generated code contains two global structures. The prefixes `FuncInternalData0_` and `FuncInternalData1_` indicate the two rates. The hybrid data is in the `DWorks (DW_)` structure.

Verification

SIL and PIL Simulations: Advanced custom storage classes support

Using the Custom Storage Class Designer, you can create an advanced custom storage class (CSC) when you set **Type** to `Other`. In R2018b, if you create a custom attributes class for the CSC and associate the custom attributes class with a Boolean property (`SupportSILPIL`) that is set to `true`, you can run software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations to test generated code that uses the advanced CSC.

For more information, see:

- Imported Data and Function Definitions
- Further Customize Generated Code by Writing TLC Code
- Finely Control Data Representation by Writing TLC Code for a Custom Storage Class

SIL and PIL Simulations: Support for imported grouped custom storage classes

SIL and PIL simulations support signals, parameters, and data stores with imported grouped custom storage classes. For information about storage classes, see [Choose Storage Class for Controlling Data Representation in Generated Code](#).

Model Block SIL and PIL: Accelerator mode SIM target is not built

Previously, a Model block SIL or PIL simulation also built a SIM target, which is required only for an accelerator mode simulation. In R2018b, a Model block SIL or PIL simulation does not build the SIM target, which provides these benefits:

- In an empty working folder, the simulation takes less time.
- If you use custom code, cross-target compatibility is not a requirement for the custom code.
- Aliased data types for input and output ports of a Model block are preserved.

For more information, see [Model Reference Simulation Targets \(Simulink\)](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2018a

Version: 7.0

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Interactive Traceability: Visualize mapping between MATLAB code and C code

In R2018a, if you generate C/C++ code by using MATLAB Coder with Embedded Coder, you can interactively trace between MATLAB code and generated C/C++ code. You can trace from the MATLAB code to the C/C++ code or from the C/C++ code to the MATLAB code. Tracing can help you understand how the code generator implemented your algorithm, debug issues in the generated code, and evaluate the quality of the generated code.

To enable tracing, in the code generation report, click **Trace Code**.

You see the generated code and the original MATLAB code next to each other. As you move your pointer over MATLAB code or C code, you see highlighted traces to the corresponding generated code or to the original MATLAB code.

```

38 while j > 0 && ~unbounded
39     [i,r] = findPivotRow(j,A,b);
40     if i == 0 || isinf(r)
41         unbounded = true;
42     else
43         b(i) = r;
44         % Take account of the variables entering and leaving
45         entering = nIdx(j);
46         leaving = bIdx(i);
47         bIdx(i) = entering;
48         nIdx(j) = leaving;
49         [cn,A,b] = pivot(cn,A,b,i,j);
50         j = findPivotColumn(cn,nIdx);
51     end
52 end
53 % Construct the solution.
54 if unbounded
55     x = nan(n,1);
56     y = inf;

```

```

218     iv0[j] = 1 + j;
219 }
220
221 j = findPivotColumn(c, iv0);
222 unbounded = false;
223 while ((j > 0) && (~unbounded)) {
224     findPivotRow(j, A, b, &i, &r);
225     if ((i == 0) || rtIsInf(r)) {
226         unbounded = true;
227     } else {
228         b[i - 1] = r;
229
230         /* Take account of the variables entering and lea
231         leaving = bIdx[i - 1];
232         bIdx[i - 1] = nIdx[j - 1];
233         nIdx[j - 1] = leaving;
234         pivot(cn, A, b, i, j);
235         j = findPivotColumn(cn, nIdx);
236     }
237 }

```

For an example of interactive tracing, see [Interactively Trace Between MATLAB Code and Generated C/C++ Code](#).

Polyspace Integration: Verify C/C++ code generated with MATLAB Coder by using simplified workflow

In R2018a, Polyspace® verification is integrated into the MATLAB Coder workflow. If you have Polyspace and Embedded Coder, you can run Polyspace in the MATLAB Coder app without additional setup. At the command line, after code generation with `codegen`, you can run Polyspace on the generated code by providing the code generation output folder to `pslinkrun`. For more information about running Polyspace verification on code generated with MATLAB Coder, see [Polyspace Verification of C/C++ Code Generated by MATLAB Coder](#).

Changes to Setup for MISRA C Compliance: Disable dynamic memory allocation and set C standard math library to C99 (ISO)

Starting in R2017b, in one step, you were able to set up code generation parameters to increase the likelihood of generating code that is compliant with MISRA C®. In R2018a, the setup has these changes:

- The standard math library for C code is C99 (ISO) instead of C89/C90 (ANSI). For C++ code, the math library is still C++03 (ISO).
- Dynamic memory allocation is disabled.

For more information, see [Increase Likelihood of Generating MISRA C Compliant Code from MATLAB Code](#).

Compatibility Considerations

For compatibility with code that depends on calls to the C89/C90 (ANSI) library, after calling `coder.setupMISRAConfig` at the command line or using the **MISRA Compliance** option in the app, change the standard math library to C89/C90 (ANSI).

If your code requires dynamic memory allocation, after calling `coder.setupMISRAConfig` at the command line or using the **MISRA Compliance** option in the app, enable dynamic memory allocation.

Model Architecture and Design

Embedded Coder Dictionary: Create custom code generation definitions for data and functions

In R2018a, you can specify default code generation settings for a category of model elements. With these default settings, you do not need to explicitly configure each element in a model. For more information, see “Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model” on page 10-14.

To standardize the code that you and your users generate from multiple models, you can create and share custom code generation definitions. When you and your users specify default settings for a model, your custom definitions appear available for selection alongside the built-in definitions. For example, you can create a storage class that appears alongside built-in storage classes, such as `ExportedGlobal`, in the **Code Mappings > Data Defaults > Storage Class** drop-down list.

Creating custom definitions can also enable you to achieve code generation goals that the built-in definitions cannot satisfy.

You can create these kinds of custom definition:

- Storage classes, which control the code generated for data—signals, states, and parameters.
- Function customization templates, which control the names of model entry-point functions such as the execution function `model_step`.
- Memory sections, which control the placement of data and function code in memory by inserting pragmas and other decorations in the generated code.

If you need to use your custom definitions in only one model, you can store the definitions in the model file. Alternatively, to share the definitions between models and projects, store the definitions in a Simulink data dictionary. With a data dictionary, to modify a shared definition, you make changes in only one place—the dictionary.

If you defined storage classes and memory sections in a previous release by creating your own package, you can configure a model or a Simulink data dictionary to refer to the package. Then, the package storage classes and memory sections appear in the Code Mapping Tool alongside new definitions that you create in R2018a.

To create new definitions, you use the Embedded Coder Dictionary. For more information, see Embedded Coder Dictionary.

When you or your users open the new Code Perspective for the first time in a model created in a previous release, the Perspective can make changes to the model and associated data dictionaries (`.sldd`). Before you or your users open the Code Perspective in existing models, consider preparing the models and dictionaries for the changes. See Migrate Memory Section and Shared Utility Settings from Configuration Parameters to Code Mapping Editor.

Multi-Instance Code Generation: Apply more control when generating reusable, reentrant code

R2018a introduces these capabilities for controlling multi-instance code generation, that is code generated when you set **Code interface packaging** to **Reusable function**:

- The Embedded Coder Quick Start now provides the option of whether you want to configure a model for multi-instance code generation. For a multi-instance C language configuration, the tool sets the model configuration parameter **Code interface packaging** to **Reusable function**.
- New example storage classes, `SignalStruct` and `ParamStruct`, which facilitate controlling generated code for signals and parameters. Embedded Coder Quick Start and Embedded Coder model templates apply these storage classes by default when you specify multi-instance code generation. You can update properties, such as naming rules for instance-specific data. The code generator produces a `struct` type definition that encapsulates signal or parameter data.
- New `$G` token for including the name of a storage class in code generation naming rules associated with model data elements. For example, you can use this token in the naming rule that you specify for the header file defined for a storage class.
- The Embedded Coder Dictionary enables you to define storage classes intended for multi-instance code generation, for example, you can specify a structured storage type and include the storage class name in the name of the generated header file.

For more information about generating multi-instance, reentrant code, see [Generate Reentrant Code from Top Models](#).

Variant Blocks Usability Enhancement: Generate Preprocessor Conditionals by using MATLAB variables as variant controls

In R2017b, to generate preprocessor conditionals for Variant blocks, you specified variant control variables as `Simulink.Parameter` objects. These objects were required to have one of these storage classes:

- `Define` or `ImportedDefine` with header file specified
- `CompilerFlag`
- `SystemConstant` (AUTOSAR)
- Your own custom storage class that defined data as a macro

In R2018a, for Variant blocks, you can specify variant control variables as MATLAB variables and generate preprocessor conditionals. You no longer have to convert MATLAB variables that you use for simulation to `Simulink.Parameters` for code generation with preprocessor conditionals. For more information, see [Variant Systems](#).

MISRA C:2012 Compliance and Deviation Considerations: Guidance for evaluating your generated code for compliance with MISRA C:2012 directives and rules

When using MISRA C:2012 coding guidelines to evaluate the quality of your generated C code, you are required per section 5.3 of the *MISRA C:2012 Guidelines for the Use of C Language in Critical Systems* document to prepare a compliance statement for the project being evaluated.

To assist you in the development of this compliance statement, MathWorks evaluates C code generated by using Embedded Coder against the MISRA C:2012 guidelines. Compliance considerations are documented in:

- [Using This Documentation When Developing a MISRA C:2012 Compliance Statement](#)
- [Evaluate Your Generated Code for MISRA C:2012 Compliance](#)

- Compliance Information Summary Tables
- Modeling Guidelines for MISRA C:2012 Compliance
- Deviations Rationale

Modeling Checks: Improve compliance of generated code by using Model Advisor check for MISRA C:2012

Use this new check to verify compliance of your generated code with MISRA C:2012 standards. To execute this check, open Model Advisor and select **By Product > Embedded Coder**.

Model Advisor Check	Description	Addresses Standards
Check bus object names that are used as element names	Check updated to identify Simulink.Bus object names that are used as Simulink.Bus element names.	<ul style="list-style-type: none"> • MISRA C:2012 Rule 5.6 • MISRA AC AGC Rule 5.3

Modifications to existing compliance checks are outlined in this table.

Model Advisor Check	Description of Change
Check for bitwise operations on signed integers	The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can produce incorrect results.
Check for blocks not recommended for C/C++ production code deployment	Check now analyzes content in library linked blocks and masked subsystems.
Check for blocks not recommended for MISRA C:2012	

For information about MISRA C versions and updates, see MISRA C Guidelines.

AUTOSAR Release 4.3: Import and export AUTOSAR XML schema version 4.3

The software now supports AUTOSAR Release 4.3 (schema version 4.3.0) for import and export of arxml files and generation of AUTOSAR-compatible C code.

4.3 is now the default value for the model configuration parameter **Generate XML file for schema version**.

For more information, see Select an AUTOSAR Schema.

AUTOSAR Perspective: Map and configure software components by using Code Mapping Editor and AUTOSAR Dictionary

After you create an AUTOSAR software component model in Simulink, use the Code Mapping Editor and AUTOSAR Dictionary to further develop the AUTOSAR component. The Code Mapping Editor and

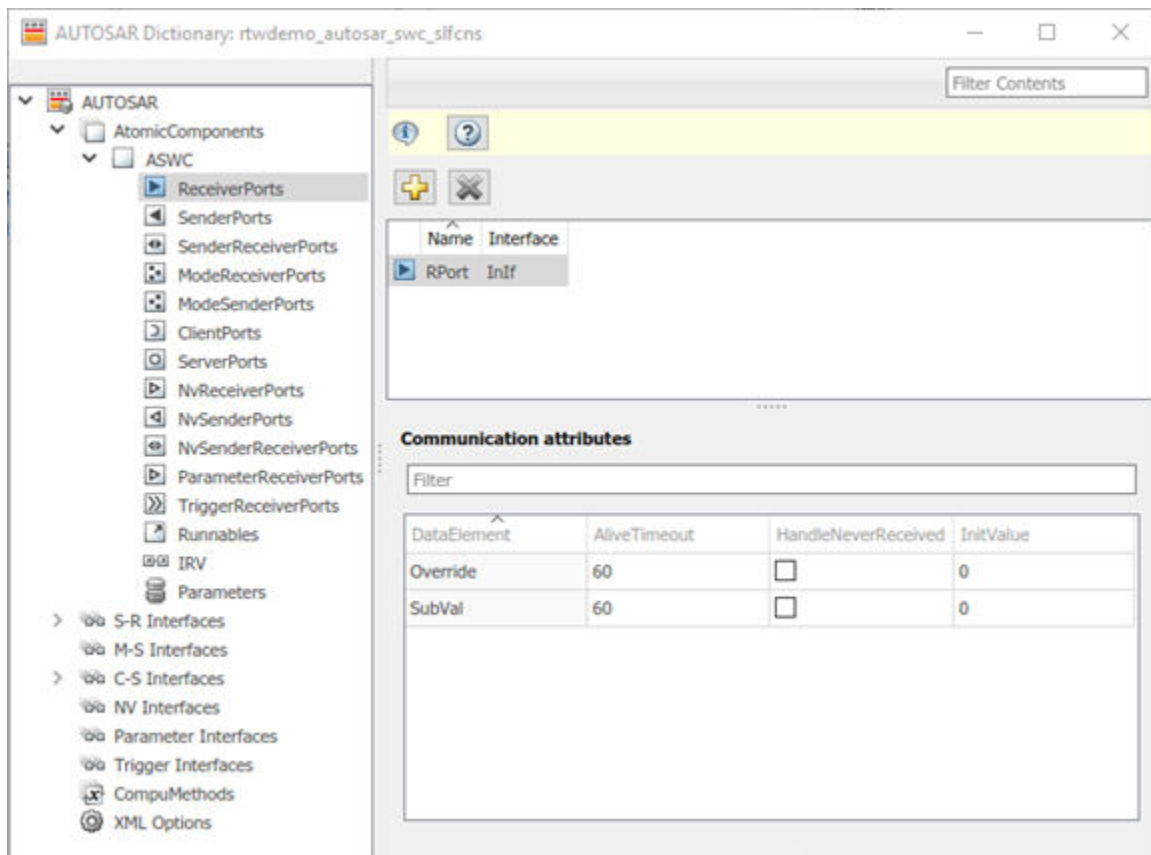
AUTOSAR Dictionary provide mapping and property views of the component model, which can be used separately and together to configure the AUTOSAR component.

The Code Mapping Editor replaces the **Simulink-AUTOSAR Mapping** view of the Configure AUTOSAR Interface dialog box. The new dialog box provides in-canvas access to AUTOSAR mapping information, with a help panel, Property Inspector dialog box, batch editing, element filtering, easy navigation to model elements and AUTOSAR properties, and model element traceability. Use this view to map model elements to AUTOSAR component elements from a Simulink model perspective.

The screenshot shows the Embedded Coder software interface. The main canvas displays a Simulink block diagram with an 'Initialize' block, a 'curValIRV' block, and a 'Runnable_1s' block containing a 'Trigger_1s' block and an 'SS1' block. The 'SS1' block has inputs for 'SubVal' and 'Override' and an output for 'DataOut'. The 'Code Mappings - AUTOSAR' window at the bottom shows a table of mappings for 'SubVal' and 'Override'.

Source	DataAccessMode	Port	Element
SubVal	ImplicitReceive	RPort	SubVal
Override	ImplicitReceive	RPort	Override

The AUTOSAR Dictionary replaces the **AUTOSAR Properties** view of the Configure AUTOSAR Interface dialog box. Using a tree format, the new dialog box displays a mapped AUTOSAR component and its elements, communication interfaces, computation methods, and XML options. Use this view to configure AUTOSAR elements from an AUTOSAR component perspective.



AUTOSAR mapping and property functions are unchanged from previous releases. They allow you to get, set, add, and remove the mapping information and component properties displayed in the Code Mapping Editor and AUTOSAR Dictionary views of the AUTOSAR component model.

For more information, see AUTOSAR Component Configuration.

AUTOSAR XML Import and Export: Round-trip ComSpecs, import bitfield CompuMethods, export interface variation points, and automate more element creation

R2018a extends arxml import and export support for AUTOSAR ComSpecs, BITFIELD_TEXTTABLE CompuMethods, and variants, and automates creation of more elements during arxml imports and model updates.

Model ComSpecs for AUTOSAR sender and receiver data

In AUTOSAR software components, a sender or receiver port optionally can specify a communication specification (ComSpec). ComSpecs describe additional communication requirements for port data.

In R2018a, to model AUTOSAR sender and receiver ComSpecs in Simulink, you can:

- Import sender and receiver ComSpecs
- Create sender and receiver ComSpecs in Simulink

- For nonqueued sender and receiver ports, modify ComSpec attribute `InitValue`
- For nonqueued receiver ports, modify ComSpec attributes `AliveTimeout` and `HandleNeverReceived`
- Export ComSpecs to `arxml` files

For more information, see [Configure AUTOSAR Sender-Receiver Port ComSpecs](#).

Note This support is available to R2017b Embedded Coder customers by installing R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.3 or later.

Import BITFIELD_TEXTTABLE CompuMethods

AUTOSAR CompuMethods of category `BITFIELD_TEXTTABLE` allow you to access bit values within an application data type of category `VALUE`. You can group bit values, assign labels to them, and define masks for accessing values within bytes of data.

In R2018a, you can import `BITFIELD_TEXTTABLE` CompuMethods from `arxml` files. After you import the CompuMethods, you can create Simulink enumerated types to represent bit groups and masks for accessing the bitfields, and reference them in Simulink bitwise and relational operator and constant blocks.

To create Simulink enumerated types for a `BITFIELD_TEXTTABLE` CompuMethod, call the AUTOSAR property function `createEnumeration`. The function creates a mask type and other enumerated types, based on what is defined in the specified CompuMethod. For example:

```
arProps = autosar.api.getAUTOSARProperties(modelName);
createEnumeration(arProps, '/Company/Module/CompuMethods/MyBitfieldCompuMethod');
```

Note This support is available to R2016a, R2016b, R2017a, and R2017b Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2016a Embedded Coder Support Package for AUTOSAR Standard, Version 16.1.8 or later
 - R2016b Embedded Coder Support Package for AUTOSAR Standard, Version 16.2.5 or later
 - R2017a Embedded Coder Support Package for AUTOSAR Standard, Version 17.1.3 or later
 - R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.3 or later
-

Export variation points for AUTOSAR interface variants

R2018a enhances AUTOSAR code generation support for variants.

- If you model an AUTOSAR port with a variant condition in Simulink, `arxml` export now generates variation points on the AUTOSAR port and data accesses.
- If you model an AUTOSAR runnable with a variant condition, `arxml` export now generates both a variation point and a corresponding variation point proxy for the runnable. The variation point and variation point proxy refer to the same AUTOSAR system constant.

Add Signal Invalidation blocks and ErrorStatus ports when required by imported AUTOSAR components

The AUTOSAR arxml importer now automatically adds Signal Invalidation blocks and ErrorStatus ports when required by an imported component that uses sender-receiver (S-R) communication. Importer function `createComponentAsModel`:

- Adds a Signal Invalidation block connected to an output if the output is mapped to an AUTOSAR sender port and the associated S-R data element uses invalidation policy KEEP or REPLACE.
- Adds an ErrorStatus port to a receiver component if the associated S-R data element meets at least one of these conditions:
 - Uses invalidation policy KEEP or REPLACE.
 - Uses an `AliveTimeout` value greater than 0.
 - Has `HandleNeverReceived` set to true.

Note This support is available to R2017b Embedded Coder customers by installing R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.1 or later.

Increased automation for AUTOSAR model updates

The arxml importer function `updateModel` now automates insertion and mapping of the following elements:

- Inter-runnable variable (IRV) lines for AUTOSAR IRVs
- Constant blocks for AUTOSAR parameters
- Data store memory (DSM) blocks for AUTOSAR per-instance memory (PIM) blocks

Previously, these elements required manual additions to the model.

Model updates also now resize added Function Caller, Constant, and DSM blocks so that block text is readable.

Note This support is available to R2016a, R2016b, R2017a, and R2017b Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2016a Embedded Coder Support Package for AUTOSAR Standard, Version 16.1.7 or later
 - R2016b Embedded Coder Support Package for AUTOSAR Standard, Version 16.2.4 or later
 - R2017a Embedded Coder Support Package for AUTOSAR Standard, Version 17.1.1 or later
 - R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.0 or later
-

Navigate AUTOSAR Update Report using search bar

The update report generated by importer function `updateModel` now provides a search bar. You can quickly navigate to specific elements or other strings of interest.

For more information, see [Import AUTOSAR Software Component Updates](#).

Note This support is available to R2016a, R2016b, R2017a, and R2017b Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2016a Embedded Coder Support Package for AUTOSAR Standard, Version 16.1.8 or later
 - R2016b Embedded Coder Support Package for AUTOSAR Standard, Version 16.2.5 or later
 - R2017a Embedded Coder Support Package for AUTOSAR Standard, Version 17.1.3 or later
 - R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.3 or later
-

Import reference definitions for AUTOSAR interface elements

The `arxml` importer function `updateModel` now imports reference definitions for AUTOSAR interface elements, such as `SenderReceiverInterface`. For a list of supported reference elements, see [Import or Update Shared AUTOSAR Reference Element Definitions](#).

AUTOSAR Signal Invalidation Block: Specify invalidation policy and initial value directly as block parameters

Embedded Coder provides the Signal Invalidation block for modeling sender-receiver data invalidation in an AUTOSAR model. R2018a enhances signal invalidation modeling for simulation and code generation. You can now:

- Specify **Signal invalidation policy** and **Initial value** for a data element directly as Signal Invalidation block parameters.
- Correctly simulate the signal invalidation policy `Replace` for an invalidated signal. Previously, simulation would keep the last valid signal value rather than replace the input data value with an initial value.
- Model the signal invalidation policy `DontInvalidate` for simulation and code generation.

For more information, see the [Signal Invalidation block reference page](#).

AUTOSAR Basic Software: Use array and bus data types with NvMServiceCaller operations

When using the Basic Software block `NvMServiceCaller` to call `ReadBlock`, `RestoreBlockDefaults`, or `WriteBlock` operations, you now can specify array and bus data types. Use the block parameter **Argument specification**.

For more information, see the [NvMServiceCaller block reference page](#).

Note This support is available to R2016b, R2017a, and R2017b Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2016b Embedded Coder Support Package for AUTOSAR Standard, Version 16.2.5 or later
 - R2017a Embedded Coder Support Package for AUTOSAR Standard, Version 17.1.3 or later
 - R2017b Embedded Coder Support Package for AUTOSAR Standard, Version 17.2.3 or later
-

Obsolete AUTOSAR functions removed

R2018a ends support for some obsolete AUTOSAR functions.

- You can no longer access the following obsolete `arxml.importer` functions. The functions have been removed from MATLAB help.

<code>getApplicationComponentNames</code>	Get AUTOSAR application software component names from <code>arxml</code> files
<code>getCalibrationComponentNames</code>	Get AUTOSAR calibration component names from <code>arxml</code> files
<code>getClientServerInterfaceNames</code>	Get AUTOSAR client-server interface names from <code>arxml</code> files
<code>getDependencies</code>	Get AUTOSAR <code>arxml</code> dependency files
<code>getFile</code>	Get AUTOSAR <code>arxml</code> software component file
<code>getSensorActuatorComponentNames</code>	Get AUTOSAR sensor/actuator software component names from <code>arxml</code> files
<code>setDependencies</code>	Set AUTOSAR <code>arxml</code> dependency files
<code>setFile</code>	Set AUTOSAR <code>arxml</code> software component file

- R2018a removes the XML option **Default aliveTimeout** from the XML options dialog box. Now you can set the **AliveTimeout** value for individual mapped AUTOSAR receiver ports, so the XML option is unnecessary. If you try to programmatically access the XML option `DefaultAliveTimeout` using AUTOSAR property function `get` or `set`, the software displays an error message.
- In R2013b, a new programmatic interface for configuring AUTOSAR properties and mapping replaced the `RTW.AutosarInterface` class. R2018a ends support for models that use the `RTW.AutosarInterface` based mapping. These models are no longer automatically converted to use the new AUTOSAR properties and mapping approach. If you attempt an operation using the old mapping approach, the software displays an error message.
- In R2013b, the `autosar_ui_launch` function replaced the `autosar_gui_launch` function, which was only briefly documented. R2018a removes `autosar_gui_launch` from the software.

Compatibility Considerations

If an AUTOSAR script or model relies on an obsolete AUTOSAR function, update it to use supported alternatives. For example:

- AUTOSAR importer scripts can use the functions listed in the `arxml.importer` reference page in place of older functions such as `getApplicationComponentNames` and `setFile`. For example:
 - Use generic function `getComponentNames` with a component type argument instead of specialized functions `getApplicationComponentNames`, `getCalibrationComponentNames`, and `getSensorActuatorComponentNames`.
 - When importing `arxml` descriptions, specify multiple `arxml` file names instead of a single component file with dependency files.
- AUTOSAR property scripts can set `AliveTimeout` values for individual mapped ports and elements, rather than setting the XML option `DefaultAliveTimeout`. For example:

```
>> set(arProps,comSpecPath{1},'AliveTimeout',90)
```

For more information, see [Configure AUTOSAR Sender-Receiver Port ComSpecs](#).

- AUTOSAR models that use pre-R2013b `RTW.AutosarInterface` based mapping should permanently migrate to using the AUTOSAR property and map functions listed in [AUTOSAR Software Components](#) and [AUTOSAR Programmatic Interfaces](#). The new functions work with the component property and mapping information displayed in [AUTOSAR Dictionary](#) and the [Code Mapping Editor](#).

To automatically convert an AUTOSAR model to use the new AUTOSAR properties and mapping approach, open the model in a MATLAB release before R2018a. The software converts the model to use the new approach.

- AUTOSAR scripts that call `autosar_gui_launch` must modify the function name to `autosar_ui_launch`. No change to function arguments is needed.

Data, Function, and File Definition

Function-Prototype Control: Configure step function name with void void interface

As of R2018a, when using function-prototype control to configure the name of a model step (execution) function, you have the option of specifying a void void interface.

For more information about using function-prototype control, see [Customize Generated C Function Interfaces](#).

Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model

R2018a simplifies configuration of data and entry-point functions for code generation, especially for larger models and models from which you generate multi-instance code. You now have the option of specifying default code generation configurations for categories of data elements and functions across a model. You can specify the default configurations interactively from a graphical user interface or programmatically with an API.

You can set a default code generation configuration for:

- Categories of model data. When producing code for the data, the code generator uses the storage class that you specify to determine properties, such as whether the data is structured, naming rules for definition and header files, and whether the data gets stored in a memory section.
- Categories of functions. When producing code for the functions, the code generator uses a function customization template that you specify to determine properties, such as a function naming rule and whether the function code gets stored in a memory section.

After applying default code generation configurations, you can override the default settings for specific data elements or functions by using the **Code** view of the Model Data Editor or Configure C/C++ Function Interface dialog box, respectively.

You can map a category of model data elements to one of the following:

- Unspecified storage class (**Default**)
- Relevant built-in storage class, such as `ExportedGlobal`
- Relevant storage class in an available package, such as `ImportFromFile`
- Storage class defined in an Embedded Coder Dictionary

New example storage classes, `SignalStruct` and `ParamStruct`, facilitate controlling generated code for signals and parameters in models that you configure for multi-instance code generation with Embedded Coder Quick Start or an Embedded Coder model template. For these storage classes, the code generator produces a `struct` type definition that encapsulates the signal or parameter data. Properties and naming rules defined for the storage classes vary depending on the category of data being mapped.

For a category of functions, you can choose from:

- Unspecified function customization template (default)
- Function customization template defined in an Embedded Coder Dictionary.

For information on how to specify code mappings, see *Configure Default Code Generation for Categories of Model Data and Functions and Code Mapping Editor*. For information about defining storage classes and function customization templates for data and function default mapping, see *Define Storage Classes, Memory Sections, and Function Templates for Software Architecture and Embedded Coder Dictionary*.

Compatibility Considerations

Starting in R2018a, to configure memory sections and shared utility function names, use the Code Mapping Editor or default mapping programming interface instead of model configuration parameters.

To Configure	Instead of Setting	Map
Memory sections	Model configuration parameters on the Memory Sections pane	Data and function categories in the Code Mapping Editor to storage classes and function customization templates that define memory sections (see <i>Configure Default Code Generation for Data and Configure Default Code Generation for Functions</i>)
Shared utility function names	Shared utilities identifier format model configuration parameter	Shared utility category on the Function Defaults tab of the Code Mapping Editor to a function customization template that defines a default function naming rule (see <i>Configure Default Code Generation for Functions</i>)

After you open the Code Perspective or use the default mapping programmatic interface to configure one or more categories of data and functions for a model, setting memory section and **Shared utilities identifier format** (formerly **Configuration Parameters > Code Generation > Symbols > Identifier format control > Shared utilities**) model configuration parameters has no effect. Also, when you open the Perspective, Simulink migrates the model configuration parameter settings to the Code Mapping Editor. If necessary, as part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses as described in *Migrate Memory Section and Shared Utility Settings from Configuration Parameters to Code Mapping Editor*.

GetSet Custom Storage Class Enhancement: Improved readability for an array of buses

In R2017b, when you used the custom storage class `GetSet` for an array of buses, the `get` function did not accept an argument. In R2018a, the `get` function accepts an integer index argument. The `get` function returns the vector value at that index. This enhancement improves the readability of the generated code and is consistent with how you apply the custom storage class `GetSet` to other signals and parameters that are vectors. For more information, see *Access Data Through Functions with Custom Storage Class GetSet*.

Compatibility Considerations

For existing models that contain array of busses and use the `GetSet` storage class, you must update the declaration and definition of the `get` function to accept an integer index argument.

Local Storage Class: Preserve local variables with Localizable storage class

In R2017b, if you created a custom storage class with the **Data scope** parameter set to `Auto`, the code generator tried to generate variables with `File` scope. If the code generator could not give a variable `File` scope, then it gave the variable `Exported` scope.

In R2018a, if you create custom storage classes that have the **Data scope** parameter set to `Auto`, for `Simulink.Signals`, the code generator first tries to generate variables that are local to a function. If generating those variables is not possible, the code generator creates variables with `File` or `Exported` scope.

For `Simulink.Signals`, there is a new `Localizable` custom storage class. You can use this custom storage class to instruct the code generator to generate variables that are local to a function.

Generating variables that are local to functions prevents the code generator from implementing optimizations that remove the variables from the generated code. The presence of local variables improves observability, readability, and is helpful in debugging the generated code. For more information, see `Generate Local Variables with Localizable Custom Storage Class`.

Accurate Header File Extension: Generate correct `#include` statements for imported data types

For custom data types (such as a `Simulink.AliasType` object), with the **Data scope** and **Header file** properties, you can configure the generated code to import the type definition from your external code. Previously, if you omitted the `.h` extension when specifying **Header file**, the code generator ignored the omission, adding the extension in `#include` statements. In R2018a, for imported types, the code generator does not add the `.h` extension.

Compatibility Considerations

If you previously omitted the `.h` extension for an imported data type, in R2018a, the generated code omits the `.h` extension in `#include` statements. If the name of the target file has a `.h` extension, you cannot compile the generated code. To generate correct code, you must add the extension in the **Header file** property.

Macro Access: Get data through a macro that your code defines

You can use the storage class `GetSet` to generate code that interacts with data by calling your custom `get` and `set` functions. In R2018a, if your external code implements the `get` mechanism for scalar or array data as a macro instead of a function, you can generate code that omits parentheses when reading that data.

Create a custom storage class by using the Custom Storage Class Designer. Set **Type** to `AccessFunction` and, on the **Access Function Attributes** tab, select **Get data through macro**

(omit parentheses). For more information, see [Access Scalar and Array Data Through Macro Instead of Function Call](#).

Tokens for Memory Sections: Use \$N token instead of identifier

Previously, when you defined a memory section by using the Custom Storage Class Designer, you used the placeholder %<identifier> to stand for the name of each relevant function or variable. In R2018a, you use \$N instead of %<identifier>.

You do not need to manually modify your existing memory sections so that they use \$N. As you create new memory sections, use \$N instead of %<identifier>.

Compatibility Considerations

If you open your existing package in the Custom Storage Class Designer and click **Save**, the Designer permanently replaces %<identifier> with \$N in your memory sections. To use the modified memory sections, other users of the package must have R2018a or a later release.

Parameter Initialization: Statically initialize tunable parameters from system constants and other macros

By default, the generated code statically initializes tunable parameters by using literal numbers. Tunable parameters include global variables and structure fields that represent block parameters, such as the **Gain** parameter of a Gain block. For example, suppose you apply the storage class `ExportedGlobal` to a `Simulink.Parameter` object named `myParam` whose value is 15. The generated static initialization code looks like this code:

```
real_T myParam = 15;
```

In R2018a, you can generate code that initializes `myParam` by using an expression that involves macros. For example, you can generate code that looks like this code:

```
#define SYSCONST 5
real_T myParam = 3 * SYSCONST;
```

The layer of abstraction that the expression provides can make the generated code easier to read and maintain.

To generate such code, you create two parameter objects: `myParam` and `SYSCONST`. By setting the value (`Value` property) of `myParam` to an expression, you explicitly model the relationship between the parameter objects. By applying a macro storage class such as `Define` to `SYSCONST`, you generate code that adheres to C syntax rules, which prohibit static initialization from data that reside in memory.

For more information about setting the value of a parameter object to an expression that involves other variables and objects, see [Parameter Dependencies: Explicitly model relationships between dependent and independent variables](#).

Model-Scoped Parameter Objects: Use FileScope to prevent name clashes between parameters in different models

Previously, in a hierarchy of referenced models, if you stored two parameter objects (such as `Simulink.Parameter`) with the same name in different model workspaces, you could not apply a

storage class other than `Auto` to the objects. If both objects used a storage class other than `Auto`, the code generator produced an error due to name clashing.

In R2018a, to prevent the names from clashing, you can apply the storage class `FileScope` to the objects. In the generated code, each object appears as a `static` global variable in different code files.

For more information, see [Prevent Name Clashes by Configuring Data Item as `static`](#).

File Packaging of Generated Code for Global Simulink Function Blocks: Code for function body placed in `model.c`

As of R2018a, the code generator places code for the body of a global Simulink Function block in the `model.c` file.

Compatibility Considerations

Prior to R2018a, by default, the code generator placed code for the body of a global Simulink Function block in a function-specific file, `function.c`, separate from code for the root model (`model.c`). To support models that include Rate Transition blocks and global Simulink Function blocks, the code generator now includes the algorithm code for a global Simulink Function block as a rate-grouped function in `model.c`. You can no longer control the file packaging of rate-grouped output functions.

This change does not impact the file packaging of the function declaration for global Simulink Function blocks. The code generator still places the declaration in a function-specific header file, `function.h`.

For more information about controlling the file packaging of generated code, see [Manage File Packaging of Generated Code Modules](#).

Identifiers: Represent name of storage class in identifier naming rules by using new token `$G`

Include the name of a storage class associated with a data item (signal, block parameters, or state) in generated code as a global variable or global type by using the new naming rule token `$G`. Apply the token to global variables or global type by including `$G` in the naming rule that you specify for the model configuration parameter **Global variables** or **Global types**.

In the Embedded Coder Dictionary, use the `$G` token for including the name of a storage class in code generation naming rules associated with a category of model data elements. For example, you can use `$G` token in the naming rule that you specify for the **Header File** defined for a storage class that you create in the Embedded Coder Dictionary. When you define a naming rule for a storage class for structured code, you can use `$G` as one of the tokens when naming the structure type and instance. For more information, see [Define Storage Classes, Memory Sections, and Function Templates for Software Architecture](#).

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
<p>Memory Sections pane in configuration parameters</p>	<p>The pane has been removed. The parameters have been moved to Code Generation > Advanced parameters.</p>	<p>Before you open the new Code Perspective for the first time in the model, you can continue to use these configuration parameters.</p> <p>After you open the Code Perspective, use the Code Mapping Editor to configure memory sections. See “Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model” on page 10-14.</p>	<p>After you open the Code Perspective, programmatically accessing these parameters (for example, with a script) generates warnings. Adjust scripts so they use the programmatic interface of the Code Mapping Editor instead (see “Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model” on page 10-14).</p>
<p>Configuration Parameters > Code Generation > Symbols > Identifier format control > Shared utilities</p>	<p>The parameter has been moved and renamed to Configuration Parameters > Code Generation > Symbols > Advanced parameters > Shared utilities identifier format.</p>	<p>Before you open the new Code Perspective for the first time in the model, you can continue to use this configuration parameter.</p> <p>After you open the Code Perspective, use the Code Mapping Editor to configure naming rules for shared utility functions. See “Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model” on page 10-14.</p>	<p>After you open the Code Perspective, programmatically accessing this parameter (for example, with a script) generates a warning. Adjust scripts so they use the programmatic interface of the Code Mapping Editor instead (see “Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model” on page 10-14).</p>

Code Generation

Code Perspective: Customize Simulink desktop for code generation workflows

In the Simulink Editor, the Code Perspective provides the tools to prepare your model for code generation.

You can:

- Apply default code generation settings to categories of model data elements and entry-point functions.
- Override these default settings for individual elements and functions by using existing tools, such as the Model Data Editor or function prototype control.
- Create custom definitions, such as storage classes, that you can apply to categories of data and functions across a model.
- Set model configuration parameters related to code generation. From the help pane, open the Configuration Parameters dialog box.
- From the ellipsis menu, easily trace from selected model elements to generated code.
- Detect model design elements that do not meet code generation requirements through edit-time checking (requires Simulink Check).

The integrated help pane provides quick access to tools, video tutorials, and links to more information.

To open the Code Perspective, select **Code > C/C++ > Configure Model in Code Perspective** or, in the Simulink Editor, click the perspective control in the lower-right corner and select **Code**. For more information about using the Code Perspective, see Environment for Configuring Model Data and Functions for Code Generation.

After you open the Code Perspective or use the default mapping programmatic interface to configure one or more categories of data and functions for a model, setting memory section and **Shared utilities identifier format** (formerly **Configuration Parameters > Code Generation > Symbols > Identifier format control > Shared utilities**) model configuration parameters has no effect. Also, when you open the Perspective, Simulink migrates the model configuration parameter settings to the Code Mapping Editor. If necessary, as part of the migration, Simulink configures the Embedded Coder Dictionary that the model uses as described in Migrate Memory Section and Shared Utility Settings from Configuration Parameters to Code Mapping Editor.

Rate Transition Block Code Customization: Separate Rate Transition block code and data from algorithm code and data

Previously, for Rate Transition blocks, the code was inlined with model code and the variable declarations were in a global state structure that was applicable to all model blocks.

In R2018a, you can use a new model configuration parameter, **Rate Transition block code**, to separate the Rate Transition block code and data from the model code and data. The generated code contains separate `get` and `set` functions that the `model_step` functions call and a dedicated structure for state data. The generated code also contains separate start and initialize functions that the `model_initialize` function calls.

Separating Rate Transition block code and data from algorithm code and data enables you to independently analyze, optimize, and test Rate Transition block and algorithm code.

Generated Files: Customize generated file names with new token \$E

You can now customize the names of the generated files. When you use Modular or Compact (with separate data file) file packaging, you can specify custom names for generated header, source, and data files. When you use Compact file packaging, you can specify custom names for generated header and source files.

On the **Code Generation > Code Placement** pane, enter custom names in **Header files**, **Source files**, and **Data files** fields.

\$E is a new token representing the type of data interface. The custom names are also applicable to the additional files generated for a data interface. \$E represents these instances of data interface file types:

- capi
- capi_host
- dt
- testinterface
- private
- types

Custom naming is supported only for .c, .h, .cpp, and .hpp files. When you have model hierarchy, custom naming is applicable to only the root model.

\$E is mandatory for **Header files** and **Source files**. It is not supported for **Data files**. Other supported tokens for **Header files** and **Source files** are \$R, \$U, or any custom user text. One of the supported tokens is mandatory for the **Header files** and **Source files**. For more information, see [Customize Generated File Names](#).

These TLC functions are added to support customization of generated file names:

- LibGetMdlDataSrcBaseName()
- LibGetMdlTypesHdrBaseName()
- LibGetMdlCapiHdrBaseName()
- LibGetMdlCapiSrcBaseName()
- LibGetMdlCapiHostHdrBaseName()
- LibGetMdlTestIfHdrBaseName()
- LibGetMdlTestIfSrcBaseName()
- LibGetDataTypeTransHdrBaseName()

These TLC functions are updated to support customization of generated file names:

- LibGetMdlSrcBaseName()
- LibGetMdlPubHdrBaseName()
- LibGetMdlPrvHdrBaseName()

Hardware Implementation Settings: Inaccurate values corrected

R2018a provides the correct values for these **Hardware Implementation** pane settings.

Device vendor	Device type	Device detail	R2018a value	Previous value
Texas Instruments	C5000	Number of bits per pointer	16	32
Texas Instruments	C5000	Number of bits per ptrdiff_t	16	32
Texas Instruments	TMS570 Cortex-R4	Byte ordering	Big Endian	Little Endian

When you open a saved model from a previous release, R2018a updates the incorrect values.

For more information, see [Hardware Implementation Pane](#).

Cross-Release Code Integration: Reuse referenced model code with instance-specific parameters

Reuse previously generated code from a referenced model multiple times within a current release model. For each instance of the SIL or PIL block that contains the referenced model code, you can specify unique values for the model arguments.

For more information, see [Use Multiple Instances of Code Generated from Reusable Referenced Model](#)

Cross-Release Code Integration: Import and simulate AUTOSAR code

Into the current release, import AUTOSAR component code that you generated in a previous release. Run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations with the imported code. You can observe the interaction of code from a previous release with components implemented in the current release. For more information, see:

- [Workflow](#)
- [Import AUTOSAR Code from Previous Releases](#)

Traceability Comments: Specify Simulink identifier in comments for Simulink blocks, Stateflow objects, and MATLAB Function blocks

You can now specify the Simulink Identifier (SID) in the comments generated for Simulink, Stateflow objects, and MATLAB Function blocks.

In the Configuration Parameters dialog box, on the **Code Generation > Comments** pane, the parameters **Simulink block comments** or **Stateflow object comments** enable an additional setting, **Trace to model using**, so that you can choose between `Simulink identifier` or `Block path`. `Block path` is the default option.

When you select `Simulink identifier`, the generated comment includes the Simulink identifier without the model name for the corresponding block or object. For example, if a block is named `Inport1` and its SID is `model_name:1`, the block identifier in the generated comment is `Inport:`

'Inport1' (':1'). When you select **Block path**, the generated comment includes the entire block path from the root. Including the entire block path results in the previously existing format of comment generation. For example, the block path for **Inport1** is '<Root>/Inport1'.

You can now pass the SID format (*model_name:number*) as an argument to:

- `rtwtrace` for tracing to generated code. For example:

```
rtwtrace('rtwdemo_comments:1')
```
- `hilite_system` for tracing elements in the model. For example:

```
hilite_system('rtwdemo_comments:1')
```

When you enable traceability through **Simulink identifier**, you obtain consistent comment generation despite changes in the model such as subsystem addition or deletion.

Newline Style: Customize linefeed character irrespective of the operating system

In the generated code, the newline character differs according to the operating system that the code is generated on. You can now customize the newline character irrespective of the operating system. On the **Code Generation > Code Style > Advanced Parameters** pane, use the configuration parameter **Newline style** to specify the newline character as **Default**, **LF (Line Feed)**, or **CRLF (Carriage Return + Line Feed)**.

The **Default** option generates the newline character based on the operating system that the code is generated on. You can select **LF (Line Feed)** and **CRLF (Carriage Return + Line Feed)** options irrespective of the operating system. This customization enables portability of the generated code to different operating systems for compilation. For more details, see **Control Newline Style in Generated Code**.

Export Functions: Generate ScratchModel file containing a Model block

When you select a subsystem and select **Code > C/C++ Code > Export Functions**, the operation creates a new model, *subsystem.slx*, that contains the content of the original subsystem and creates a **ScratchModel** that contains a **Model** block. This block references the newly created *subsystem.slx* model.

If the original model has the **Configuration Parameters > Code Generation > Verification > Advanced parameters > Create block** parameter set to **SIL** or **PIL**, the software creates a **Model** block with **Simulation mode** set to **Software-in-the-loop (SIL)** or **Processor-in-the-loop (PIL)**.

For more information, see **Generate Code for Export-Function Subsystems and Create block**.

Deployment

Build Process: Specify toolchain for template makefile

To build code generated from Simulink models, you can specify a process that uses a template makefile that is associated with a toolchain.

You can still use the template makefile approach that you used with previous releases, that is, you can use a template makefile build process that is not associated with a toolchain.

For more information, see [Choose Build Approach and Configure Build Process](#).

Build Process Status for Parallel Builds: View and interact with build process status for parallel builds of referenced model hierarchies

You can now view and interact with build process status for parallel builds through the **Build Process Status** window. In the window, you see the status of referenced model builds, the elapsed time for builds, and a **Cancel** button that you can use to end the build process without creating incomplete build artifacts. For more information, see [View Build Process Status](#).

TI C2000 IPC Block: Support for Inter-Processor Communications for F2837xD in TI C2000 Support Package

Inter-Processor Communications (IPC) Receive and Transmit blocks are supported for F2837xD processors.

C2000 F28004x: Support for peripherals in Texas Instruments C2000 Support Package

eCAP, eQEP, SPI, I2C, and CLA peripherals are supported for code generation in C2000 F28004x processors.

STM32F7 Audio: Multiple channel Mic-In, Line-In, and Speaker out for STM32F769I-Discovery in STM32 Support Package

Multiple channel Mic-In, Line-In, and Speaker out blocks are supported for STM32F769I-Discovery boards.

STM32F7 External Mode: Support for TCP/IP and Serial Communication for STM32F769I-Discovery board in STM32 Support Package

External Mode over TCP/IP (static and DHCP) and External Mode over Serial are supported for the STM32F769I-Discovery board. External mode performance over TCP/IP has been improved through the use of Universal Measurement and Calibration Protocol (XCP).

External Mode Simulation: Upload execution-time metrics through XCP transport layer

For XCP-based external mode simulations, you can:

- Configure execution-time profiling for the target code.
- Stream execution-time metrics to the Simulation Data Inspector.

You can use this feature in external mode simulations that run target applications on:

- Your development computer
- Xilinx® Zynq® ZC7000 development kits:
 - ZedBoard
 - ZC702 Evaluation Kit
 - ZC706 Evaluation Kit
- Intel SoC FPGA evaluation boards:
 - Cyclone® V SoC Development Kit
 - Arrow® SoCkit
 - Arria® 10 SoC Development Kit
- ARM Cortex-A9 processors
- STMicroelectronics Discovery boards:
 - STM32F746G
 - STM32F769I

For more information, see [External Mode Simulation: Use XCP communication protocol \(Simulink Coder\)](#).

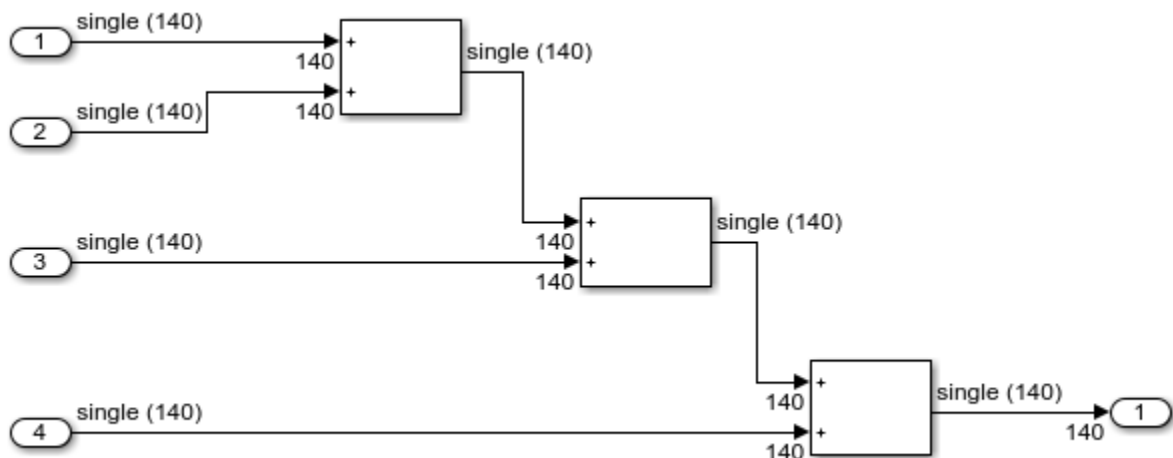
Performance

Single Instruction, Multiple Data (SIMD) Intrinsic: Generate code with optimized load and store operations for use with Intel processors with SSE/AVX support

In R2017b, for Intel processors with SSE support, you generated code with functions that utilized SIMD instructions by choosing an Intel IPP/SSE code replacement library. The generated code processed multiple data inputs in a single instruction.

In R2018a, for element-wise arithmetic operations involving `single` and `double` data types, you can generate more efficient code containing SIMD intrinsics. There are less data copies and no wrapper functions for the SIMD intrinsics. To generate this code, in the Configuration Parameters dialog box, for the **Code replacement library** parameter, choose the Intel SSE (Windows) or Intel SSE (Linux) library.

For example, the model, `simd_model`, contains three simple addition operations.



In R2017b, the `simd_model_step` function contained this code:

```

void simd_model_step(void)
{
    __attribute__((aligned(16))) real32_T rtb_Add[140];
    __attribute__((aligned(16))) real32_T rtb_Add1[140];
    mw_gcc_sse_mm_add_f32x4(simd_model_U.In1, 140, 1, simd_model_U.In2, rtb_Add);
    mw_gcc_sse_mm_add_f32x4(rtb_Add, 140, 1, simd_model_U.In3, rtb_Add1);
    mw_gcc_sse_mm_add_f32x4(rtb_Add1, 140, 1, simd_model_U.In4, simd_model_Y.Out1);
}

```

The generated code contained two temporary buffers `Add` and `Add1` that held data that was passed to the `mw_gcc_sse_mm_add_f32x4` function. The `mw_sse.c` file contained the `mw_gcc_sse_mm_add_f32x4` function definition. This function contained the SIMD intrinsics `_mm_load_ps`, `_mm_add_ps`, and `_mm_store_ps`.

```

void mw_gcc_sse_mm_add_f32x4(const float * A, int Row, int Col, const float * B, float * C)
{

```

```

__m128 sse_a, sse_b, sse_c;
int size = Row*Col;
int i;
int k=0;

for (i = 0; i < size ; i+=4)
{
    sse_a = _mm_load_ps(A+i);
    sse_b = _mm_load_ps(B+i);
    sse_c = _mm_add_ps(sse_a, sse_b);
    _mm_store_ps(C+i, sse_c);
}

k=i-4;
for (i = 0; i < size%4 ; i++)
{
    C[k+i] = A[k+i]+B[k+i];
}
}

```

In R2018a, the `simd_model.c` file contains this code:

```

void simd_model_step(void)
{
    int32_T idx;
    __m128 tmp;
    __m128 tmp_0;
    for (idx = 0; idx <= 136; idx += 4) {
        tmp = _mm_load_ps(&simd_model_U.In1[idx]);
        tmp_0 = _mm_load_ps(&simd_model_U.In2[idx]);
        tmp = _mm_add_ps(tmp, tmp_0);
        tmp_0 = _mm_load_ps(&simd_model_U.In3[idx]);
        tmp_0 = _mm_add_ps(tmp, tmp_0);
        tmp = _mm_load_ps(&simd_model_U.In4[idx]);
        tmp = _mm_add_ps(tmp_0, tmp);
        _mm_store_ps(&simd_model_Y.Out1[idx], tmp);
    }
}

```

In R2018a, the generated code does not contain the buffers `Add` and `Add1`. There is no function wrapper for the SIMD intrinsics. For Intel processors with AVX/AVX2 support, you can choose the Intel AVX (Windows) or Intel AVX (Linux) libraries. Using these libraries, you can generate code that processes even more data in a single instruction. For more information on Code Replacement Libraries, see [What Is Code Replacement?](#)

Preprocessor Conditionals: Obtain better readability of generated code for variant systems

You can generate code from Simulink models containing one or more variant choices. The generated code contains preprocessor conditionals that control the activation of each variant choice.

In R2017b, the generated code for variant systems and variant subsystems had nested or consecutive preprocessor conditionals with the same condition, resulting in redundant `#if` conditions.

For example, the generated code for a Switch block has nested `#if` conditions. The variable `rtb_Merge` is assigned to `rtb_VariantMerge_For_Variant_So` in another redundant `#if` condition.

```

real_T rtb_Merge;
real_T rtb_VariantMerge_For_Variant_So;
#if isfoo

```

```

switch (mMergeLocalize_U.In1) {
  case 0:
    #if isfoo
      rtb_Merge = mMergeLocalize_P.Constant_Value;
    #endif
    break;

  case 1:
    #if isfoo
      rtb_Merge = mMergeLocalize_P.Constant1_Value;
    #endif
    break;

  default:
    #if isfoo
      rtb_Merge = mMergeLocalize_P.Constant2_Value;
    #endif
    break;
}
#endif
#if isfoo
  rtb_VariantMerge_For_Variant_So = rtb_Merge;
#endif

```

In R2018a, the nested and redundant `#if` conditions are removed. Because the variable `rtb_Merge` is not necessary, it has been removed. The generated code is optimized for better readability and code efficiency.

```

real_T rtb_VariantMerge_For_Variant_So;
#if isfoo
  switch (mMergeLocalize_U.In1) {
    case 0:
      rtb_VariantMerge_For_Variant_So = mMergeLocalize_P.Constant_Value;
      break;

    case 1:
      rtb_VariantMerge_For_Variant_So = mMergeLocalize_P.Constant1_Value;
      break;

    default:
      rtb_VariantMerge_For_Variant_So = mMergeLocalize_P.Constant2_Value;
      break;
  }
#endif

```

Some other instances of code efficiency are:

- Expression folding
- Fusion of consecutive preprocessor conditional regions with identical conditions

Not all instances of redundant preprocessor conditionals can be optimized for better readability and code efficiency.

Buffer Reuse: Prioritize buffer reuse based on signal labels in model diagram

In R2017b, you used the same `Reusable` custom storage class specification on different signal lines to specify which buffers to reuse. You can now specify which buffers to reuse without the `Reusable` custom storage class specification by labeling different signals with the same name and by selecting the **Use signal labels to guide buffer reuse** configuration parameter. If possible, the code generator reuses these buffers in the generated code.

After you study the generated code, the Static Code Metrics Report, and identify areas where you think buffer reuse is possible, use signal labels to remove additional data copies. Specifying buffer

reuse reduces RAM consumption and improves execution speed. For more details and an example, see [Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse](#).

Configuration Set: New location and layout for optimization model configuration parameters

In R2018a, there are these changes to the optimization parameters:

- New parameters that make it easier for you to optimize the generated code to meet your specific objectives.

Parameter	Pane
Level	To optimize the generated code, choose from these levels: <ul style="list-style-type: none"> • Minimum (debugging) • Balanced with Readability • Maximum (default setting)
Priority	If you set the Level parameter to Maximum , choose one of these priorities: <ul style="list-style-type: none"> • Balance RAM and speed (default setting) • Maximum execution speed • Minimize RAM
Specify custom optimizations	Select individual parameter settings. Selecting this parameter deactivates the Level and Priority parameters.

On the **Optimization** pane, in the **Details** section, you can view the optimization parameter settings that correspond with the **Level** and **Priority** parameters.

Existing models that you load in R2018a have the **Specify custom optimizations** parameter selected. Models that you create in R2018a have a **Level** parameter setting of **Maximum** and a **Priority** parameter setting of **Balance RAM and speed**.

If you plan on upgrading your software, be aware that:

- Setting the **Priority** and **Level** parameters enables the latest optimizations corresponding with these settings for each subsequent release.
- Selecting **Specify custom optimizations** means that when you load a model in a future release, any optimization parameters that were introduced in releases after you adopted to when you upgrade are set to **off**. If you want to reduce the amount of changes in the generated code when you upgrade your software, this option may be a good choice.
- Previously, there were three optimization panes. There is now a single **Optimization** pane, which is under **Code Generation**.
- For the parameters in this table, there are new default settings. In R2018a, these default settings correspond to a **Level** parameter setting of **Maximum** and a **Priority** parameter setting of **Balance RAM and speed**.

Parameter	R2017b Default Setting	R2018a Default Setting
Optimize global data access	None	Use global to hold temporary results
Optimize block operation order in the generated code	Off	Improved Code Execution Speed
Reuse buffers of different sizes and dimensions	Off	On

- Some parameters that were on the **Optimization** pane are now on either the **Math and Data Types** pane or the **Simulation Target** pane. This table lists these parameters and their locations.

Parameter	R2018a Pane
Default for underspecified data type	Math and Data Types
Use division for fixed-point net slope computation	Math and Data Types
Use floating-point multiplication to handle net slope corrections	Math and Data Types
Application lifespan	Math and Data Types
Implement logic signals as Boolean data (vs. double)	Math and Data Types
Evaluated application lifespan	Math and Data Types
Block reduction	Simulation Target
Compiler optimization level	Simulation Target
Conditional input branch execution	Simulation Target
Signal storage reuse	Simulation Target
Verbose accelerator builds	Simulation Target

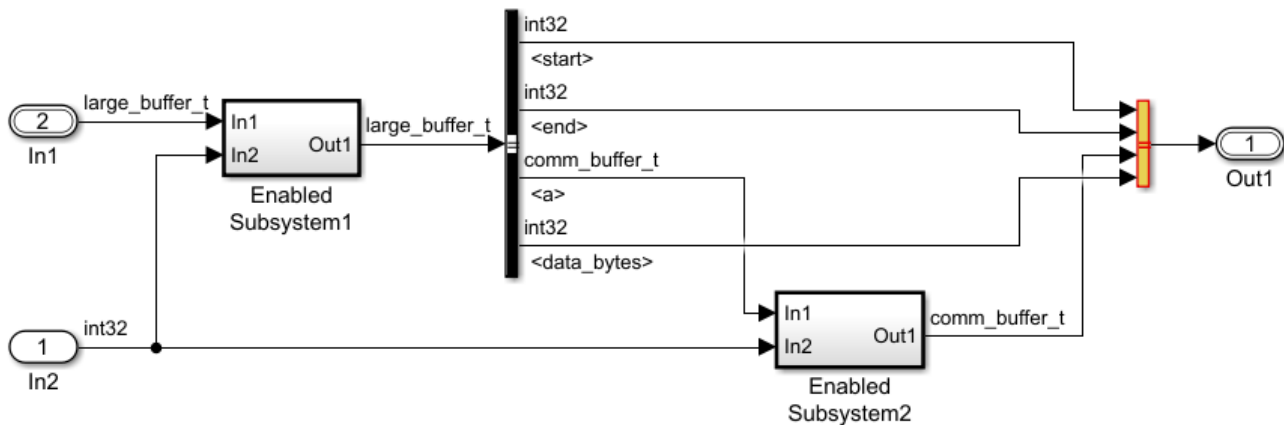
For more information, see Performance.

Data Copy Reduction: Generate code with fewer data copies for writes to structure fields and matrix elements and for control flow patterns

In R2018a, the generated code contains fewer data copies for writes to structure fields or matrix elements and for complex control flow patterns. These optimizations reduce RAM consumption and improve execution speed.

Data copy reduction for structure fields

In the model `ex_bus_fold`, the `In1` output and the `Out1` input are bus signals containing four elements. The Enabled Subsystems contain Bus Assignment blocks.



In R2017b, the `ex_bus_fold_step` function contained this code:

```
void ex_bus_fold_step(void)
{
    int32_T rtb_MultiportSwitch_start;
    int32_T rtb_MultiportSwitch_end;
    int32_T rtb_MultiportSwitch_data_bytes;
    comm_buffer_t rtb_MultiportSwitch_a;
    const comm_buffer_t *rtb_MultiportSwitch_a_0;
    if (ex_bus_fold_U.In2 == 1) {
        rtb_MultiportSwitch_start = ex_bus_fold_U.In2;
        rtb_MultiportSwitch_end = ex_bus_fold_U.In1.end;
        rtb_MultiportSwitch_a_0 = &ex_bus_fold_U.In1.a;
        rtb_MultiportSwitch_data_bytes = ex_bus_fold_U.In1.data_bytes;
    } else {
        rtb_MultiportSwitch_start = ex_bus_fold_U.In1.start;
        rtb_MultiportSwitch_end = ex_bus_fold_U.In1.end;
        rtb_MultiportSwitch_a_0 = &ex_bus_fold_U.In1.a;
        rtb_MultiportSwitch_data_bytes = ex_bus_fold_U.In1.data_bytes;
    }

    ex_bus_fold_EnabledSubsystem2(rtb_MultiportSwitch_a_0, ex_bus_fold_U.In2,
        &rtb_MultiportSwitch_a);
    ex_bus_fold_Y.Out1.start = rtb_MultiportSwitch_start;
    ex_bus_fold_Y.Out1.end = rtb_MultiportSwitch_end;
    ex_bus_fold_Y.Out1.a = rtb_MultiportSwitch_a;
    ex_bus_fold_Y.Out1.data_bytes = rtb_MultiportSwitch_data_bytes;
}
```

The variables `rtb_MultiportSwitch_start`, `rtb_MultiportSwitch_end`, and `rtb_MultiportSwitch_data_bytes`, and the structure `rtb_MultiportSwitch_a` hold temporary copies of bus data. At the end of the step function, the code contains data copies from these variables back to the fields of the `ex_bus_fold_Y.Out1` structure.

In R2018a, the `ex_bus_fold_step` function contains this code:

```
void ex_bus_fold_step(void)
{
    const comm_buffer_t *rtb_MultiportSwitch_a;
    if (ex_bus_fold_U.In2 == 1) {
        ex_bus_fold_Y.Out1.start = ex_bus_fold_U.In2;
```

```

    ex_bus_fold_Y.Out1.end = ex_bus_fold_U.In1.end;
    rtb_MultiportSwitch_a = &ex_bus_fold_U.In1.a;
    ex_bus_fold_Y.Out1.data_bytes = ex_bus_fold_U.In1.data_bytes;
} else {
    ex_bus_fold_Y.Out1.start = ex_bus_fold_U.In1.start;
    ex_bus_fold_Y.Out1.end = ex_bus_fold_U.In1.end;
    rtb_MultiportSwitch_a = &ex_bus_fold_U.In1.a;
    ex_bus_fold_Y.Out1.data_bytes = ex_bus_fold_U.In1.data_bytes;
}

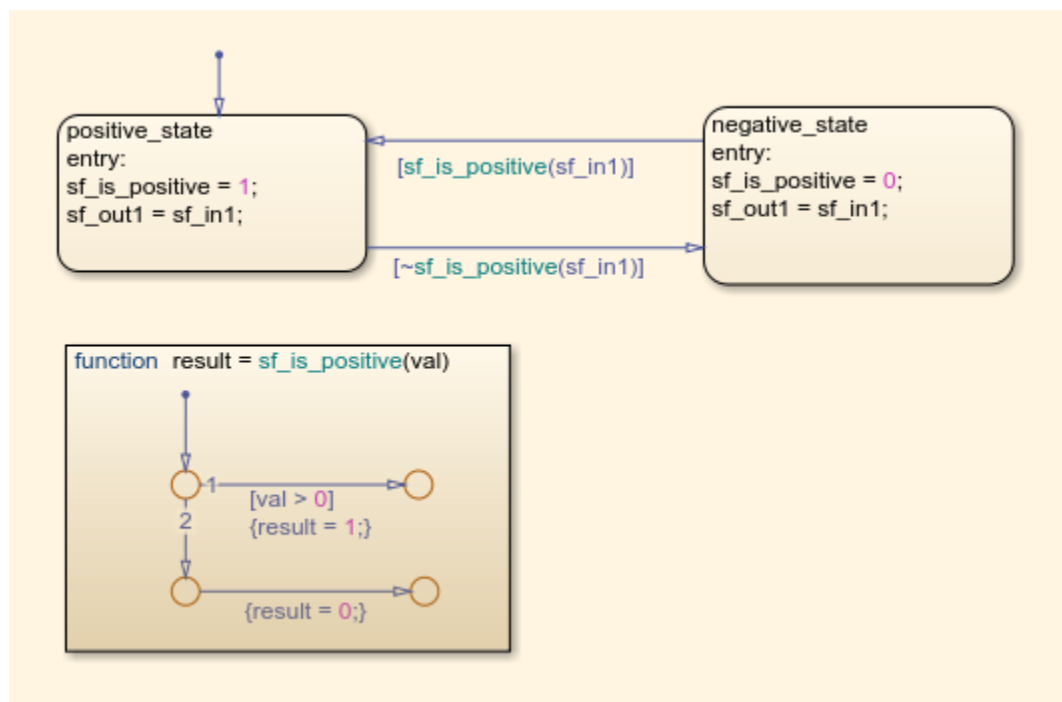
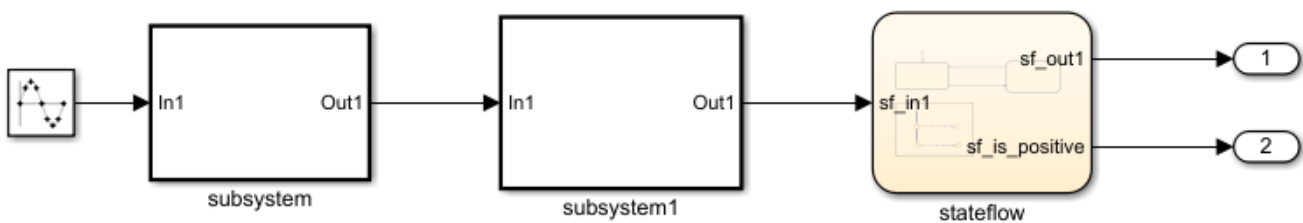
ex_bus_fold_EnabledSubsystem2(rtb_MultiportSwitch_a, ex_bus_fold_U.In2,
    &ex_bus_fold_Y.Out1.a);
}

```

In R2018a, these variables and their corresponding data copies are not in the generated code.

Data copy reduction for complex control flow modeling patterns

The model `ex_control_flow` contains a Stateflow chart with a complex control flow pattern.



In R2017b, the `ex_control_flow_step` function contained this code:


```

void ex_control_flow_step(void)
{
    int32_T rtb_sf_is_positive;
    real_T rtb_Gain;
    rtb_Gain = ex_control_flow_subsystem(sin((real_T)ex_control_flow_DW.counter *
        2.0 * 3.1415926535897931 / 10.0) * 100.0);
    rtb_Gain = ex_control_flow_subsystem1(rtb_Gain);
    if (ex_control_flow_DW.is_active_cl_ex_control_flow == 0U) {
        ex_control_flow_DW.is_active_cl_ex_control_flow = 1U;
        ex_control_flow_DW.is_cl_ex_control_flow = ex_control_fl_IN_positive_state;
        rtb_sf_is_positive = 1;
        ex_control_flow_Y.Out1 = rtb_Gain;
    } else if (ex_control_flow_DW.is_cl_ex_control_flow ==
        ex_control_fl_IN_negative_state) {
        rtb_sf_is_positive = 0;
        if (ex_control_flow_sf_is_positive(rtb_Gain) != 0.0) {
            ex_control_flow_DW.is_cl_ex_control_flow = ex_control_fl_IN_positive_state;
            rtb_sf_is_positive = 1;
            ex_control_flow_Y.Out1 = rtb_Gain;
        }
    } else {
        rtb_sf_is_positive = 1;
        if (!(ex_control_flow_sf_is_positive(rtb_Gain) != 0.0)) {
            ex_control_flow_DW.is_cl_ex_control_flow = ex_control_fl_IN_negative_state;
            rtb_sf_is_positive = 0;
            ex_control_flow_Y.Out1 = rtb_Gain;
        }
    }
}

ex_control_flow_Y.Out2 = rtb_sf_is_positive;
ex_control_flow_DW.counter++;
if (ex_control_flow_DW.counter == 10) {
    ex_control_flow_DW.counter = 0;
}
}

```

The generated code contained the local variable `rtb_sf_is_positive` for holding a temporary value. In the control flow region, the code contained a write to this variable. At the end of the function, the code contained a copy from this variable to the `ex_control_flow_Y.Out2` variable.

In R2018a, the `ex_control_flow_step` function contains this code:

```

void ex_control_flow_step(void)
{
    real_T rtb_Gain;
    rtb_Gain = ex_control_flow_subsystem(sin((real_T)ex_control_flow_DW.counter *
        2.0 * 3.1415926535897931 / 10.0) * 100.0);
    rtb_Gain = ex_control_flow_subsystem1(rtb_Gain);
    if (ex_control_flow_DW.is_active_cl_ex_control_flow == 0U) {
        ex_control_flow_DW.is_active_cl_ex_control_flow = 1U;
        ex_control_flow_DW.is_cl_ex_control_flow = ex_control_fl_IN_positive_state;
        ex_control_flow_Y.Out2 = 1.0;
        ex_control_flow_Y.Out1 = rtb_Gain;
    } else if (ex_control_flow_DW.is_cl_ex_control_flow ==
        ex_control_fl_IN_negative_state) {
        ex_control_flow_Y.Out2 = 0.0;
        if (ex_control_flow_sf_is_positive(rtb_Gain) != 0.0) {
            ex_control_flow_DW.is_cl_ex_control_flow = ex_control_fl_IN_positive_state;

```

```

    ex_control_flow_Y.Out2 = 1.0;
    ex_control_flow_Y.Out1 = rtb_Gain;
}
} else {
    ex_control_flow_Y.Out2 = 1.0;
    if (!(ex_control_flow_sf_is_positive(rtb_Gain) != 0.0)) {
        ex_control_flow_DW.is_c1_ex_control_flow = ex_control_fl_IN_negative_state;
        ex_control_flow_Y.Out2 = 0.0;
        ex_control_flow_Y.Out1 = rtb_Gain;
    }
}

ex_control_flow_DW.counter++;
if (ex_control_flow_DW.counter == 10) {
    ex_control_flow_DW.counter = 0;
}
}

```

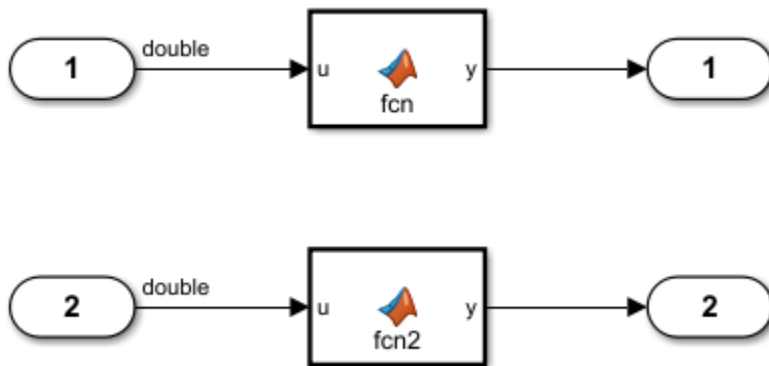
In R2018a, the variable `rtb_sf_is_positive` and its corresponding data copy are not in the generated code. In the control flow code, the writes occur directly to `ex_control_flow_Y.Out2`.

Code Size Reduction: Eliminate identical functions in the generated code

In R2017b, for a model that contained two or more MATLAB Function blocks or Stateflow Charts that called the same function, the generated code contained identical function definitions.

In R2018a, for these identical functions, the code generator creates one function that the MATLAB Function block or Stateflow Chart code can call. Creating one function reduces ROM consumption by eliminating redundant code.

For example, the model `ex_function_elim` contains two identical MATLAB Function blocks. The code for these blocks contains two calls to the external function `sfn_normVector`.



In R2017b, the code generator produced this code:

```

static real_T sfn_normVector(const real_T v[3]);
static real_T sfn_normVector_g(const real_T v[3]);
static real_T sfn_normVector(const real_T v[3])
{

```

```

    return sqrt((v[0] * v[0] + v[1] * v[1]) + v[2] * v[2]);
}

static real_T sfn_normVector_g(const real_T v[3])
{
    return sqrt((v[0] * v[0] + v[1] * v[1]) + v[2] * v[2]);
}

void ex_function_elim_step(void)
{
    real_T tmp[3];
    real_T tmp_0[3];
    tmp[0] = rtU.In1;
    tmp_0[0] = 4.0 * rtU.In1;
    tmp[1] = 2.0 * rtU.In1;
    tmp_0[1] = 5.0 * rtU.In1;
    tmp[2] = 3.0 * rtU.In1;
    tmp_0[2] = 6.0 * rtU.In1;
    rtY.Out1 = sfn_normVector(tmp) + sfn_normVector(tmp_0);
    tmp[0] = 3.0 * rtU.In2;
    tmp_0[0] = 6.0 * rtU.In2;
    tmp[1] = 4.0 * rtU.In2;
    tmp_0[1] = 7.0 * rtU.In2;
    tmp[2] = 5.0 * rtU.In2;
    tmp_0[2] = 8.0 * rtU.In2;
    rtY.Out2 = sfn_normVector_g(tmp) + sfn_normVector_g(tmp_0);
}

```

The code contained two identical functions: `sfn_normVector` and `sfn_normVector_g`.

In R2018a, the code generator produces this code:

```

static real_T sfn_normVector(const real_T v[3]);
static real_T sfn_normVector(const real_T v[3])
{
    return sqrt((v[0] * v[0] + v[1] * v[1]) + v[2] * v[2]);
}

void ex_function_elim_step(void)
{
    real_T tmp[3];
    real_T tmp_0[3];
    tmp[0] = rtU.In1;
    tmp_0[0] = 4.0 * rtU.In1;
    tmp[1] = 2.0 * rtU.In1;
    tmp_0[1] = 5.0 * rtU.In1;
    tmp[2] = 3.0 * rtU.In1;
    tmp_0[2] = 6.0 * rtU.In1;
    rtY.Out1 = sfn_normVector(tmp) + sfn_normVector(tmp_0);
    tmp[0] = 3.0 * rtU.In2;
    tmp_0[0] = 6.0 * rtU.In2;
    tmp[1] = 4.0 * rtU.In2;
    tmp_0[1] = 7.0 * rtU.In2;
    tmp[2] = 5.0 * rtU.In2;
    tmp_0[2] = 8.0 * rtU.In2;
    rtY.Out2 = sfn_normVector(tmp) + sfn_normVector(tmp_0);
}

```

The code contains one function `sf_normVector`.

Code Replacement: Optimize generated code with SIMD and row-major order support and improved library header file packaging

R2018a includes these code replacement enhancements:

- New Single Instruction, Multiple Data (SIMD) code replacement libraries that optimize load and store operations for Intel SSE processors.
 - Intel SSE (Windows)
 - Intel AVX (Windows)
 - Intel SSE (Linux)
 - Intel AVX (Linux)

The code generator optimizes the code by producing SIMD instructions. For more information, under “Performance” on page 10-26, see “Single Instruction, Multiple Data (SIMD) Instructions: Generate code with optimized load and store operations for use with Intel SSE processor.”

- For the MATLAB Coder environment, there is now this support for row-major order:
 - **Array layout supported by entry** menu in the Code Replacement Tool for creating row-major code replacement table entries. The menu appears when you set **Argument type** to **Matrix**. You can set **Array layout supported by entry** to **Column-major** (default), **Row-major**, or **Column-and-Row**.
 - Code replacement table entry property `ArrayLayout` for specifying row-major order programmatically. You can set the property to `COLUMN_MAJOR` or `ROW_MAJOR`.

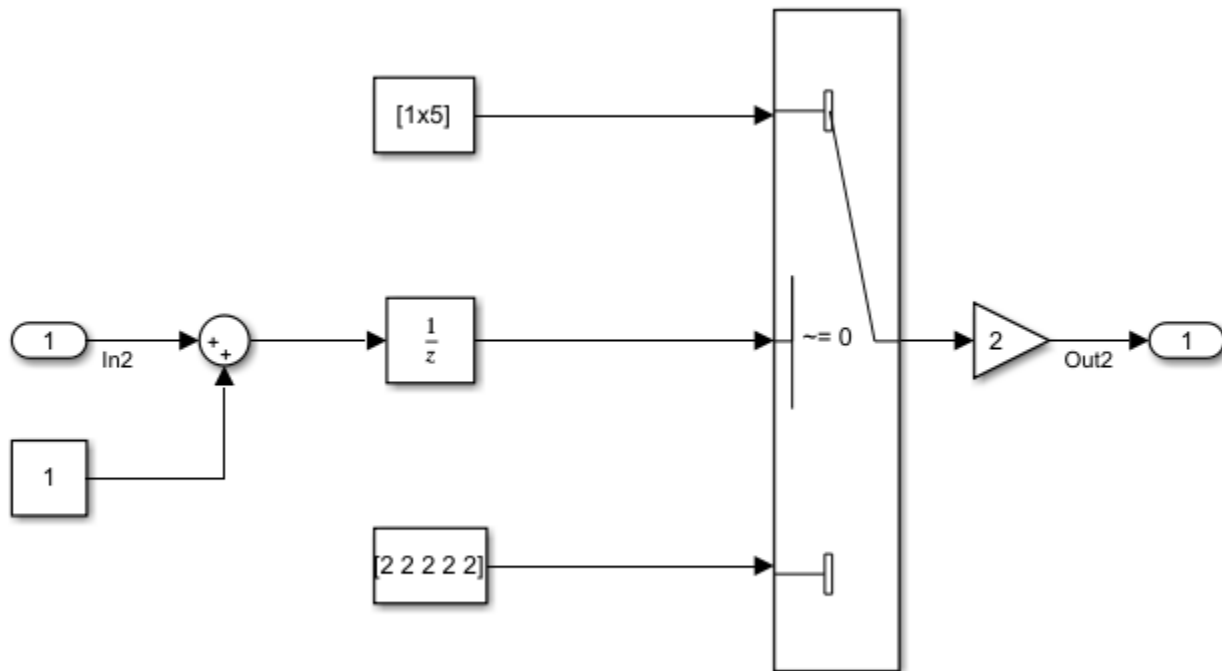
For more information, see [Define Code Replacement Mappings and Array Layout and Code Replacement](#).

- For the MATLAB Coder environment, the code generator now places code replacement library header files that it uses in the generated `.c` file instead of in the generated `.h` file.
- The list of available libraries has been updated to reflect more current and commonly used platforms and hardware.

Execution Speed: Move invariant code containing global variables out of for loops

In R2017b, if possible, the code generator moved invariant code out of `for` loops. In R2018a, the code generator can move invariant code containing global variables out of a `for` loop. This optimization improves execution speed because code that does not depend on a `for` loop executes only once instead of with every iteration of a `for` loop.

For example, in the model `invariant_global`, a Unit Delay block is the control input to the Switch block. The other two inputs are vectors of length five.



In R2017b, the model step function contained this code:

```
/* Model step function */
void invariant_global_step(void)
{
    int32_T i;
    uint8_T tmp;
    for (i = 0; i < 5; i++) {
        if (DWork.UnitDelay_DSTATE != 0.0) {
            tmp = 1U;
        } else {
            tmp = 2U;
        }

        Y.Out1[i] = (uint8_T)(tmp << 1);
    }

    DWork.UnitDelay_DSTATE = U.In2 + 1.0;
}
```

The for loop contained the if-else logic even though the for loop had no effect on this logic. The if-else logic contains the global variable, `DWork.UnitDelay_DSTATE`.

In R2018a, the model step function contains this code:

```
/* Model step function */
void invariant_global_step(void)
{
    int32_T i;
    uint8_T tmp;
    if (DWork.UnitDelay_DSTATE != 0.0) {
```

```
    tmp = 1U;
} else {
    tmp = 2U;
}

for (i = 0; i < 5; i++) {
    Y.Out1[i] = (uint8_T)(tmp << 1);
}

DWork.UnitDelay_DSTATE = U.In2 + 1.0;
}
```

The if-else logic executes before the for loop.

Verification

PIL Simulation: Verify initial values of global variables

At the start of a processor-in-the-loop (PIL) simulation, if **Remove root level I/O zero initialization** or **Remove internal data zero initialization** is selected and initial values of global variables in the target application are not zero, the software generates a warning. For more information, see [Verification of Code Generation Assumptions](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2017b

Version: 6.13

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Setup for MISRA C Compliance: Configure code generation parameters to increase compliance with MISRA C:2012 guidelines

When you generate C/C++ code from MATLAB code, if you have Embedded Coder, you can configure certain code generation parameters to increase the likelihood of generating code that complies with MISRA C:2012 guidelines. In R2017b, you can set these parameters in one step. If you generate code by using `codegen`, set the parameters with `coder.setupMISRAConfig`. To set the parameters in the MATLAB Coder app, see [Increase Likelihood of Generating MISRA C Compliant Code from MATLAB Code](#).

SIL/PIL Execution Performance: Speed up SIL or PIL execution by disabling constant input checking and global data synchronization

In R2017b, to speed up a SIL or PIL execution, you can disable constant input checking and global data synchronization. To disable these features, see [Speed Up SIL/PIL Execution by Disabling Constant Input Checking and Global Data Synchronization](#).

If you disable constant input checking or global data synchronization, the results of a SIL or PIL execution might be different from the results in MATLAB.

Execution-Time Profiling: Display time units in code execution profiling report

The execution-time profiling report from a SIL or PIL execution gives the time units for displayed metrics. For more information, see [View Execution Times](#).

Default Case for Switch Statements: Increase generated code compliance with coding standards

In previous releases, when you generated C/C++ code from MATLAB code, the code generator could produce switch statements without a default case. In R2017b, if you have Embedded Coder, you can specify that you want the code generator to produce a default case for all switch statements in the generated code. If you generate code with `codegen`, set the `GenerateDefaultInSwitch` configuration object property to `true`. In the MATLAB Coder app, set **Always generate a default case for switch** to **Yes**.

Some coding standards, such as MISRA, require the default case for switch statements.

Model Architecture and Design

Function Interfaces: Generate multi-instance functions from export-function models and control scope of Simulink functions

Generate reusable C/C++ function interfaces from export-function models

The code generator supports new Simulink component modeling styles so that you can include multiple instances of a Simulink function in an export function model (top or referenced), each with its own instance-specific data. The code generator produces multi-instance functions when you configure a:

- Top model with **Reusable function** set to `Reusable` or `C++ class` (C++ only).
- Referenced model with **Total number of instances allowed per top model** set to `Multiple`.

For more information, see [Design Models for Generated Embedded Code Deployment](#), [Generate Component Source Code for Export to External Code Base](#), and [Generate Reentrant Code from Simulink Function Blocks](#).

Compatibility Considerations

Before R2017b, for referenced models, the code generator produced entry-point functions that passed data as individual arguments. As a result, a small algorithmic change could produce a significant change to an entry-point function interface. Now the code generator captures instance-specific data in the real-time model (rtM) data structure and passes it as a self-argument in the entry-point functions.

Control whether accessibility of generated function code is global or scoped

The code generator supports added control over accessibility to Simulink function definitions within a model hierarchy that Simulink introduces in R2017b. The new Trigger block parameter **Function visibility** controls Simulink function accessibility within the context of a model hierarchy:

- **Global**--Can call the function from anywhere in the hierarchy. You define the function within a virtual subsystem or model and set the **Function visibility** parameter to `Global`. The function name must be unique.
- **Scoped**--Can call the function from one level above, at the same level, or from a level below the level of the function definition. You define the function within a virtual subsystem or model and set the **Function visibility** parameter to `Scoped`. You can scope a function in an atomic or nonvirtual subsystem, but function call accessibility is limited to the same level or below of the hierarchy. The function name does not have to be unique.

For more information, see [Simulink Function Blocks and Code Generation, Scoped and Global Simulink Function Blocks \(Simulink\)](#), [Scoping Simulink Functions in Subsystems \(Simulink\)](#), and [Scope Simulink Functions in Models \(Simulink\)](#).

AUTOSAR Compositions and Basic Software: Import AUTOSAR compositions and simulate diagnostic and memory services

Import AUTOSAR compositions as Simulink models

You can now import AUTOSAR compositions from `arxml` files into Simulink. AUTOSAR compositions aggregate AUTOSAR software components and potentially other compositions. Use the `arxml.importer.createCompositionAsModel` to import a composition. Use the function `updateModel` to update an imported composition with changes from `arxml` files.

For more information, see the `createCompositionAsModel` reference page and live-script example [Import AUTOSAR Composition to Simulink \(Embedded Coder Support Package for AUTOSAR Standard\)](#).

Simulate AUTOSAR diagnostic and memory services

R2016b introduced the AUTOSAR Basic Software (BSW) block library. The library provides preconfigured Function Caller blocks for modeling component calls to AUTOSAR BSW services. The BSW caller blocks support AUTOSAR code generation.

Before R2017b, you could not easily simulate a component that used the BSW blocks, because no model-level implementations existed for the BSW service functions called by the blocks.

In R2017b, the software provides reference implementations of the AUTOSAR Dem and NvM services supported by the BSW caller blocks. When coupled with the caller blocks, the reference implementations allow you to run system- or composition-level simulations of AUTOSAR BSW service calls. The ability to simulate calls into BSW services can help identify modeling problems before the AUTOSAR generated code reaches the AUTOSAR Runtime Environment (RTE).

For more information, see [Model AUTOSAR Basic Software Service Calls, Configure AUTOSAR Basic Software Service Implementations for Simulation, and live-script example Simulate AUTOSAR Basic Software Services and Runtime Environment \(Embedded Coder Support Package for AUTOSAR Standard\)](#).

AUTOSAR Sender-Receiver Communication: Model AUTOSAR queued send and receive using Simulink messages

In R2017b, you can use Simulink messages to model AUTOSAR queued sender-receiver communication between automotive components. Previously, you could model only nonqueued sender-receiver communication.

In Simulink, you can now model sending and receiving AUTOSAR data using a queue, and handling errors that occur when the queue is empty or full.

For more information, see [Configure AUTOSAR Queued Sender-Receiver Communication](#).

MISRA C: 2012 Modeling Checks: Improve compliance of generated code by using new MISRA C: 2012 standards checks

To improve MISRA C:2012 compliance, these new checks are available through the Model Advisor. To execute these checks, Select and Run Model Advisor Checks (Simulink) and select **By Product > Embedded Coder**.

Check	Description	Addresses MISRA C:2012
Check for missing error ports for AUTOSAR receiver interfaces	Identifies AUTOSAR receiver interface inports that do not have matching error ports.	Directive 4.7
Check for missing const qualifiers in model functions	Identifies input data pointers that do not have a const qualifier.	Rule 8.13
Check integer word length	Identifies integer word lengths that do not comply with hardware implementation settings.	Rule 10.1

Modifications to existing MISRA C:2012 compliance checks are outlined in this table.

Check	Description of Modification to the Check
Check for blocks not recommended for MISRA C:2012	Flags the inclusion of From Workspace blocks
Check configuration parameters for MISRA C:2012	<p>Flags the following parameter settings:</p> <ul style="list-style-type: none"> • Configuration parameter Wrap on overflow is set to none. • Configuration parameter Inf or NaN block output is set to none • Configuration parameter Inf or NaN block output set to none. • Configuration parameter Dynamic memory allocation in MATLAB Function blocks is selected. • Parameter <code>ERTFilePackagingFormat</code> is set to <code>Modular</code>. • Parameter <code>PreserveStaticInFcnDecls</code> is set to <code>off</code>. <p>hisl_0060: Configuration parameters that improve MISRA C:2012 compliance reflects these parameter settings.</p>
Check for switch case expressions without a default case	<p>Check can be executed on library models.</p> <p>Check can exclude blocks when you have Simulink Check.</p>
Check for bitwise operations on signed integers	Check can exclude blocks when you have Simulink Check.
Check for equality and inequality operations on floating-point values	Check can exclude blocks when you have Simulink Check.

For information about MISRA C versions and updates, see MISRA C Guidelines

Modeling Support for Secure Coding Standards: Check model for compliance with secure coding requirements in CERT C, CWE, ISO/IEC TS 17961 standards to improve security of generated code

You can use Model Advisor to check the model or subsystem for compliance with secure coding requirements in CERT C, CWE, and ISO/IEC TS 17961 standards. To execute these checks, Select and Run Model Advisor Checks (Simulink) and select **By Task > Modeling Guidelines for Secure Coding (CERT C, CWE, ISO/IEC TS 17961)**.

This table summarizes the Modeling Standards for Secure Coding checks.

Check	Description	Addresses Secure Coding Standards
Check configuration parameters for secure coding standards	Identifies configuration parameters that might impact code security.	
Check for blocks not recommended for C/C++ production code deployment	Identifies blocks not supported by code generation or not recommended for C/C++ production code deployment.	
Check for blocks not recommended for secure coding standards	Identifies blocks not supported by secure coding standards.	
Check usage of Assignment blocks	Identifies Assignment blocks that do not have block parameter Action if any output element is not assigned set to Error or Warning	<ul style="list-style-type: none"> • ISO/IEC TS 17961: 2013, uninitref • CERT C, EXP33-C • CWE, CWE-908
Check for switch case expressions without a default case	Identifies switch case expressions that do not have a default case.	<ul style="list-style-type: none"> • ISO/IEC TS 17961: 2013, swtchdflt • CERT C, MSC01-C • CWE, CWE-478
Check for bitwise operations on signed integers	Identifies Simulink blocks that contain bitwise operations on signed integers. The check does not flag MATLAB Function or Stateflow blocks that use signed operands for bitwise operators.	<ul style="list-style-type: none"> • CERT C, INT13-C • CWE, CWE-682
Check for equality and inequality operations on floating-point values	Identifies equality and inequality operations on floating-point values.	<ul style="list-style-type: none"> • CERT C, FLP00-C • CWE, CWE-697
Check integer word length	Identifies integer word lengths that do not comply with hardware implementation settings.	<ul style="list-style-type: none"> • CERT C, INT13-C • CWE, CWE-682

If you have Simulink Design Verifier™, the following design error detection checks are also available as part of the Modeling Standards for Secure Coding checks.

Check	Description	Addresses Secure Coding Standards
Detect Dead Logic	Identifies logic that stays inactive during simulation.	<ul style="list-style-type: none"> CERT C, MSC07-C CWE, CWE-561
Detect Integer Overflow	Identifies operations that exceed the data type range for integer or fixed-point operations.	<ul style="list-style-type: none"> ISO/IEC TS 17961: 2013, intoflow CERT C, INT30-C and INT32-C CWE, CWE-190
Detect Division by Zero	Identifies operations in the model that cause division-by-zero errors.	<ul style="list-style-type: none"> ISO/IEC TS 17961: 2013, diverr CERT C, INT33-C and FLP03-C CWE, CWE-369
Detect Out Of Bound Array Access	Detects operations that access outside the bounds of an array index	<ul style="list-style-type: none"> ISO/IEC TS 17961: 2013, invptr CERT C, ARR30-C CWE, CWE-118
Detect Violation of Specified Minimum and Maximum Values	Checks the specified minimum and maximum values (the design ranges) on intermediate signals throughout the model and on the output ports. If the analysis detects that a signal exceeds the design range, the results identify where in the model the errors occurred.	<ul style="list-style-type: none"> CERT C, API00-C CWE, CWE-628

For information about the secure coding standards organizations, see Secure Coding Standards.

Code Reuse: Generate reusable code for subsystems that contain data objects with imported custom storage classes

In R2017b, you can generate reusable subsystem code for models containing data objects with the following custom storage classes:

- ImportedDefine
- ImportFromFile
- user-defined custom storage class with the **Data Scope** parameter set to Imported

The reusable code is in the shared utilities folder (`slprj/target/_sharedutils`). Generating reusable code conserves ROM consumption and improves code execution speed. See Generate Reusable Code from Library Subsystems Shared Across Models (Simulink Coder).

Data, Function, and File Definition

Storage Class for Model Workspace Parameters: Apply custom storage classes to parameter objects in a model workspace

Before R2017b, you could not apply a storage class other than `Auto` to parameter objects (such as `Simulink.Parameter`) that you stored in a model workspace. In R2017b, you can apply a storage class, built-in custom storage class, or custom storage class that you create by using the Custom Storage Class Designer. For more information, see “Tunable Parameters: Tune parameters in model workspace”.

Custom Storage Class Simplification: Default removed from drop-down lists

In R2017b, as you apply a storage class to a data item interactively (for example, by using a block dialog box), by default, the built-in custom storage class `Default` does not appear in the drop-down list. However, if the data item already uses `Default` due to application in a previous release or to programmatic application, the custom storage class appears in the list.

Compatibility Considerations

You cannot use drop-down lists to apply `Default` to data items. However, your existing scripts that apply `Default` continue to work.

Instead of `Default`, consider using the built-in storage class `ExportedGlobal`.

Code Generation

Cross-Release Code Integration: Reuse code from models containing model references, global I/O, data stores, and parameters

The R2017b cross-release code integration workflow supports:

- Root-level I/O through global variables in generated code.
- Data store memory across the boundaries of code generated by different releases. In an integration model, current and previous release components can communicate through global data stores associated with Simulink.Signal objects in the MATLAB base workspace or a Simulink data dictionary.
- Parameter tuning in an integration model where component code from previous releases contains tunable parameters.
- The GetSet storage class for data store memory and tunable parameters.
- Model blocks inside components exported from previous releases.
- Multiple instances of a referenced model in these cases:
 - A cross-release SIL or PIL block contains code from one top model that calls multiple instances of the referenced model code. The integration model contains only one instance of the SIL or PIL block.
 - A cross-release SIL or PIL block contains code from a Model block that supports multiple instances. The integration model contains multiple instances of the SIL or PIL block.

For more information, see [Cross-Release Code Integration](#).

Cross-Release Code Integration: Run all workflow tasks from current release

To export code from a previous release, you can run `crossReleaseExport` from the current release provided the previous release is registered with `sharedCodeMATLABVersions`. You can also specify the location of the parent folder for the subfolders that contain cross-release artifacts .

Through the `sharedCodeUpdate` command, you can copy shared code source files from the shared code location specified by a cross-release artifact to the folder specified by the `ExistingSharedCode` parameter of a Simulink configuration set or model.

Through the `crossReleaseImport` command, you can import generated model code with custom code or include paths that have been relocated from their original folders.

For more information, see [Cross-Release Code Integration](#).

AUTOSAR Run-Time Calibration: Measure and calibrate signal and discrete state data using `arTypedPerInstanceMemory`

AUTOSAR typed per-instance memory (`arTypedPerInstanceMemory`), introduced in AUTOSAR schema version 4.0, defines an AUTOSAR typed memory block that is available for each instance of an AUTOSAR software component. In the AUTOSAR Runtime Environment (RTE), calibration tools can access `arTypedPerInstanceMemory` blocks for measurement and calibration.

Previously, you could model `arTypedPerInstanceMemory` in Simulink by creating an `AUTOSAR.Signal` data object and referencing it in a Data Store Memory block.

In R2017b, you can also generate `arTypedPerInstanceMemory` blocks for block signal and discrete state data in your AUTOSAR model. Configure the signals and states to use `SimulinkGlobal` storage class. For more information, see [Per-Instance Memory and Configure AUTOSAR Per-Instance Memory](#).

Stateflow Element Traceability: Obtain enhanced inline traceability

In the Configuration Parameters dialog box, you check the **Code-to-model** and the **Model-to-code** parameters to get inline traceability in the generated code. In R2017a, only the Stateflow states and transitions with actions had inline traceability support. R2017b provides complete line-level traceability coverage for Stateflow elements with or without comments.

From the code generation report, click a hyperlinked line of code to navigate to corresponding blocks in the model. When you click the hyperlink, it highlights single or multiple Stateflow elements at the same time. From a block or blocks in your model, right-click the block and select **C/C++ Code > Navigate To C/C++ Code**. In the code generation report, highlighted lines of code correspond to your model blocks.

For more information, see [Inline Traceability for Stateflow Elements](#)

Stateflow Objects and MATLAB User Comments: Configure comments flexibly

You can now separately control comment configuration for **Simulink block comments** and **Stateflow object comments**. To enable control over MATLAB user comments, in the Configuration Parameters dialog box, you must turn on the **MATLAB user comments**.

Stateflow object comments and **MATLAB user comments** are off by default for the ERT target. When you load an existing model in Simulink, all the comment parameters keep their current values except for the new parameter **Stateflow object comments** which takes the same value as **Simulink block comments**.

This table lists the default values for these comment parameters.

Parameter Name		GRT		ERT	
R2017a	R2017b	R2017a	R2017b	R2017a	R2017b
Simulink block/Stateflow object comments	Simulink block comments	On	On	On	On
Simulink block/Stateflow object comments	Stateflow object comments	On	Off	On	Off

Parameter Name		GRT		ERT	
R2017a	R2017b	R2017a	R2017b	R2017a	R2017b
MATLAB source code as comments	MATLAB source code as comments	Off	Off	On	Off
MATLAB function help text	MATLAB user comments	N/A	N/A	On	Off

Enhanced Shared Utilities Naming: Customize the names of shared utility functions that are inside MATLAB Function blocks

In R2017a, the code generator mangled the names of shared utility functions that were inside a MATLAB Function block. In R2017b, the code generator uses the value of the **Shared utilities** parameter to name these functions. In the Configuration Parameters dialog box, this parameter is on the **Code Generation > Symbols** pane. For more information, see Control Naming of Generated Functions (Simulink Coder).

Checksum Length: Specify the character length of the \$C token

In R2017b, in the Configuration Parameters dialog box, you can use the new Shared checksum length (Simulink Coder) parameter to specify the length of the \$C token. This parameter default value is eight characters. During code generation, if you get an error informing you of a potential naming clash, you can increase this parameter value to avoid the clash.

Code Style: Generate static keyword for locally scoped functions

When you use Compact/Compact (with separate data file) file packaging, you can now generate static functions. Enable or disable the generation of static functions by selecting Preserve static keyword in function declarations parameter. This parameter is on by default for Compact/Compact (with separate data file) packaging. When you select this parameter, the generated code is compliant with MISRA C:2012 Rule 8.10.

You can link different executables that refer to locally scoped subsystem and utility functions with the same name. This parameter also impacts these functions:

- Stateflow graphical function
- Variant subsystem
- MATLAB subfunction
- Privately scoped Simulink function

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
```

- The name of the **Threshold** parameter is now **Maximum number of arguments for subsystem outputs**.

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

Improved Readability of the Generated Code: Include parentheses around compound expressions containing right-shift operators

In R2107b, the code generator inserts parentheses around compound expressions that are on either side of right-shift operators. The inclusion of parentheses improves the readability of the code and satisfies MISRA C:2012 Rule 12.1, which states that the precedence of operators within expressions should be made explicit.

For example, in R2017a, the generated code did not contain parentheses around the multiplicative operation:

```
modelex_DW.Delay_DSTATE[0] = modelex_P.Gain_Gain_o *  
    modelex_DW.Delay_DSTATE[0] >> 6;
```

In R2017b, the generated code contains parentheses around the multiplicative operation:

```
modelex_DW.Delay_DSTATE[0] = (modelex_P.Gain_Gain_o *  
    modelex_DW.Delay_DSTATE[0]) >> 6;
```

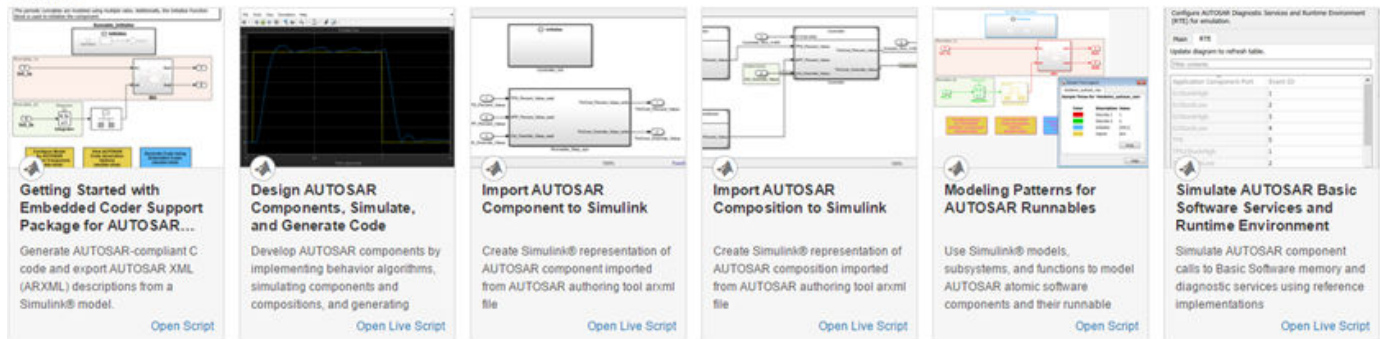
In the Configuration Parameters dialog box, for the **Parentheses level** parameter, this enhancement occurs for the Nominal (Optimize for readability) setting.

Deployment

AUTOSAR Support Package: Run live-script examples for AUTOSAR compositions and Basic Software

The Embedded Coder Support Package for AUTOSAR Standard now installs AUTOSAR featured examples, including new live-script examples.

Examples



Support Package renamed to Embedded Coder Support Package for Intel SoC Devices

The Embedded Coder Support Package for Altera® SoC Platform is now called to the Embedded Coder Support Package for Intel SoC Devices.

Support Package renamed to Embedded Coder Support Package for Xilinx Zynq Platform

The Embedded Coder Support Package for Xilinx Zynq-7000 Platform is now called the Embedded Coder Support Package for Xilinx Zynq Platform.

Removed Support for Wind River VxWorks Hardware

The Embedded Coder Support Package for Wind River® VxWorks® RTOS has been removed and is no longer available. You can still generate ANSI/ISO C/C++ code for the processors that are supported by the Wind River VxWorks real-time operating system (RTOS). However, you must manually integrate the generated code with your own scheduler and drivers.

Performance

RAM Reduction: Reduce data copies in For Each subsystems and reuse buffers of different sizes

In R2017a, the code generator could reuse buffers for matrices that had the same sizes and shapes. In R2017b, the code generator can reuse buffers for matrices that have different sizes and shapes.

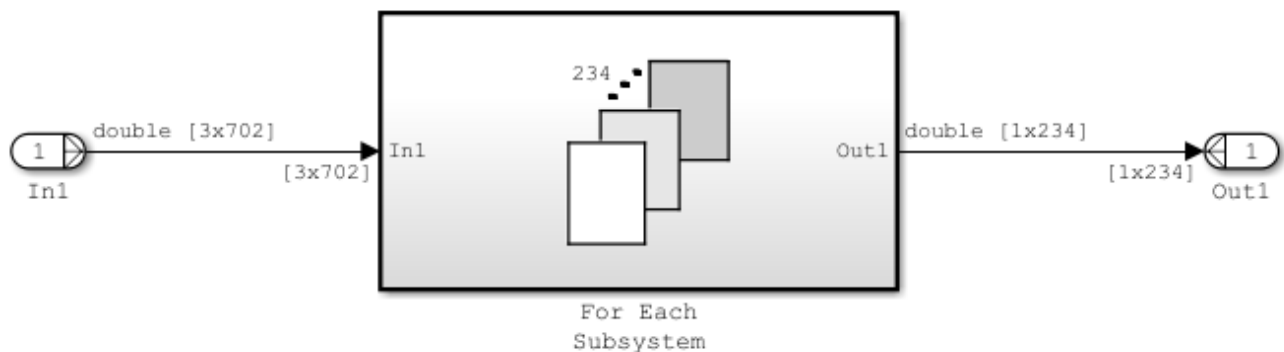
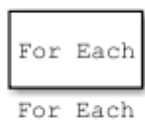
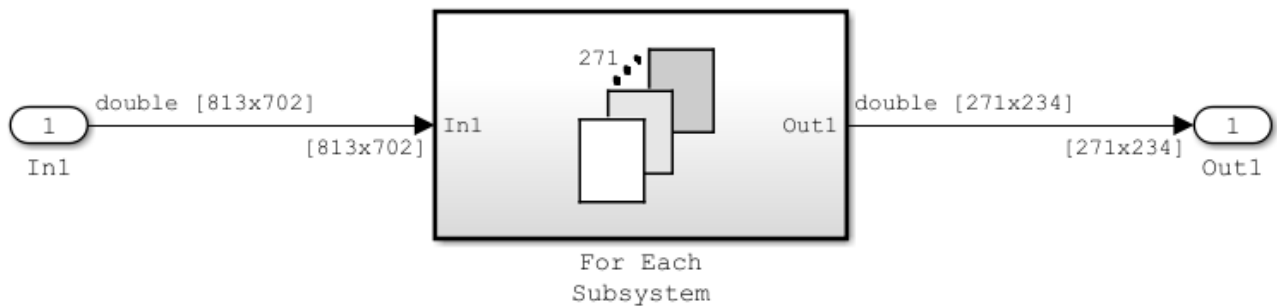
For For Each Subsystem blocks, the code generator can perform these optimizations:

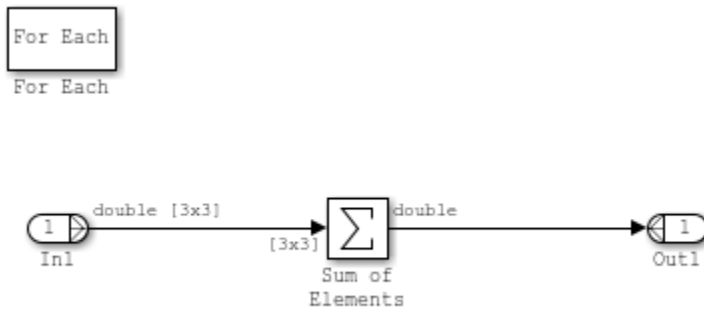
- Combine more for loops.
- Unroll more for loops whose size is under the **Loop Unrolling Threshold** parameter value.
- Generate fewer data copies for For Each Subsystem block input and output ports.

These optimizations conserve RAM and ROM consumption and improve code execution speed.

Buffer reuse in For Each subsystems

The model `foreach_codegen` contains a For Each Subsystem block inside of a For Each Subsystem block. The nested For Each Subsystem block contains a Sum of Elements block.





In R2017a, the code generator produced this code:

```
void foreach_codegen_step(void)
{
    int32_T ForEach_itr;
    int32_T ForEach_itr_d;
    real_T tmp;
    real_T rtb_Out1_CoreSubsysCanOut[234];
    real_T rtb_ImpSel_InsertedFor_In1_at_c[9];
    int32_T i;
    int32_T i_0;
    for (ForEach_itr = 0; ForEach_itr < 271; ForEach_itr++) {
        i = ForEach_itr * 3;
        for (i_0 = 0; i_0 < 702; i_0++) {
            foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[3 * i_0] =
                foreach_codegen_U.In1[813 * i_0 + i];
            foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[1 + 3 * i_0] =
                foreach_codegen_U.In1[(813 * i_0 + i) + 1];
            foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[2 + 3 * i_0] =
                foreach_codegen_U.In1[(813 * i_0 + i) + 2];
        }

        for (ForEach_itr_d = 0; ForEach_itr_d < 234; ForEach_itr_d++) {
            i = ForEach_itr_d * 3;
            for (i_0 = 0; i_0 < 3; i_0++) {
                rtb_ImpSel_InsertedFor_In1_at_c[3 * i_0] =
                    foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[(i_0 + i) * 3];
                rtb_ImpSel_InsertedFor_In1_at_c[1 + 3 * i_0] =
                    foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[(i_0 + i) * 3 + 1];
                rtb_ImpSel_InsertedFor_In1_at_c[2 + 3 * i_0] =
                    foreach_codegen_B.ImpSel_InsertedFor_In1_at_o[(i_0 + i) * 3 + 2];
            }

            tmp = rtb_ImpSel_InsertedFor_In1_at_c[0];
            for (i = 0; i < 8; i++) {
                tmp += rtb_ImpSel_InsertedFor_In1_at_c[i + 1];
            }

            rtb_Out1_CoreSubsysCanOut[ForEach_itr_d] = tmp;
        }

        for (i = 0; i < 234; i++) {
            foreach_codegen_Y.Out1[ForEach_itr + 271 * i] =
                rtb_Out1_CoreSubsysCanOut[i];
        }
    }
}
```



```

    }
  }
}

```

For holding copies of input and output data, the generated code contained:

- The temporary local array `rtb_Out1_CoreSubsysCanOut`
- The temporary local variable `rtb_ImpSel_InsertedFor_In1_at_c[9]`
- The global temporary array `ImpSel_InsertedFor_In1_at_o`

For the `for` loops that contained these data copies, the code contained the iterator `i_0`.

In R2017b, the code generator produces this code:

```

void foreach_codegen_step(void)
{
    real_T tmp;
    int32_T i;
    int32_T ForEach_itr;
    int32_T ForEach_itr_d;
    for (ForEach_itr = 0; ForEach_itr < 271; ForEach_itr++) {
        for (ForEach_itr_d = 0; ForEach_itr_d < 234; ForEach_itr_d++) {
            tmp = -0.0;
            for (i = 0; i < 9; i++) {
                tmp += foreach_codegen_U.In1[((i / 3 + ForEach_itr_d * 3) * 813 + i % 3)
                    + ForEach_itr * 3];
            }

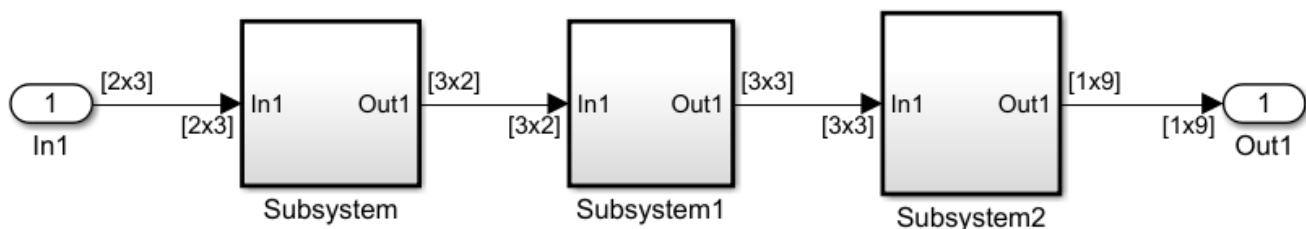
            foreach_codegen_Y.Out1[ForEach_itr + 271 * ForEach_itr_d] = tmp;
        }
    }
}

```

The variables and their data copies are not in the generated code.

Buffer reuse for arrays of different sizes and dimensions

The model `diffSizeAutoReuse` contains matrices of different sizes and shapes.



In R2017a, the `diffSizeAutoReuse.c` file contained this code:

```

void diffSizeAutoReuse_step(void)
{
    real_T rtb_y_k[6];
    real_T rtb_y_l3[9];

```

```
diffSizeAutoReuse_Subsystem(diffSizeAutoReuse_U.In1, rtb_y_k);
diffSizeAutoReuse_Subsystem1(rtb_y_k, rtb_y_l3);
diffSizeAutoReuse_Subsystem2(rtb_y_l3, diffSizeAutoReuse_Y.Out1);
}
```

The arrays `rtb_y_k` and `rtb_y_l3` held the data of different sizes.

In R2017b, the `diffSizeAutoReuse.c` file contains this code:

```
void diffSizeAutoReuse_step(void)
{
    diffSizeAutoReuse_Subsystem(diffSizeAutoReuse_U.In1,
        &diffSizeAutoReuse_Y.Out1[0]);
    diffSizeAutoReuse_Subsystem1(&diffSizeAutoReuse_Y.Out1[0],
        diffSizeAutoReuse_Y.Out1);
    diffSizeAutoReuse_Subsystem2(diffSizeAutoReuse_Y.Out1,
        diffSizeAutoReuse_Y.Out1);
}
```

The variable `diffSizeAutoReuse_Y.Out1` holds the data of different sizes, so there are two fewer buffers in the generated code. For reusing buffers for matrices of different sizes and shapes, note these limitations:

- The code generator does not replace a buffer with a lower priority buffer that has a smaller size.
- The code generator does not reuse buffers that have different sizes and symbolic dimensions.

Reusable Storage Class: Specify reusable custom storage classes anywhere on a path

Previously, you could use reusable custom storage classes to specify buffer reuse on multiple signals in a path. Now, you can use reusable custom storage classes to specify buffer reuse on discontinuous signals. For example, you can interleave reusable custom storage classes on a path. Specifying buffer reuse by applying custom storage classes has these benefits:

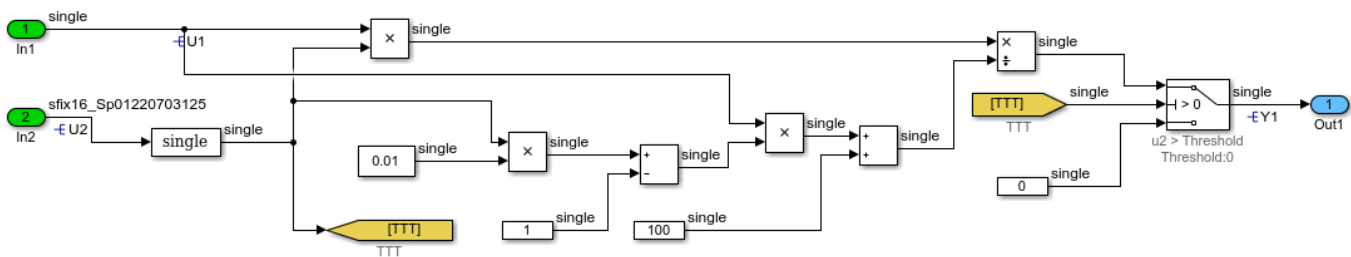
- Eliminate data copies.
- Conserve ROM and RAM consumption.
- Improve execution speed.
- Control how signal and state data interface with externally-written code.

You can specify buffer reuse on signals that the code generator cannot honor. For those cases, use two new diagnostics to specify the message type that the model displays. In the Configuration Parameters dialog box, these diagnostics are Detect non-reused custom storage classes (Simulink) and Detect ambiguous custom storage class final values (Simulink). For more information, see Specify Buffer Reuse by Using Simulink.Signal Objects.

Execution Speed: Eliminate redundant subexpressions

Previously, for some models that contained redundant subexpressions (that is, an expression that is part of another expression), the generated code repeatedly calculated the value of the subexpression. In R2017b, the generated code contains a temporary variable that holds the value of these subexpressions. This optimization improves the execution speed of the generated code because it eliminates redundant calculations. The parameter **Eliminate superfluous local variables (expression folding)** enables this optimization.

For example, the model M01 contains a series of math operations prior to the Switch block.



In R2017a, the M01_step function contained this code:

```
void M01_step(void)
{
    if ((real32_T)U2 * 0.0122070313F > 0.0F) {
        Y1 = (real32_T)U2 * 0.0122070313F * U1 / (((real32_T)U2 * 0.0122070313F *
            0.01F - 1.0F) * U1 + 100.0F);
    } else {
        Y1 = 0.0F;
    }
}
```

The M01_step function contained three instances of the subexpression $U2 * 0.0122070313F$.

In R2017b, the M01_step function contains this code:

```
void M01_step(void)
{
    real32_T tmp;
    tmp = (real32_T)U2 * 0.0122070313F;
    if (tmp > 0.0F) {
        Y1 = tmp * U1 / ((tmp * 0.01F - 1.0F) * U1 + 100.0F);
    } else {
        Y1 = 0.0F;
    }
}
```

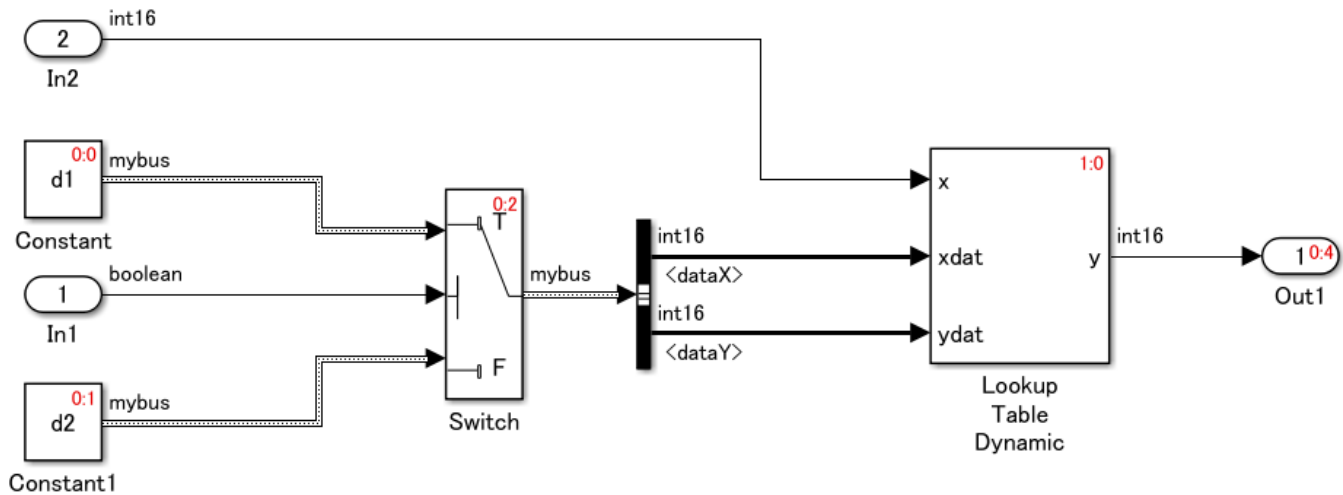
The M01_step function contains the variable tmp to hold the value of the subexpression $U2 * 0.0122070313F$.

Note This optimization does not occur when the simulation mode is Rapid Accelerator.

Execution Speed: Convert data copies to pointer assignments for more modeling patterns

In R2017b, the code generator can eliminate data copies for more modeling patterns involving vector signal assignments. The code generator can eliminate data copies for generated code that transfers data to and from structure fields. This optimization eliminates data copies for modeling patterns involving data transference between Simulink.Bus objects and Lookup Table blocks or reusable subsystems.

For example, two inputs to the model structure_pointer_conversion are the Simulink.Bus object mybus. mybus contains two vectors that each have a dimension of 100.



In R2017a, the `structure_pointer_conversion.c` file contained this code:

```

/* Model step function */
void structure_pointer_conversion_step(void)
{
    int16_T rtb_Switch_dataX[100];
    int16_T rtb_Switch_dataY[100];

    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/In1'
     */
    if (rtU.In1) {
        /* Switch: '<Root>/Switch' incorporates:
         * Constant: '<Root>/Constant'
         */
        memcpy(&rtb_Switch_dataX[0], &rtConstP.Constant_Value.dataX[0], 100U *
            sizeof(int16_T));
        memcpy(&rtb_Switch_dataY[0], &rtConstP.Constant_Value.dataY[0], 100U *
            sizeof(int16_T));
    } else {
        /* Switch: '<Root>/Switch' incorporates:
         * Constant: '<Root>/Constant1'
         */
        memcpy(&rtb_Switch_dataX[0], &rtConstP.Constant1_Value.dataX[0], 100U *
            sizeof(int16_T));
        memcpy(&rtb_Switch_dataY[0], &rtConstP.Constant1_Value.dataY[0], 100U *
            sizeof(int16_T));
    }

    /* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
     * Inport: '<Root>/In2'
     * Outport: '<Root>/Out1'
     */
    /* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
     * Input0 Data Type: Integer      S16
     * Input1 Data Type: Integer      S16
     * Input2 Data Type: Integer      S16
     * Output0 Data Type: Integer     S16
     * Lookup Method: Linear_Endpoint
    */
}

```

```

    *
    */
    LookUp_S16_S16( &(rtY.Out1), &rtb_Switch_dataY[0], rtU.In2, &rtb_Switch_dataX
                  [0], 99U);
}

```

In R2017b, the `structure_pointer_conversion.h` file contains the same variable declaration as the file did in R2017a.

```

typedef struct {
    int16_T dataX[100];
    int16_T dataY[100];
} mybus;

#endif

/* Constant parameters (auto storage) */
typedef struct {
    /* Expression: d1
    * Referenced by: '<Root>/Constant'
    */
    mybus Constant_Value;

    /* Expression: d2
    * Referenced by: '<Root>/Constant1'
    */
    mybus Constant1_Value;
} ConstParam;

```

The generated code contained four data copies from the structure fields `Constant_Value.dataX`, `Constant_Value.dataY`, `Constant1_Value.dataX`, and `Constant1_Value.dataY` to the local variables `rtb_Switch_dataX` and `rtb_Switch_dataY`.

In R2017b, the `structure_pointer_conversion.c` file contains this code:

```

/* Model step function */
void structure_pointer_conversion_step(void)
{
    const int16_T *rtb_Switch_dataX;
    const int16_T *rtb_Switch_dataY;

    /* Switch: '<Root>/Switch' incorporates:
    * Inport: '<Root>/In1'
    */
    if (rtU.In1) {
        /* Switch: '<Root>/Switch' incorporates:
        * Constant: '<Root>/Constant'
        */
        rtb_Switch_dataX = (&rtConstP.Constant_Value.dataX[0]);
        rtb_Switch_dataY = (&rtConstP.Constant_Value.dataY[0]);
    } else {
        /* Switch: '<Root>/Switch' incorporates:
        * Constant: '<Root>/Constant1'
        */
        rtb_Switch_dataX = (&rtConstP.Constant1_Value.dataX[0]);
        rtb_Switch_dataY = (&rtConstP.Constant1_Value.dataY[0]);
    }
}

```

```

/* S-Function (sfix_look1_dyn): '<Root>/Lookup Table Dynamic' incorporates:
 * Inport: '<Root>/In2'
 * Output: '<Root>/Out1'
 */
/* Dynamic Look-Up Table Block: '<Root>/Lookup Table Dynamic'
 * Input0 Data Type: Integer      S16
 * Input1 Data Type: Integer      S16
 * Input2 Data Type: Integer      S16
 * Output0 Data Type: Integer     S16
 * Lookup Method: Linear_Endpoint
 *
 */
Lookup_S16_S16( &(rtY.Out1), &rtb_Switch_dataY[0], rtU.In2, &rtb_Switch_dataX
               [0], 99U);
}

```

The `structure_pointer_conversion.h` file contains the same variable declaration as was in R2017a.

```

typedef struct {
    int16_T dataX[100];
    int16_T dataY[100];
} mybus;

#endif

/* Constant parameters (auto storage) */
typedef struct {
    /* Expression: d1
     * Referenced by: '<Root>/Constant'
     */
    mybus Constant_Value;

    /* Expression: d2
     * Referenced by: '<Root>/Constant1'
     */
    mybus Constant1_Value;
} ConstParam;

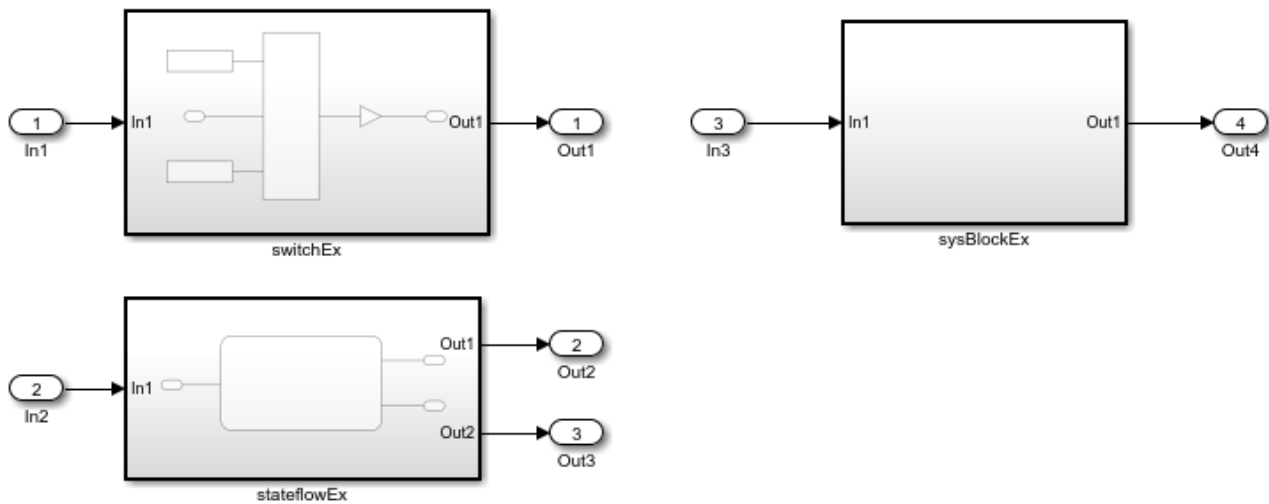
```

The generated code contains pointer assignments from the structure fields `Constant_Value.dataX`, `Constant_Value.dataY`, `Constant1_Value.dataX`, and `Constant1_Value.dataY` to the local pointers `rtb_Switch_dataX` and `rtb_Switch_dataY`. The data copies are not in the generated code. For more information, see [Convert Data Copies to Pointer Assignments](#).

Execution Speed: Move invariant code out of for loops

In R2017b, the code generator can move invariant code out of a `for` loop. This optimization improves execution speed because code that does not depend on a `for` loop executes only once instead of with every iteration of the `for` loop.

For example, the model `for_loop` contains three modeling patterns that produce `for` loops in the generated code. The subsystems `switchEx` and `stateflowEx` contain vectors that the code generator assigns to the subsystem outputs. The subsystem `sysBlockEx` contains a MATLAB System block with `switch-case` logic.



In R2017a, the code generator produced this code:

```
void for_loop_stateflowEx(real_T rtu_In1, real_T rty_Out1[7], real_T rty_Out2[7])
{
    int32_T i;
    for (i = 0; i < 7; i++) {
        rty_Out1[i] = 0.0;
        rty_Out2[i] = 0.0;
    }

    for (i = 0; i < 7; i++) {
        if (rtu_In1 < 0.0) {
            rty_Out1[2] = 45.0;
        }

        rty_Out2[i] = ((real_T)i + 1.0) + rtu_In1;
    }
}
```

```
void for_loop_switchEx(real_T rtu_In1, real_T rty_Out1[10])
{
    int32_T i;
    int32_T rtu_In1_0;
    for (i = 0; i < 10; i++) {
        if (rtu_In1 > 0.0) {
            rtu_In1_0 = 1;
        } else {
            rtu_In1_0 = 0;
        }

        rty_Out1[i] = 13.0 * (real_T)rtu_In1_0;
    }
}
...
```

```
void for_loop_sysBlockEx(const real_T rtu_In1[50], B_sysBlockEx_for_loop_T
```

```

*localB)
{
    real_T varargout_1[50];
    real_T scalarTmp;
    real_T readOnlyMatrixTmp[50];
    real_T switchExpr;
    int32_T i;
    scalarTmp = rtu_In1[49] - 1.0;
    for (i = 0; i < 50; i++) {
        varargout_1[i] = rtu_In1[i] - 1.0;
        readOnlyMatrixTmp[i] = rtu_In1[i] - 1.0;
    }

    for (i = 0; i < 47; i++) {
        switchExpr = 2.0 * readOnlyMatrixTmp[0];
        switch ((int32_T)switchExpr) {
            case 0:
                scalarTmp = 13.0;
                break;

            case 2:
                scalarTmp = 23.0;
                break;

            default:
                scalarTmp = 33.0;
                break;
        }

        varargout_1[i]++;
    }

    varargout_1[0] = scalarTmp + switchExpr;
    memcpy(&localB->systemBlockEx[0], &varargout_1[0], 50U * sizeof(real_T));
}

```

- In the `for_loop_stateflowEx` function, the `if` statement is invariant to the `for` loop.
- In the `for_loop_switchEx` function, the `if-else` statement is invariant to the `for` loop.
- In the `for_loop_sysBlockEx` function, the `switch-case` statements are invariant to the `for` loop.

```

void for_loop_stateflowEx(real_T rtu_In1, real_T rty_Out1[7], real_T rty_Out2[7])
{
    int32_T i;
    for (i = 0; i < 7; i++) {
        rty_Out1[i] = 0.0;
        rty_Out2[i] = 0.0;
    }

    if (rtu_In1 < 0.0) {
        rty_Out1[2] = 45.0;
    }

    for (i = 0; i < 7; i++) {
        rty_Out2[i] = ((real_T)i + 1.0) + rtu_In1;
    }
}

```



```

void for_loop_switchEx(real_T rtu_In1, real_T rty_Out1[10])
{
    int32_T i;
    int32_T rtu_In1_0;
    if (rtu_In1 > 0.0) {
        rtu_In1_0 = 1;
    } else {
        rtu_In1_0 = 0;
    }

    for (i = 0; i < 10; i++) {
        rty_Out1[i] = 13.0 * (real_T)rtu_In1_0;
    }
}...

void for_loop_sysBlockEx(const real_T rtu_In1[50], B_sysBlockEx_for_loop_T
*localB)
{
    real_T tmp[50];
    real_T scalarTmp;
    real_T readOnlyMatrixTmp[50];
    real_T switchExpr;
    int32_T i;
    for (i = 0; i < 50; i++) {
        tmp[i] = rtu_In1[i] - 1.0;
        readOnlyMatrixTmp[i] = rtu_In1[i] - 1.0;
    }

    switchExpr = 2.0 * readOnlyMatrixTmp[0];
    switch ((int32_T)switchExpr) {
        case 0:
            scalarTmp = 13.0;
            break;

        case 2:
            scalarTmp = 23.0;
            break;

        default:
            scalarTmp = 33.0;
            break;
    }

    for (i = 0; i < 47; i++) {
        tmp[i]++;
    }

    tmp[0] = scalarTmp + switchExpr;
    memcpy(&localB->systemBlockEx[0], &tmp[0], 50U * sizeof(real_T));
}

```

- In the `for_loop_stateflowEx` function, the `if` statement is not in the `for` loop.
- In the `for_loop_switchEx` function, the `if-else` statement is not in the `for` loop.
- In the `for_loop_sysBlockEx` function, the `switch-case` statements are not in the `for` loop.

Block Reordering for Improved Execution Efficiency: Change block execution order to enable buffer reuse and loop fusion

In R2017b, for more modeling patterns, the code generator can optimize the block execution order to improve execution efficiency. In the Configuration Parameters dialog box, when you set the **Optimize block operation order in the generated code** to Improved Execution Speed, the code generator can reorder block operations to perform these optimizations:

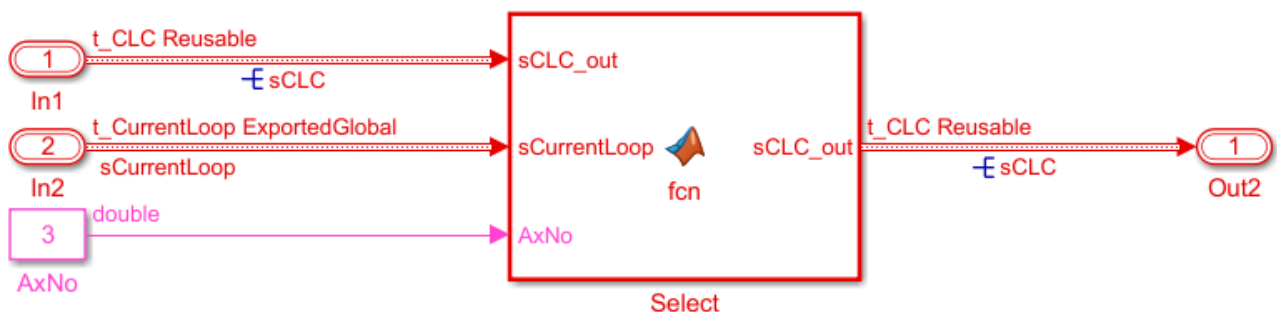
- Reuse the same variable for the input, output, and state of a Unit Delay block by executing the Unit Delay block before upstream blocks.
- Combine more `for` loops by executing blocks together that have the same size.
- Eliminate data copies by executing blocks together that meet these conditions:
 - Perform inplace operations (that is, use the same input and output variable).
 - Contain algorithm code with unnecessary data copies.

These optimizations improve execution speed and conserve RAM and ROM consumption. For more information, see Improve Execution Efficiency by Reordering Block Operations in the Generated Code.

MATLAB Function Block Buffer Reuse: Perform inplace assignment with root I/O

Since R2016a, you can specify the same variable name for the input and output of a MATLAB Function block. The code generator tries to reuse the input and output variables. When a MATLAB Function block connects directly to the root-level input and output ports, the code generator can now reuse the input and output variables. In this case, you must specify the same reusable custom storage class on the input and output signals. This optimization conserves RAM/ROM consumption by reducing the number of local variables and data copies in the generated code.

For example, in the model `SmallSelect`, the MATLAB Function block assigns a value to the signal coming from the root-level input port `In1`. The output signal connects directly to the root-level output port `Out2`.



The `SmallSelect_step` function contains this code:

```
/* Model step function */
void SmallSelect_step(void)
{
```

```

/* MATLAB Function: '<Root>/Select' incorporates:
 * Inport: '<Root>/In2'
 */
/* MATLAB Function 'Select': '<S1>:1' */
/* '<S1>:1:1' */
/* '<S1>:1:9' */
sCLC.asAx[2].sCurrentLoop = sCurrentLoop;
}

```

There are no unnecessary data copies in the generated code. For more information on how to specify buffer reuse with MATLAB Function blocks, see [Specify Buffer Reuse for MATLAB Function Blocks in a Path](#).

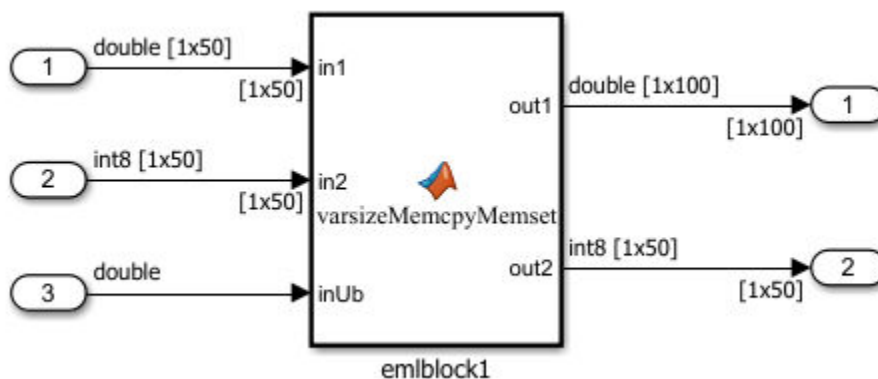
Execution-Time Profiling: Display time units in code execution profiling report and Simulation Data Inspector

The execution-time profiling report from a SIL or PIL simulation gives the time units for the displayed execution-time metrics. For more information, see [View and Compare Code Execution Times](#).

memcpy and memset Optimization: Generate more efficient code for variable-size arrays

In R2017a, the code generator attempted to replace fixed-size for loop controlled array element assignments with `memcpy` and `memset` function calls. A fixed-size array is one in which the number of array elements to assign is known at compile time. In R2017b, the code generator can replace variable-size for loop controlled array element assignments with `memcpy` and `memset` function calls. This optimization improves execution efficiency.

Modeling patterns that can produce variable-size for loop controlled array element assignments include variable-size signals and MATLAB function blocks containing variable-size arrays or data copies for a variable number of elements. For example, the model `varsize_ex` contains a MATLAB Function block.



The MATLAB function block contains this code:

```

function [out1, out2] = varsizeMemcpyMemset(in1,in2,inUb)
    out1 = zeros(1,100);
    %memcpy
    out1(1:inUb) = in1(1:inUb);

```

```

    out2 = in2;
    %memset
    out2(1:inUb) = repmat(int8(127),1,inUb);
end

```

In R2017a, the code generator produced this code:

```

void varsize_ex_step(void)
{
    real_T rtb_out1[100];
    int8_T rtb_out2[50];
    int32_T i;
    int32_T loop_ub;
    for (i = 0; i < 100; i++) {
        rtb_out1[i] = 0.0;
    }

    if (1.0 > varsize_ex_U.In3) {
        i = 0;
    } else {
        i = (int32_T)varsize_ex_U.In3;
    }

    loop_ub = i - 1;
    for (i = 0; i <= loop_ub; i++) {
        rtb_out1[i] = varsize_ex_U.In1[i];
    }

    memcpy(&rtb_out2[0], &varsize_ex_U.In2[0], 50U * sizeof(int8_T));
    loop_ub = (int32_T)varsize_ex_U.In3;
    for (i = 0; i < loop_ub; i++) {
        rtb_out2[i] = MAX_int8_T;
    }

    memcpy(&varsize_ex_Y.Out1[0], &rtb_out1[0], 100U * sizeof(real_T));
    memcpy(&varsize_ex_Y.Out2[0], &rtb_out2[0], 50U * sizeof(int8_T));
}

```

The generated code contained variable-size for loop controlled array element assignments to `rtb_out1` and `rtb_out2`.

In R2017b, the code generator produces this code:

```

void varsize_ex_step(void)
{
    real_T rtb_out1[100];
    int8_T rtb_out2[50];
    int32_T i;
    for (i = 0; i < 100; i++) {
        rtb_out1[i] = 0.0;
    }

    if (1.0 > varsize_ex_U.In3) {
        i = 0;
    } else {
        i = (int32_T)varsize_ex_U.In3;
    }

    i--;
    if (0 <= i) {
        memcpy(&rtb_out1[0], &varsize_ex_U.In1[0], (i + 1) * sizeof(real_T));
    }
}

```

```

}

memcpy(&rtb_out2[0], &varsize_ex_U.In2[0], 50U * sizeof(int8_T));
i = (int32_T)varsize_ex_U.In3;
if (0 <= i - 1) {
    memset(&rtb_out2[0], 127, i * sizeof(int8_T));
}

memcpy(&varsize_ex_Y.Out1[0], &rtb_out1[0], 100U * sizeof(real_T));
memcpy(&varsize_ex_Y.Out2[0], &rtb_out2[0], 50U * sizeof(int8_T));
}

```

For assigning values to `rtb_out1` and `rtb_out2`, the generated code contains `memcpy` and `memset` functions.

Data Copy Reduction: Generate fewer data copies at function call sites

In R2017b, the generated code contains fewer data copies for blocks that have a dedicated function. The code generator eliminates these data copies by generating a function that writes directly to the destination variable rather than to a temporary buffer at the block output.

For example, the model `backfolding_ex` contains a Constant block feeding into a Discrete FIR Filter block.



In R2017a, the code generator produced this code:

```

/* Model step function */
void backfolding_ex_step(void)
{
    real32_T rtb_DiscreteFIRFilter[320];

    /* DiscreteFir: '<Root>/Discrete FIR Filter' incorporates:
     * Constant: '<Root>/Constant'
     */
    arm_fir_f32(&backfolding_ex_DW.S, &backfolding_ex_DW.pState[0],
                &backfolding_ex_P.tInput[0], &backfolding_ex_P.firCoeffs32[0],
                -0.0341458619F, 0.250496089F, 320U, &rtb_DiscreteFIRFilter[0], 32U);

    /* Output: '<Root>/Out1' */
    memcpy(&backfolding_ex_Y.Out1[0], &rtb_DiscreteFIRFilter[0], 320U * sizeof
        (real32_T));
}

```

The generated code contained the temporary variable `rtb_DiscreteFIRFilter` and a `memcpy` function for copying data from `rtb_DiscreteFIRFilter` to the destination variable `backfolding_ex_Y.Out1`.

In R2017b, the code generator produces this code:

```
/* Model step function */
void backfolding_ex_step(void)
{
    /* DiscreteFir: '<Root>/Discrete FIR Filter' incorporates:
     * Constant: '<Root>/Constant'
     * Outport: '<Root>/Out1'
     */
    arm_fir_f32(&backfolding_ex_DW.S, &backfolding_ex_DW.pState[0],
               &backfolding_ex_P.tInput[0], &backfolding_ex_P.firCoeffs32[0],
               -0.0341458619F, 0.250496089F, 320U, &backfolding_ex_Y.Out1[0], 32U);
}
```

The generated code does not contain the extra temporary variable `rtb_DiscreteFIRFilter` or the data copy to this temporary variable. Instead, the function writes directly to destination variable `backfolding_ex_Y.Out1`.

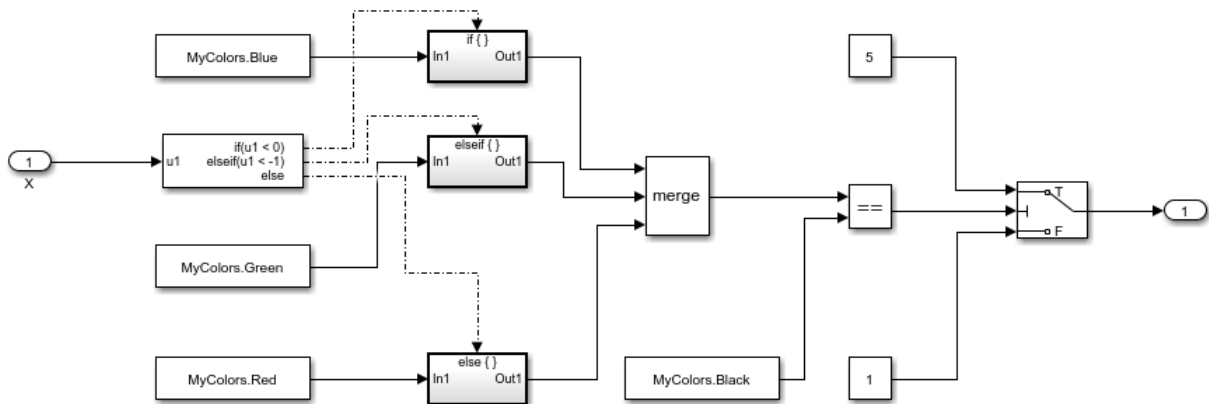
Code Replacement: Apply MustHaveZeroNetBias and SlopesMustBeTheSame properties for fixed-point operator code replacement

In R2017b, operation of the `MustHaveZeroNetBias` and `SlopesMustBeTheSame` properties for fixed-point operator code replacement is clarified. For information about applying these properties for code replacement matches of add, subtract, multiply, divide, cast, and shift operators, see `Fixed-Point Operator Code Replacement` and `setTf1COperationEntryParameters`.

Enumerated Data Types Optimization: Improve the efficiency of the generated code for enumerated data types

In R2017b, the generated code for enumerated data types contains optimizations that in previous releases, applied only to scalar data. These optimizations include reducing the storage size of variables, constant folding, redundant assignment elimination, and control flow simplification. The optimizations reduce ROM and RAM consumption and increase execution speed.

For example, the model `Enum_example` contains a combination of `Enumerated Constant` blocks and control flow subsystems.



In R2017a, the code generator produced this code:

```
void Enum_example_step(void)
{
    MyColors tmp;
    if (Enum_example_U.X < 0.0) {
        tmp = Blue;
    } else if (Enum_example_U.X < -1.0) {
        tmp = Green;
    } else {
        tmp = Red;
    }

    if (tmp == Black) {
        Enum_example_Y.Out1 = 5.0;
    } else {
        Enum_example_Y.Out1 = 1.0;
    }
}

void Enum_example_initialize(void)
{
    rtmSetErrorStatus(Enum_example_M, (NULL));
    Enum_example_U.X = 0.0;
    Enum_example_Y.Out1 = 0.0;
}

void Enum_example_terminate(void)
{
}

```

The generated code contains control flow constructs. Enumerated constants are variables.

In R2017b, the code generator produces this code:

```
void Enum_example_step(void)
{
    Enum_example_Y.Out1 = 1.0;
}

```

The code generator evaluates the enumerated constants and determines that `temp` can never equal `Black`. The generated code does not contain the control flow constructs. `Enum_example_Y.Out1` equals 1.

Verification

Multiple Processor SIL/PIL Testing: Perform SIL or PIL component tests on different processors simultaneously

If you have a model with Model block components that are configured to generate code for different target environments, you can test the generated code components simultaneously by running simulations with the Model blocks in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

For more information, see:

- “Simplified Build Folder Layout: Generate code for different hardware settings in separate folders”
- Manage Build Process Folders (Simulink Coder)
- Model Block SIL/PIL Limitations

SIL Simulation: Simplified configuration of hardware implementation settings

The default **Hardware Implementation** parameter settings support SIL simulations on a development computer that uses a 64-bit Windows® operating system. Previously, you had to set the `ProdLongLongMode` configuration parameter to `on`.

On this computer, you can run SIL simulations to test generated code for many 32-bit devices without enabling the `PortableWordSizes` configuration parameter.

For more information, see [Configure Hardware Implementation Settings](#).

SIL/PIL Configuration: Parent model code coverage, execution-time profiling, and SIL debugging settings apply to Model blocks with Top-model code interface

If a model contains Model blocks with the **Simulation mode** block parameter set to `Software-in-the-loop (SIL)` or `Processor-in-the-loop (PIL)` and the **Code interface** block parameter set to `Top model`, these parameters of the parent model override the corresponding parameters of the models referenced by the Model blocks:

- **Code coverage for this model**
- **Code coverage for referenced models**
- **Measure task execution time.** Disabling task profiling for the top model also disables function profiling for all referenced models.
- **Workspace variable**
- **Save options**
- **Enable source-level debugging for SIL**

Previously, if the referenced model parameter settings did not match the parent model settings, an error occurred.

For more information, see:

- Code Coverage
- Code Execution Profiling with SIL and PIL
- Debug Generated Code During SIL Simulation

Hardware Implementation Settings: SIL checks relaxed for data type sizes and byte ordering

A software-in-the-loop (SIL) simulation checks the **Hardware Implementation** pane settings with respect to your development computer. In R2017b, when **Code Generation > Verification > Enable portable word sizes** is *not* selected, SIL simulation is possible when the values of these parameters on the **Hardware Implementation** pane are less than or equal to the values for your development computer:

- **Number of bits: native**
- **Number of bits: pointer**
- **Number of bits: size_t**
- **Number of bits: ptrdiff_t**

SIL simulation is also possible when **Byte ordering** is set to **Big Endian** and **Code Generation > Interface > Support: non-finite numbers** is not selected. If you use custom code with a specific endianness, SIL and PIL simulation results can differ.

Previously, if there were mismatches between the parameter values on the **Hardware Implementation** pane and your development computer values, the SIL simulation produced errors.

For more information, see [Configure Hardware Implementation Settings](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.12

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

SIL and PIL execution improvements for MATLAB Coder

This table lists software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution improvements.

Feature	R2017a	Previous releases
Interface type: Global data	Supported	Not supported
Size: Dynamic variable-size arrays	Supported	Not supported

For more information, see [SIL/PIL Execution Support and Limitations](#).

Verification of PIL target connectivity configuration

The `piltest` function provides additional tests for verifying your custom processor-in-the-loop (PIL) target connectivity configuration. You can specify tests by using the 'Testpoint' argument.

'Testpoint' Value	Description
'verifyPILConfig'	<p>For a given set of input values, the function:</p> <ul style="list-style-type: none"> • Runs a MATLAB function on your development computer. • Performs PIL executions of generated MATLAB code on your target hardware with <code>config.TargetLang</code> settings 'C' and 'C++'. <p>The function compares results from the MATLAB function run and the PIL executions. If the function detects differences, it produces an error.</p>

For more information, see [Create PIL Target Connectivity Configuration](#).

Code Replacement for MATLAB Coder: Create code replacement library entries for target implementations that require data alignment

As of R2017a, you can take advantage of function implementations that require aligned data to optimize application performance when using MATLAB Coder.

For more information, see [Data Alignment for Code Replacement](#).

Model Architecture and Design

AUTOSAR arxml File Import: Flexibly model imported periodic, asynchronous, and initialization runnables

The AUTOSAR arxml importer now supports AUTOSAR modeling styles for which Simulink modeling support was added in R2016b. For example, you can

- Import periodic and asynchronous runnables in a JMAAB type beta modeling configuration. The modeling style is described in [Add Top-Level Asynchronous Trigger to Periodic Rate-Based System](#).
- Import an initialize runnable, which the importer now represents with a Simulink Initialize Function block.

To import an AUTOSAR software component with multiple runnable entities into a Simulink model, you use the arxml importer method `createComponentAsModel`. As part of improved runnable modeling, the `createComponentAsModel` method now provides the property `ModelPeriodicRunnablesAs`, which replaces the property `CreateInternalBehavior`. At model creation time, set `ModelPeriodicRunnablesAs` to one of these values:

- `AtomicSubsystem` (default) — Import AUTOSAR periodic runnables found in arxml files. Model periodic runnables as atomic subsystems with periodic rates in a rate-based model. If conditions prevent use of atomic subsystems, the importer throws an error.
- `FunctionCallSubsystem` — Model periodic runnables as function-call subsystems with periodic rates.
- `Auto` — Attempt to model periodic runnables as atomic subsystems. If conditions prevent use of atomic subsystems, model periodic runnables as function-call subsystems.

Set `ModelPeriodicRunnablesAs` to `AtomicSubsystem` unless your design requires use of function-call subsystems. The following call directs the importer to import a multirunnable AUTOSAR software component and map it into a new rate-based model.

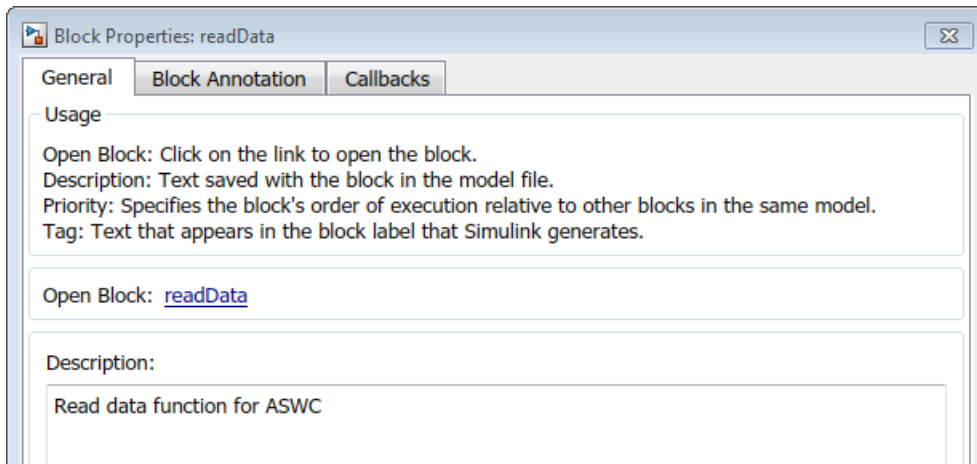
```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj, '/pkg/swc/ASWC', 'ModelPeriodicRunnablesAs', 'AtomicSubsystem')
```

For more information, see [Import AUTOSAR Software Component and Model AUTOSAR Software Components](#).

AUTOSAR DESC elements populate Simulink Description fields

Importing AUTOSAR DESC information associated with an AUTOSAR identifiable element now populates the **Description** property in the corresponding Simulink element or data object. Correspondingly, exporting a Simulink element or data object **Description** property now populates the DESC information in the corresponding AUTOSAR element. Previously, Embedded Coder preserved AUTOSAR DESC information across arxml round-trips but did not leverage the information to add a readable text description to the Simulink model.

For example, suppose that you open the example model `rtwdemo_autosar_swc_slfcns` and add a description to the Simulink Function block `read_data`. Use the block properties dialog.



When you export a `rxml` for the model, the generated runnable description contains the Simulink description text.

```
<RUNNABLE-ENTITY UUID="...">
  <SHORT-NAME>Runnable_readData</SHORT-NAME>
  <DESC>
    <L-2 L="FOR-ALL">Read data function for ASWC</L-2>
  </DESC>
  ...
  <SYMBOL>readData</SYMBOL>
</RUNNABLE-ENTITY>
```

Note This support is available to R2015b, R2016a, and R2016b Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.8 or later
- R2016a Embedded Coder Support Package for AUTOSAR Standard, Version 16.1.5 or later
- R2016b Embedded Coder Support Package for AUTOSAR Standard, Version 16.2.2 or later

External mode code generation for a model containing inline variant blocks

In R2017a, for a model containing Variant Source or Variant Sink blocks, you can generate code for the external mode data interface. In the block parameters dialog box, clear the **Analyze all choices during update diagram and generate preprocessor conditionals** parameter. For more information on external mode, see Set Up and Use Host/Target Communication Channel.

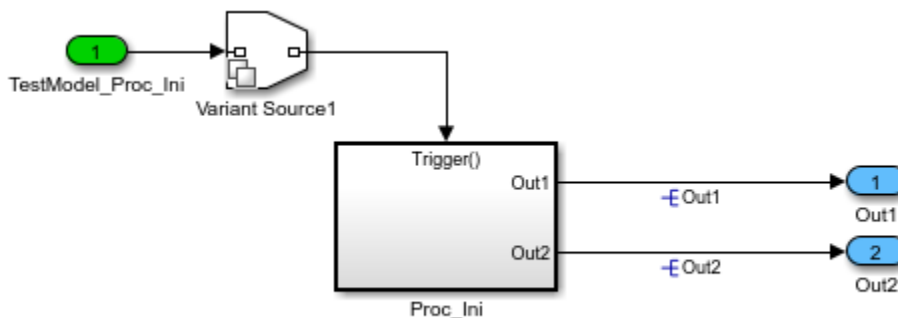
Code generation support for Variant Subsystems containing global signals

In R2017a, you can generate code for a model containing a Variant Subsystem with global signals inside it. You declare signals as global by assigning them a storage class other than Auto. See Storage Classes for Signals Used with Model Blocks.

Preprocessor conditionals guard content inside and outside of function-call site

In R2016b, for a model that contained a conditional function-call subsystem, preprocessor conditionals guarded only the content inside the function. In R2017a, preprocessor conditionals guard the content and the function-call site.

For example, in the model `func_call_guards`, a Variant Source block connects to the function-call subsystem `Proc_Ini`.



In R2016b, the code generator produced this code:

```
/* Model step function */
void TestModel_Proc_Ini(void)
{
    /* RootInportFunctionCallGenerator: '<Root>' */
    /* RootFcnCall_InsertedFor_TestModel_Proc_Ini_at_outport_1' */
    #if W == 1

        /* Outputs for Function Call SubSystem: '<Root>/Proc_Ini' */
        /* SignalConversion: '<S1>/OutputBufferForOut1' */
        Out1 = 1.0F;

        /* SignalConversion: '<S1>/OutputBufferForOut2' */
        Out2 = 1.0F;

        /* End of Outputs for SubSystem: '<Root>/Proc_Ini' */
    #endif
        /* W == 1 */

        /* End of Outputs for RootInportFunctionCallGenerator: '<Root>' */
        /* RootFcnCall_InsertedFor_TestModel_Proc_Ini_at_outport_1' */
    }
}
```

The preprocessor conditionals guarded the content inside the function `Proc_Ini`.

In R2017a, the code generator produces this code:

```
/* Model step function */
#if W == 1

void TestModel_Proc_Ini(void)
{
```

```
/* RootInportFunctionCallGenerator: '<Root>
/RootFcnCall_InsertedFor_TestModel_Proc_Ini_at_outport_1' */
#if W == 1

/* Outputs for Function Call SubSystem: '<Root>/Proc_Ini' */
/* SignalConversion: '<S2>/OutportBufferForOut1' */
Out1 = 1.0F;

/* SignalConversion: '<S2>/OutportBufferForOut2' */
Out2 = 1.0F;

/* End of Outputs for SubSystem: '<Root>/Proc_Ini' */
#endif                                     /* W == 1 */

/* End of Outputs for RootInportFunctionCallGenerator: '<Root>
/RootFcnCall_InsertedFor_TestModel_Proc_Ini_at_outport_1' */
}

#endif                                     /* W == 1 */
```

The preprocessor conditionals guard the content inside and outside of the function Proc_Ini.

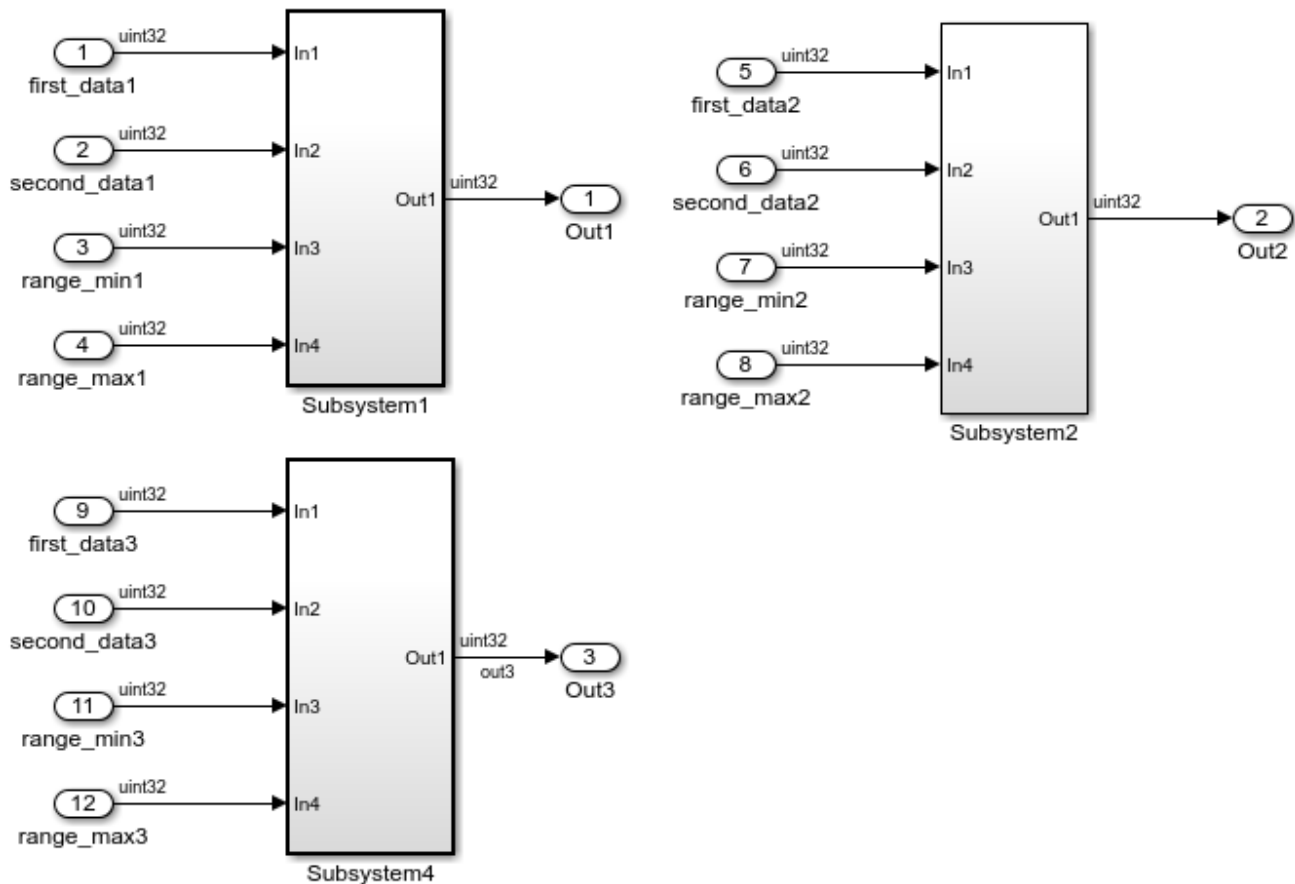
Data, Function, and File Definition

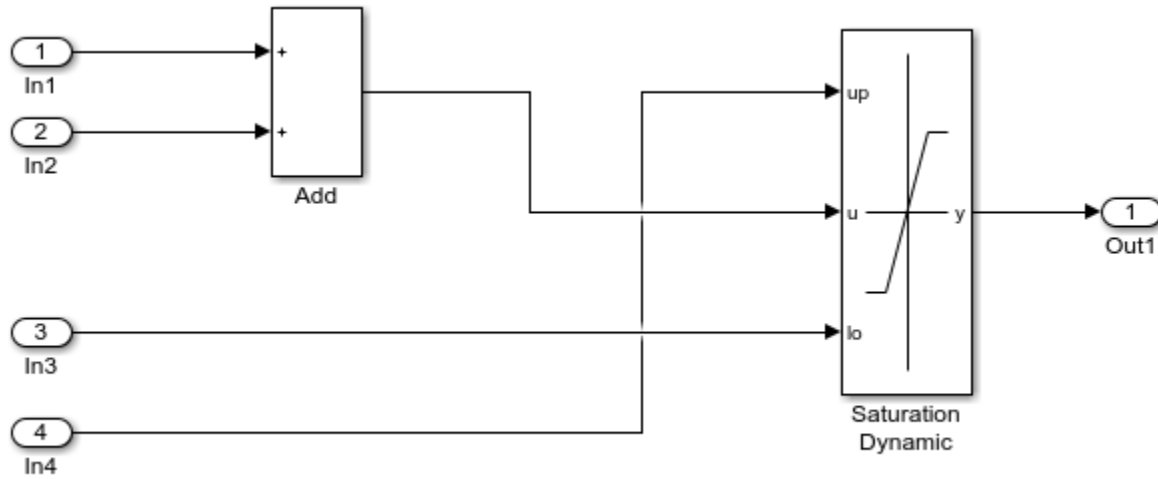
Function Interface: Return nonvoid type for scalar output of reusable functions

In R2016b, reusable functions had a return type of void. In R2017a, reusable functions can return a nonvoid type. The code generator can return a nonvoid type if the reusable function has one output parameter that is a scalar and in the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, the **Pass reusable subsystem outputs as** parameter is set to Individual arguments.

Returning a nonvoid type conserves RAM consumption because the generated code does not contain a global variable to hold the output parameter value. There are also minor improvements in ROM consumption because the function call site and the function body are smaller.

For example, the model reusable_sub contains four reusable subsystems. Subsystem2 contains Subsystem3. Subsystem1, Subsystem3, and Subsystem4 contain the blocks shown in this diagram following the model. The subsystem output is a scalar.





In R2016b, the reusable subsystem function contained this code:

```
void reusable_sub_Subsystem1(uint32_T rtu_In1, uint32_T rtu_In2, uint32_T
    rtu_In3, uint32_T rtu_In4, uint32_T *rty_Out1)
{
    uint32_T rtb_Add;
    rtb_Add = rtu_In1 + rtu_In2;
    if (rtb_Add > rtu_In4) {
        *rty_Out1 = rtu_In4;
    } else if (rtb_Add < rtu_In3) {
        *rty_Out1 = rtu_In3;
    } else {
        *rty_Out1 = rtb_Add;
    }
}

void reusable_sub_step(RT_MODEL_reusable_sub *const reusable_sub_M,
    ExternalInputs_reusable_sub *reusable_sub_U, ExternalOutputs_reusable_sub
    *reusable_sub_Y)
{
    reusable_sub_Subsystem1(reusable_sub_U->first_data1,
        reusable_sub_U->second_data1, reusable_sub_U->range_min1,
        reusable_sub_U->range_max1, &reusable_sub_Y->Out1);
    reusable_sub_Subsystem1(reusable_sub_U->first_data2,
        reusable_sub_U->second_data2, reusable_sub_U->range_min2,
        reusable_sub_U->range_max2, &reusable_sub_Y->Out2);
    reusable_sub_Subsystem1(reusable_sub_U->first_data3,
        reusable_sub_U->second_data3, reusable_sub_U->range_min3,
        reusable_sub_U->range_max3, &reusable_sub_Y->Out3);
    UNUSED_PARAMETER(reusable_sub_M);
}
```

In R2016b, the generated code contained the global variable `rty_Out1` to hold the output. `rty_Out1` was passed to `reusable_sub_Subsystem1`.

In R2017a, the `reusable_sub.c` file contains this code:

```

uint32_T reusable_sub_Subsystem1(uint32_T rtu_In1, uint32_T rtu_In2, uint32_T
    rtu_In3, uint32_T rtu_In4)
{
    uint32_T rty_Out1_0;
    rty_Out1_0 = rtu_In1 + rtu_In2;
    if (rty_Out1_0 > rtu_In4) {
        rty_Out1_0 = rtu_In4;
    } else {
        if (rty_Out1_0 < rtu_In3) {
            rty_Out1_0 = rtu_In3;
        }
    }

    return rty_Out1_0;
}

void reusable_sub_step(RT_MODEL_reusable_sub *const reusable_sub_M,
    ExternalInputs_reusable_sub *reusable_sub_U, ExternalOutputs_reusable_sub
    *reusable_sub_Y)
{
    reusable_sub_Y->Out1 = (uint32_T) reusable_sub_Subsystem1
        (reusable_sub_U->first_data1, reusable_sub_U->second_data1,
        reusable_sub_U->range_min1, reusable_sub_U->range_max1);
    reusable_sub_Y->Out2 = (uint32_T) reusable_sub_Subsystem1
        (reusable_sub_U->first_data2, reusable_sub_U->second_data2,
        reusable_sub_U->range_min2, reusable_sub_U->range_max2);
    reusable_sub_Y->Out3 = (uint32_T) reusable_sub_Subsystem1
        (reusable_sub_U->first_data3, reusable_sub_U->second_data3,
        reusable_sub_U->range_min3, reusable_sub_U->range_max3);
    UNUSED_PARAMETER(reusable_sub_M);
}

```

The generated code does not contain a global variable to hold output. Instead, the function returns the local variable `rty_Out1_0`.

Utility to generate Simulink representations of struct and enum types defined by external C code

Before R2017a, to generate code that used `struct` and `enum` types defined by your external code, you had to manually create the corresponding definitions in Simulink (for example, `Simulink.Bus` objects).

In R2017a, you can generate these corresponding Simulink definitions by using a programmatic utility. The utility parses your external C code for `struct` and `enum` type definitions. For more information, see [Utility to generate Simulink representations of custom data types defined by external C code \(Simulink\)](#).

Code Generation

Cross-Release Code Integration: Reuse model reference code generated from previous releases

In R2017a, you can integrate exported component code that uses the model reference code interface. Previously, the cross-release integration workflow supported only component code that used the standalone code interface. For more information, see Cross-Release Code Integration.

Compatibility Considerations

For the `crossReleaseImport` function, the value for the `CodeLocation` argument specifies the path to an anchor folder that contains the relocated model code folder. Previously, the `CodeLocation` value specified the path to the relocated model code folder.

For R2017a, if you relocate generated model code, use an anchor folder and maintain the original code folder names and structure.

Model Component	Code Interface	Original Code Location	New Code Location
Top model	Standalone	<code>codeGenFolder/modelName_ert_rtw</code>	<code>anchorFolder/modelName_ert_rtw</code>
Referenced model	Model reference	<code>codeGenFolder/slprj/ert/refModelName</code>	<code>anchorFolder/slprj/ert/refModelName</code>
Subsystem	Standalone	<code>codeGenFolder/subSysName_ert_rtw</code>	<code>anchorFolder/subSysName_ert_rtw</code>

Code Replacement for Cast and Multiply Operations: Detect overflow and rounding mode equivalence for increased matches and code efficiency

As of R2017a, the code replacement software support for detecting overflow and rounding mode equivalence is enhanced for cast operations and multiply operations:

- Cast operations — When an operation does not overflow, based on input and output data types, a match occurs for code replacement table entries with the saturation mode set to **Wrap on Overflow** (RTW_WRAP_ON_OVERFLOW). Similarly, if the code replacement software detects equivalent rounding modes, a match occurs.
- Multiplication operations — The detection of overflow and rounding modes equivalence is enhanced to support a mixture of fixed-point and floating-point types.

For more information, see Develop a Code Replacement Library.

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**
- **Memory Information** (if you select parameter **Code Generation > Report > Static code metrics**)
- **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.

Code Interface Report: Includes entry-point function for code generated from Reset Function block

Starting in R2017a, the Code Interface Report section of the Code Generation Report includes entry-point function information for code generated from Reset Function blocks. For more information, see Generate Code That Responds to Initialize, Reset, and Terminate Events and Analyze the Generated Code Interface.

Shared utility memory section associated with subfunctions

Previously, you could not predict which memory section was associated with subfunctions in the generated code. Simulink Coder generates these subfunctions for intrinsic math utilities, Stateflow graphical functions, and MATLAB subfunctions. The possible associations included:

- The **Shared utility** memory section that you specify at the model level.
- The **Execution** memory section that you specify at the model level.
- The **Execution** memory section that you specify for one of the subsystems.

In R2017a, the memory section associated with these subfunctions is always the **Shared utility** memory section that you specify at the model level.

Inline traceability for generated code

Model-to-code and code-to-model navigation are enhanced for Embedded Coder in R2017a. Inline traceability is now fully supported:

- For MATLAB functions
- For Simulink blocks, with the exception of From Workspace and From File blocks

For more information on bidirectional traceability, see What Is Code Tracing?.

Clear file section content from TLC file

The ability to reset a file section buffer in TLC was removed in R2015a. In R2017a, you can use the TLC function `LibClearFileSectionContents` to clear a file section buffer so that you can reset it. This function can be applied to the following sections:

- Banner
- Includes
- ModelTypesTypedefs
- Defines
- ModelTypesDefines
- IntrinsicTypes
- PrimitiveTypedefs
- UserTop
- Typedefs
- Enums
- Definitions
- ExternData
- ExternFcns
- FcnPrototypes
- Declarations
- Functions
- CompilerErrors
- CompilerWarnings
- Documentation
- UserBottom

Identifier case control with token decorators and custom text token \$U

\$U Token for Specifying Text in Generated Identifiers

On the **Code Generation > Symbols** pane, you can use the \$U token to specify text to include in the generated identifiers. All the identifiers on the **Symbols** pane accept this new token.

You set the value of \$U by specifying a character vector for the **Custom token text** parameter. The **Custom token text** parameter is on the **All Parameters** tab in the Configuration Parameters dialog box.

For more information, see Identifier Format Control and Custom token text.

Case Control with Token Decorators

On the **Code Generation > Symbols** pane, you can use new token decorators to control the case of generated identifiers. For example, use this technique to apply camel case style.

Place a decorator immediately after a token and enclose the decorator in square brackets []. For example, you can set **Global variables** to `$R[uL]$N$M`, which capitalizes the first letter of the model name and forces the remaining characters in the model name to lowercase.

For more information, see Control Case with Token Decorators.

Name change for AUTOSAR local temporary variables

Previously, for an AUTOSAR model, the name for local temporary variables in the generated code was `tmp`. In R2017a, the name is `tmp` plus an identifier associated with the data access mode of the variable, such as `IRead` or `IWrite`. For example, in R2017a, the name of a local temporary variable with an `ImplicitReceive` data access mode is `tmpIRead`.

Additional checks against MISRA C:2012 guidelines in Code Generation Advisor

In R2017a, when the Code Generation Advisor checks your model against the MISRA C:2012 guidelines objective, it executes these additional checks:

- Check for blocks not recommended for C/C++ production code deployment
- Check for unsupported block names
- Check usage of Assignment blocks
- Check for bitwise operations on signed integers
- Check for recursive function calls
- Check for equality and inequality operations on floating-point values
- Check for switch case expressions without a default case

Also for the MISRA C:2012 guidelines objective, the Code Generation Advisor considers these additional parameters:

- Shared code placement (Simulink Coder) (`UtilityFuncGeneration`)
- System-generated identifiers (Simulink Coder) (`InternalIdentifier`)
- Use dynamic memory allocation for model initialization (Simulink Coder) (`GenerateAllocFcn`)

Deployment

TI Code Composer Studio (CCS): Generate projects for CCS versions 5 and 6 with Embedded Coder Target for TI C2000

When you build Simulink models for TI C2000 targets with CCS v5 or v6 toolchains, the Code Composer Studio project is also generated. You can use this project for debugging the generated code.

Customize generated makefiles for S-Functions

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

Release notes and workflow overview documentation added to AUTOSAR support package

R2017a adds release notes and workflow overview documentation to the Embedded Coder Support Package for AUTOSAR Standard. The release notes describe AUTOSAR support changes from the current release back through R2014b. Other help topics provide an overview of AUTOSAR workflows, with links to the main AUTOSAR help.

After you install the support package, restart MATLAB, open help (for example, with the MATLAB `doc` command), and go to the Hardware Support section. To access support package help and release notes, click the support package name.

SPI and I2C blocks added to TI C2000 support package

This table lists the support for the new blocks.

Block	Usage
SPI Receive	Receive data via serial peripheral interface (SPI) on target.
SPI Transmit	Transmit data via serial peripheral interface (SPI) to host.
I2C Receive	Configure inter-integrated circuit (I2C) module to receive data from I2C bus.

Block	Usage
I2C Transmit	Configure inter-integrated circuit (I2C) module to transmit data to I2C bus.

CCS v3.3 IDE automation support for TI C2000 has been removed

The support for TI C2000 with `idelink_ert.tlc` as system target file has been removed. You can still use the TI C2000 support by using the `ert.tlc` as the system target file.

Real-time multitasking profiling for TI C2000

You can use real-time execution profiling to verify whether generated code meets the real-time performance requirements.

TCP and UDP blocks added to STMicroelectronics STM32F746G-Discovery board

This table lists the support for these new blocks.

Block	Usage
TCP Receive	Receive TCP packets from another TCP host on TCP/IP network
TCP Send	Send TCP packets to another TCP host on TCP/IP network
UDP Receive	Receive UDP packets from another UDP host
UDP Send	Send UDP packets to another UDP host

MATLAB Coder PIL with STMicroelectronics STM32F4-Discovery Board

In R2017a, you can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware using a MATLAB Coder workflow with an Embedded Coder license. By using PIL with hardware, you can generate customized code for your hardware more effectively by profiling speed and algorithm performance. You have the option of using the command-line workflow or the MATLAB Coder app to configure your target hardware for PIL executions.

To use this feature, you must have MATLAB Coder and the support package installed.

This example shows how to use a PIL execution to verify generated code.

- 1 In the command window, select the hardware for PIL execution.

```
hw = coder.hardware('STM32F4-Discovery');
```

- 2 Add the hardware to the MATLAB Coder configuration object.

```
cfg = coder.config('lib', 'ecoder', true);
cfg.VerificationMode = 'PIL';
cfg.Hardware = hw;
```

- 3 As the stack space in the target hardware is limited, set the maximum stack space that the generated code uses.

```
cfg.StackUsageMax = 512;
```

- 4 Generate PIL code for a function, computeFFT.

```
codegen -config cfg computeFFT -args {inp}
```

Here, `computeFFT` is a user-defined function. The `inp` parameter declares the data type and size for input arguments to MATLAB function `computeFFT`. The `codegen` command generates code into following folders:

- `codegen\lib\computeFFT`: Standalone code for `computeFFT`.
- `codegen\lib\computeFFT\pil`: PIL interface code for `computeFFT`.

Also, this step creates `computeFFT_pil` PIL MEX function in the current folder. This function allows you to test the MATLAB code and the PIL MEX function and compare the results between both.

- 5 Run the PIL MEX function to compare its behavior to that of the original MATLAB function and to check for defects.

```
u1 = uint16(zeros(1,16));  
y = computeFFT_pil(u1);
```

Terminate PIL execution with the following command.

```
clear computeFFT_pil;
```

Using the MATLAB Coder app workflow:

- 1 Configure the build type and hardware board. On the **Generate Code** page, in the **Generate** dialog box:

- Set the **Build Type** to `Static Library`.
- Clear the **Generate code only** check box.
- Set the **Hardware Board** to `STM32F4-Discovery`.

- 2 You can modify the settings for your board. To modify the settings, click **Settings > All Settings**.

Specify the maximum stack space required by the generated code in the **Memory > Stack usage max** parameter. The stack space in the target hardware is limited, and a default value of 20000 is beyond the stack size available in the target hardware. A value of 512 is recommended. You can specify the stack size based on the requirement of your application.

- 3 Click **Hardware**.
- 4 To generate the library, click **Generate**.
- 5 Set up for PIL execution. Click **Verify Code** to open the **Verify Code** dialog box.

Because the hardware board is not MATLAB Host Computer, the **Verify Code** dialog box is configured for PIL execution.

In the **Verify Code** dialog box:

- Enter the name of the test file to use for PIL execution.
- Select **Generated code**.

- 6 To start the PIL execution, click **Run Generated Code**.
- 7 To stop the PIL execution, click **Stop**.

For more information, on how to compile your code using the MATLAB Coder app, see [Opening the MATLAB Coder™ App \(MATLAB Coder\)](#).

For more information, on how to use the Embedded Coder Support Package for STMicroelectronics Discovery Boards for Processor-in-the-Loop (PIL) verification of MATLAB functions, see [Processor in the Loop Verification of MATLAB Functions \(Embedded Coder Support Package for STMicroelectronics Discovery Boards\)](#).

External Mode and PIL supported over TCP/IP by STMicroelectronics STM32F746G-Discovery board

The STMicroelectronics STM32F746G-Discovery™ board supports PIL and external mode over TCP/IP.

Install the Embedded Coder Support Package for STMicroelectronics Discovery Boards to use this support.

To install or update this support package, perform the steps described in [Install Support for STMicroelectronics Discovery Boards \(Embedded Coder Support Package for STMicroelectronics Discovery Boards\)](#).

For more information, see [Embedded Coder Support Package for STMicroelectronics Discovery Boards](#).

Linux Support: Connect to ARM Cortex-M processor on Linux platform

You can use the Embedded Coder Support Package for ARM Cortex-M Processors on the Linux host platform to generate and build ARM Cortex-M optimized code from models.

Note You cannot load and run code generated from a model on the Linux host platform using ARM Cortex-M QEMU emulator.

ARM Cortex-R optimized code

Use the Embedded Coder Support Package for ARM Cortex-R Processors to build optimized executables with automatic code replacement from the Hercules™ Safety MCU Cortex™-R4 CMSIS DSP Library.

Develop a Target for ARM Cortex-R processors

The Embedded Coder Support Package for ARM Cortex-R Processors supports the development of user specified Targets. Targets include deployment, scheduling, processor-in-the-loop, external mode, code replacement, and profiler features.

Support for Wind River VxWorks RTOS will be removed

Embedded Coder support for Wind River VxWorks RTOS will be removed in a future release. You will still be able to use Embedded Coder for VxWorks RTOS, but will need to manually integrate the generated code with hand written scheduler and drivers.

Performance

Data Copy Reduction: Generate fewer data copies and use less RAM for buses, data stores, and model blocks

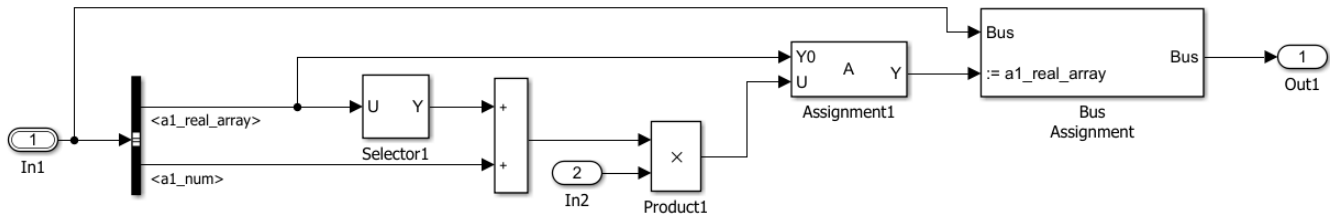
In R2017a, the generated code contains less temporary variables and associated data copies for modeling patterns involving Bus Assignment, Data Store Read, Data Store Write, and Model blocks. These optimizations conserve RAM usage and improve code execution speed. The following examples highlight these improvements:

- “Data copy reduction for Bus Assignment block” on page 12-19
- “Data copy reduction for Data Store Read and Data Store Write blocks” on page 12-21
- “More efficient code for Model blocks” on page 12-23

Data copy reduction for Bus Assignment block

Previously, for a model that contained a Bus Assignment block, there was an extra temporary variable and associated data copy in the generated code. In R2017a, the code generator can remove this data copy. This optimization increases code execution speed and conserves RAM consumption.

For example, in `bus_assignoptim`, a bus signal containing six elements feeds into a Bus Assignment block and a Bus Selector block. The Bus Assignment block assigns new values to the bus element `a1_real_array`. This bus signal feeds into `Out1`.



In R2016b, the code generator produced this code in the `bus_assignoptim_step` function:

```
/* Model step function */
void bus_assignoptim_step(void)
{
    real_T rtb_Assignment[36];
    int32_T i;

    /* Assignment: '<Root>/Assignment' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Product: '<Root>/Product'
     * Selector: '<Root>/Selector'
     * Sum: '<Root>/Sum1'
     */
    for (i = 0; i < 36; i++) {
        rtb_Assignment[i] = bus_assignoptim_U.In1.a1_real_array[i];
    }

    for (i = 0; i < 2; i++) {
```

```

    rtb_Assignment[(int32_T)(i + 22)] = (bus_assignoptim_U.In1.a1_real_array
      [(int32_T)(i + 22)] + bus_assignoptim_U.In1.a1_num) *
      bus_assignoptim_U.In2;
  }

  /* End of Assignment: '<Root>/Assignment' */

  /* BusAssignment: '<Root>/Bus Assignment' incorporates:
   * Inport: '<Root>/In1'
   */
  bus_assignoptim_Y.Out = bus_assignoptim_U.In1;
  for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.a1_real_array[i] = rtb_Assignment[i];
  }

  /* End of BusAssignment: '<Root>/Bus Assignment' */
}

```

The generated code contains the temporary array `rtb_Assignment1` for holding data before this data is assigned to `bus_assignoptim_Y.Out2.dbl_real_array`.

In R2017a, the `bus_assignoptim_step` function contains this code:

```

/* Model step function */
void bus_assignoptim_step(void)
{
  int32_T i;

  /* SignalConversion: '<Root>/TmpBusAssignmentBufferAtBus...'
   * Inport: '<Root>/In1'
   */
  bus_assignoptim_Y.Out = bus_assignoptim_U.In1;

  /* Assignment: '<Root>/Assignment' incorporates:
   * Inport: '<Root>/In1'
   * Inport: '<Root>/In2'
   * Product: '<Root>/Product'
   * Selector: '<Root>/Selector'
   * Sum: '<Root>/Sum1'
   */
  for (i = 0; i < 36; i++) {
    bus_assignoptim_Y.Out.a1_real_array[i] =
      bus_assignoptim_U.In1.a1_real_array[i];
  }

  for (i = 0; i < 2; i++) {
    bus_assignoptim_Y.Out.a1_real_array[(int32_T)(i + 22)] =
      (bus_assignoptim_U.In1.a1_real_array[(int32_T)(i + 22)] +
       bus_assignoptim_U.In1.a1_num) * bus_assignoptim_U.In2;
  }

  /* End of Assignment: '<Root>/Assignment' */
}

```

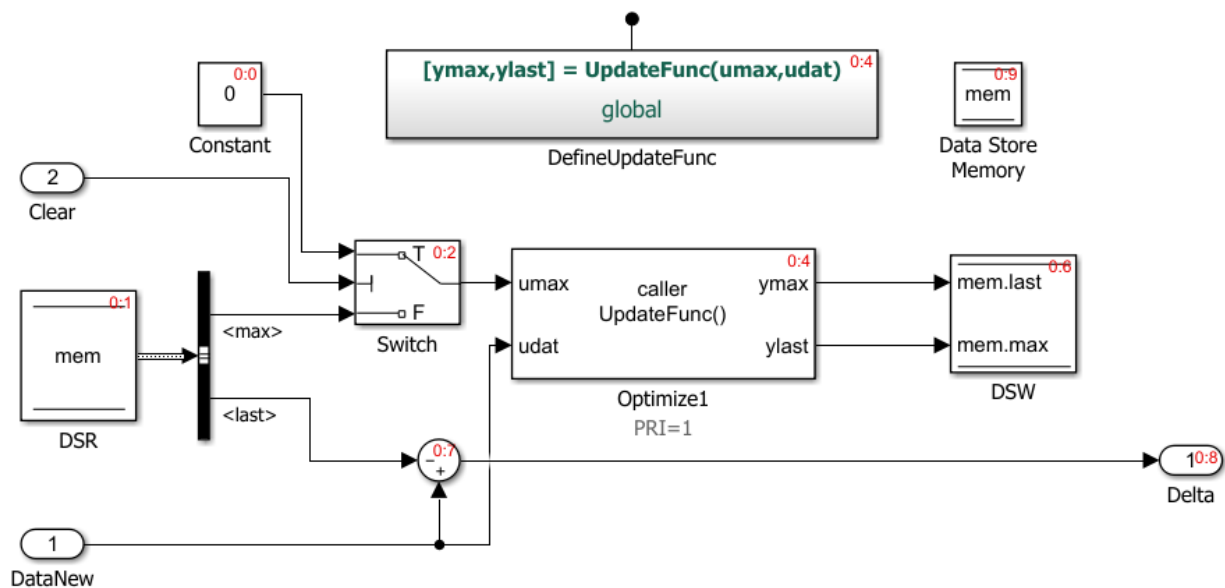
The generated code does not contain the temporary array `rtb_Assignment1` for holding data. The generated code directly assigns the data to `bus_assignoptim_Y.Out2.dbl_real_array`.

Note You can disable this optimization by clearing the **Perform inplace updates for Bus Assignment blocks** parameter. In the Configuration Parameters dialog box, this parameter is on the **All Parameters** tab.

Data copy reduction for Data Store Read and Data Store Write blocks

In R2016b, the generated code contained an extra buffer when reading from a Data Store Read block or when writing to a Data Store Write block. In R2017a, the code generator can eliminate this extra data copy. This optimization conserves RAM consumption and improves code execution speed.

For example, in the model `rtwdemo_optimizedatastorebuffers`, the Function caller `UpdateFunc` calls the Simulink Function `DefineUpdateFunc`. The Data Store Read block `DSR` reads from `mem`. The Data Store Write block `DSW` writes to `mem`.



In R2016b, the code generator produced this code:

```

/* Model step function */
void rtwdemo_optimizedatastorebuffers_step(void)
{
    real_T rtb_DSR_last;
    real_T rtb_Optimize1_o1;
    real_T rtb_Optimize1_o2;

    /* DataStoreRead: '<Root>/DSR' */
    rtb_DSR_last = mem.last;

    /* Switch: '<Root>/Switch' incorporates:
     * Constant: '<Root>/Constant'
     * DataStoreRead: '<Root>/DSR'
     * Inport: '<Root>/Clear'
     */
    if (rtU.Clear) {
        rtb_Optimize1_o1 = 0.0;
    }
}

```

```

} else {
    rtb_Optimize1_o1 = mem.max;
}

/* End of Switch: '<Root>/Switch' */

/* FunctionCaller: '<Root>/Optimize1' incorporates:
 * Inport: '<Root>/DataNew'
 */
UpdateFunc(rtb_Optimize1_o1, rtU.DataNew, &rtb_Optimize1_o1, &rtb_Optimize1_o2);

/* DataStoreWrite: '<Root>/DSW' */
mem.last = rtb_Optimize1_o1;
mem.max = rtb_Optimize1_o2;

/* Outport: '<Root>/Delta' incorporates:
 * Inport: '<Root>/DataNew'
 * Sum: '<Root>/Sum'
 */
rtY.Delta = rtU.DataNew - rtb_DSR_last;
}

```

The generated code contained data copies for the Data Store Read and Data Store Write blocks, respectively.

In R2017a, the code generator produces this code:

```

/* Model step function */
void rtwdemo_optimizedatastorebuffers_step(void)
{
    real_T rtb_DSR_last;
    real_T tmp;

    /* DataStoreRead: '<Root>/DSR' */
    rtb_DSR_last = mem.last;

    /* Switch: '<Root>/Switch' incorporates:
     * Constant: '<Root>/Constant'
     * DataStoreRead: '<Root>/DSR'
     * Inport: '<Root>/Clear'
     */
    if (rtU.Clear) {
        tmp = 0.0;
    } else {
        tmp = mem.max;
    }

    /* End of Switch: '<Root>/Switch' */

    /* FunctionCaller: '<Root>/Optimize1' incorporates:
     * Inport: '<Root>/DataNew'
     */
    UpdateFunc(tmp, rtU.DataNew, mem.last, mem.max);

    /* Outport: '<Root>/Delta' incorporates:
     * Inport: '<Root>/DataNew'
     * Sum: '<Root>/Sum'
     */
}

```

```
    rtY.Delta = rtU.DataNew - rtb_DSR_last;  
}
```

The data copy for the Data Store Write block is not in the generated code. The code contains the data copy for the Data Store Read block because the Sum block executes after the Data Store Write block. The generated code contains the variable `rtb_DSR_last` to hold the output of the Sum block. Therefore, the Sum block gets the values that `Optimize1` calculates at the start of the time step rather than those values at the next time step. If the priority of the Sum block is lower than `Optimize1`, the code generator can remove the data copy for the Data Store Read block.

Some other cases in which the code generator might not eliminate data copies are:

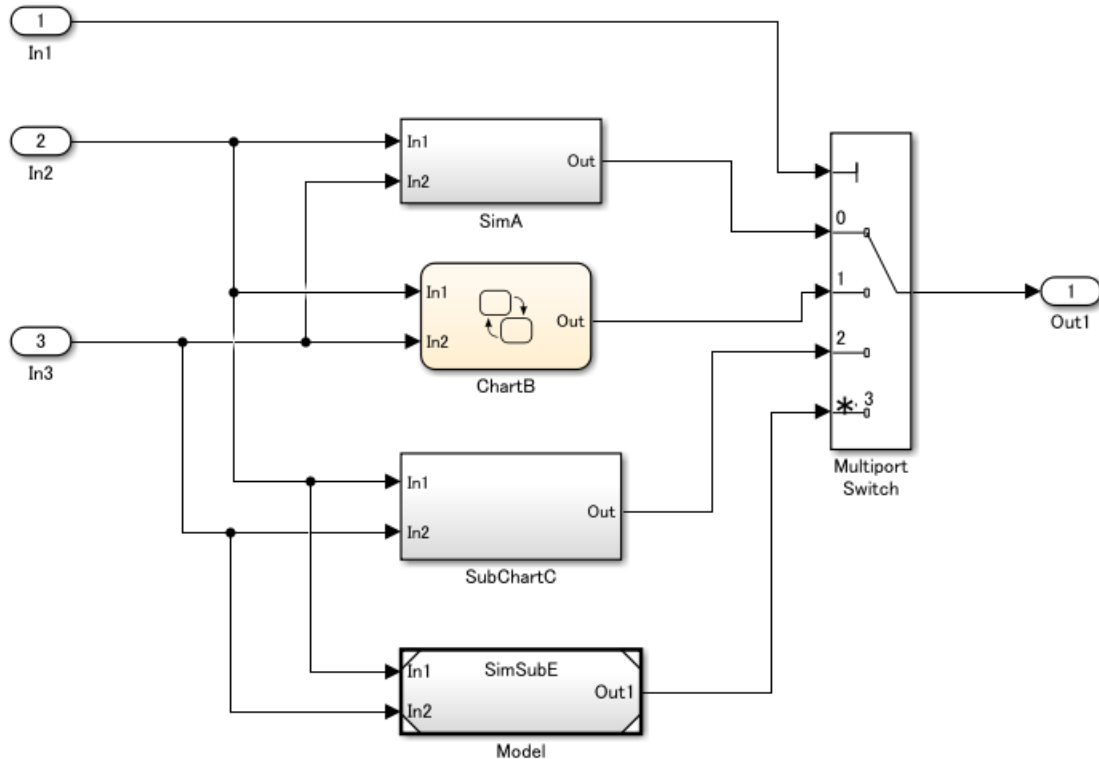
- A Simulink Function internally writes to the Data Store Memory block.
- The Data Store Read or Data Store Write blocks select elements of an array from the Data Store Memory block.
- The Data Store Memory block has a custom storage class.
- The Data Store Read and Data Store Write blocks occur on the same block unless that block is a Bus Assignment block or an Assignment block.

Note You can disable this optimization by setting the **Reuse buffers for Data Store Read and Data Store Write blocks** parameter to off. In the Configuration Parameters dialog box, this parameter is on the **All Parameters** tab.

More efficient code for Model blocks

In R2017a, the generated code contains additional optimizations for modeling patterns involving Model blocks. These optimizations include turning global variables into local variables, buffer elimination, data copy reduction, and expression folding. The optimizations improve ROM and RAM consumption and increase code execution speed.

For example, the model `model_ref` contains the Model block `SimSubE`.



In R2016b, the code generator produced this code:

```

/* Model step function */
void model_ref_step(void)
{
    /* local block i/o variables */
    uint16_T u16_Model;
    uint16_T u16_Out_m;
    uint16_T u16_Out;
    if (model_ref_U.In3 > 30U) {
        u16_Out_m = model_ref_U.In2 + /*MW:0vSat0k*/ 5U;
        if (u16_Out_m < model_ref_U.In2) {
            u16_Out_m = MAX_uint16_T;
        }
    } else {
        u16_Out_m = model_ref_U.In2 + /*MW:0vSat0k*/ 12U;
        if (u16_Out_m < model_ref_U.In2) {
            u16_Out_m = MAX_uint16_T;
        }
    }

    if (model_ref_U.In2 > 20U) {
        u16_Out = model_ref_U.In2 + model_ref_U.In3;
    } else {
        u16_Out = model_ref_U.In2 + /*MW:0vSat0k*/ 2U;
        if (u16_Out < model_ref_U.In2) {
            u16_Out = MAX_uint16_T;
        }
    }
}

```

```

SimSubE(&model_ref_U.In2, &model_ref_U.In3, &u16_Model);
switch (model_ref_U.In1) {
  case 0:
    model_ref_Y.Out1 = (model_ref_U.In2 + model_ref_U.In3) * 5U;
    break;

  case 1:
    model_ref_Y.Out1 = u16_Out_m;
    break;

  case 2:
    model_ref_Y.Out1 = u16_Out;
    break;

  default:
    model_ref_Y.Out1 = u16_Model;
    break;
}
}

```

In the `model_ref_step` function, there are three local variables. The `if-else` statements are above the `switch-case` statements, so they are unconditionally executed.

In R2017a, the code generator produces this code:

```

/* Model step function */
void model_ref_step(void)
{
  /* local block i/o variables */
  uint16_T u16_Model;
  uint16_T u16_qY;
  SimSubE(&model_ref_U.In2, &model_ref_U.In3, &u16_Model);
  switch (model_ref_U.In1) {
    case 0:
      model_ref_Y.Out1 = (model_ref_U.In2 + model_ref_U.In3) * 5U;
      break;

    case 1:
      if (model_ref_U.In3 > 30U) {
        u16_qY = model_ref_U.In2 + /*MW:0vSat0k*/ 5U;
        if (u16_qY < model_ref_U.In2) {
          u16_qY = MAX_uint16_T;
        }

        model_ref_Y.Out1 = u16_qY;
      } else {
        u16_qY = model_ref_U.In2 + /*MW:0vSat0k*/ 12U;
        if (u16_qY < model_ref_U.In2) {
          u16_qY = MAX_uint16_T;
        }

        model_ref_Y.Out1 = u16_qY;
      }
      break;

    case 2:
      if (model_ref_U.In2 > 20U) {

```

```

    model_ref_Y.Out1 = model_ref_U.In2 + model_ref_U.In3;
} else {
    u16_qY = model_ref_U.In2 + /*MW:0vSat0k*/ 2U;
    if (u16_qY < model_ref_U.In2) {
        u16_qY = MAX_uint16_T;
    }

    model_ref_Y.Out1 = u16_qY;
}
break;

default:
    model_ref_Y.Out1 = u16_Model;
    break;
}
}
}

```

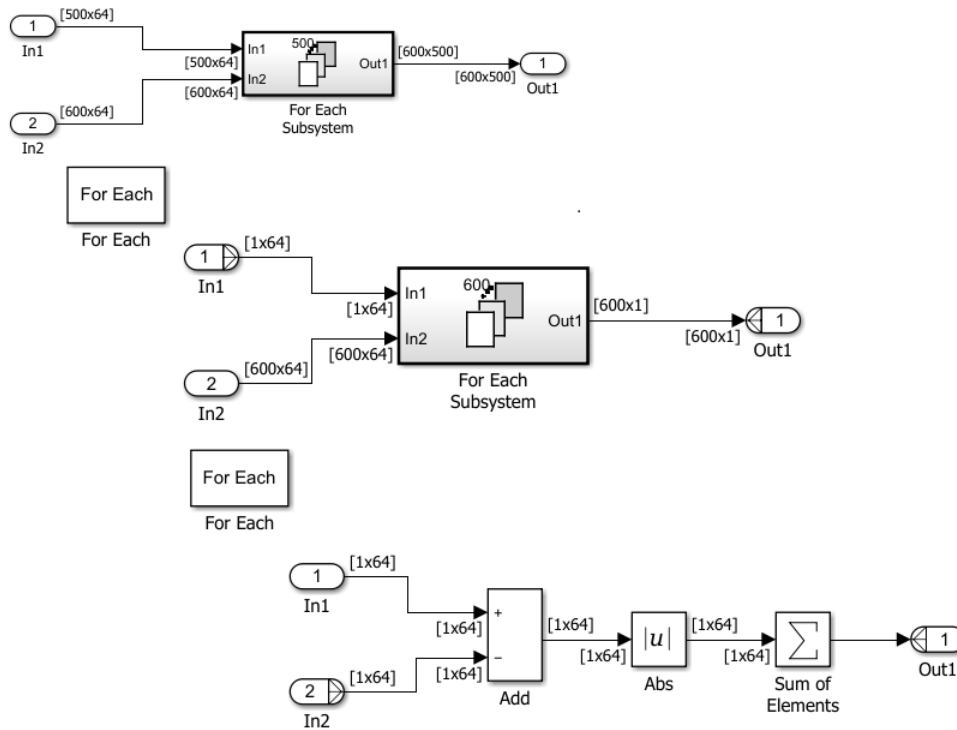
In the `model_ref_step` function, there are two local variables instead of three local variables which conserves stack space. Each `switch-case` statement includes the corresponding `if-else` statement. Including the `if-else` statements in the `switch-case` statements increases code execution speed because each `if-else` statement is only executed if the corresponding case statement is true.

Code Efficiency: Improve loop fusion for Sum of Elements blocks and generate less code for temporal logic in Stateflow

Loop fusion for Sum of Elements blocks

In R2017a, the code generator can fuse more `for` loops involving Sum of Elements blocks. This optimization conserves ROM consumption and improves code execution speed.

For example, the model `loop_fuse` contains a Sum of Elements block inside two nested For Each subsystems. The diagram shows the model `loop_fuse`, the For Each Subsystems and signal dimensions.



In R2016b, the code generator produced this code:

```

void loop_fuse_step(void)
{
    int32_T ForEach_itr;
    int32_T ForEach_itr_d;
    real_T tmp;
    real_T rtb_Abs[64];
    int32_T i;
    for (ForEach_itr = 0; ForEach_itr < 500; ForEach_itr++) {
        for (ForEach_itr_d = 0; ForEach_itr_d < 600; ForEach_itr_d++) {
            for (i = 0; i < 64; i++) {
                rtb_Abs[i] = fabs(loop_fuse_U.In1[500 * i + ForEach_itr] -
                                loop_fuse_U.In2[600 * i + ForEach_itr_d]);
            }

            tmp = rtb_Abs[0];
            for (i = 0; i < 63; i++) {
                tmp += rtb_Abs[i + 1];
            }

            loop_fuse_B.Out1_CoreSubsysCanOut[ForEach_itr_d] = tmp;
        }

        for (i = 0; i < 600; i++) {
            loop_fuse_Y.Out1[i + 600 * ForEach_itr] =
                loop_fuse_B.Out1_CoreSubsysCanOut[i];
        }
    }
}

```

```

    }
}

```

The generated code contained separate for loops for the Add and Abs blocks and the Sum of Elements block.

In R2017a, the code generator produces this code:

```

void loop_fuse_step(void)
{
    int32_T ForEach_itr;
    int32_T ForEach_itr_d;
    real_T tmp;
    int32_T i;
    for (ForEach_itr = 0; ForEach_itr < 500; ForEach_itr++) {
        for (ForEach_itr_d = 0; ForEach_itr_d < 600; ForEach_itr_d++) {
            tmp = 0.0;
            for (i = 0; i < 64; i++) {
                tmp += fabs(loop_fuse_U.In1[500 * i + ForEach_itr] - loop_fuse_U.In2[600
                    * i + ForEach_itr_d]);
            }

            loop_fuse_B.Out1_CoreSubsysCanOut[ForEach_itr_d] = tmp;
        }

        for (i = 0; i < 600; i++) {
            loop_fuse_Y.Out1[i + 600 * ForEach_itr] =
                loop_fuse_B.Out1_CoreSubsysCanOut[i];
        }
    }
}

```

The generated code contains one for loop for the Add and Abs blocks and the Sum of Elements block.

More efficient code for temporal logic in Stateflow

For some absolute-time constructs using fixed-point parameters, Stateflow generates more efficient code that does not contain floating-point operations.

For example, consider `after(DELAY, sec)` in a chart with a sample time of the chart < 1 second. DELAY is a fixed-point parameter. Previously the code generator created the following code:

```

counter >= (uint32_T)ceil((real_T)DELAY * 0.05 / 0.1 - 1e-9)
.

```

Now, it generates:

```

(counter >> 1) >= DELAY

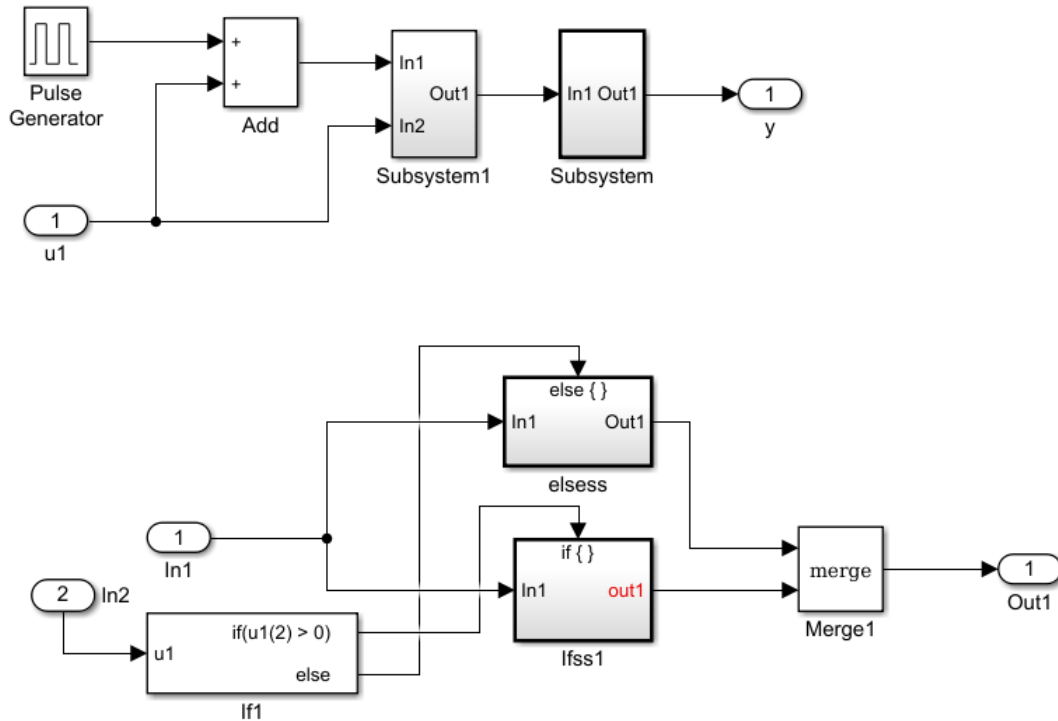
```

This code contains fewer operations and does not include floating-point operations.

Data copy reduction for Merge blocks

In R2017a, the code generator is improved to better reuse buffers around Merge blocks. This optimization conserves RAM and ROM consumption and increases code execution speed.

For example, the model `cond_reuse` contains the virtual subsystem `Subsystem1`. `Subsystem1` contains an if-else conditional structure that connects to a Merge block.



In R2016b, the code generator produced this code:

```

B_cond_reuse_T cond_reuse_B;
DW_cond_reuse_T cond_reuse_DW;
ExtU_cond_reuse_T cond_reuse_U;
ExtY_cond_reuse_T cond_reuse_Y;
RT_MODEL_cond_reuse_T cond_reuse_M_;
RT_MODEL_cond_reuse_T *const cond_reuse_M = &cond_reuse_M_;
void cond_reuse_Subsystem(void)
{
    int32_T i;
    for (i = 0; i < 64; i++) {
        cond_reuse_Y.y[i] = -3.0 * cond_reuse_B.Merge1[i];
    }
}
void cond_reuse_step(void)
{
    int32_T rtb_PulseGenerator;
    real_T rtb_Add[64];
    int32_T i;
    rtb_PulseGenerator = ((cond_reuse_DW.clockTickCounter < 1) &&
        (cond_reuse_DW.clockTickCounter >= 0));
    if (cond_reuse_DW.clockTickCounter >= 19) {
        cond_reuse_DW.clockTickCounter = 0;
    } else {
        cond_reuse_DW.clockTickCounter++;
    }
}

```

```

for (i = 0; i < 64; i++) {
    rtb_Add[i] = (real_T)rtb_PulseGenerator + cond_reuse_U.u1[i];
}

if (cond_reuse_U.u1[1] > 0.0) {
    memcpy(&cond_reuse_B.Mergel[0], &rtb_Add[0], sizeof(real_T) << 6U);
} else {
    for (i = 0; i < 64; i++) {
        cond_reuse_B.Mergel[i] = 22.0 * rtb_Add[i] * -3.0;
    }
}

cond_reuse_Subsystem();
}

```

The generated code contained full data copies to the temporary arrays `rtb_Add` and `cond_reuse_B.Mergel`.

In R2017a, the code generator produces this code:

```

DW_cond_reuse_T cond_reuse_DW;
ExtU_cond_reuse_T cond_reuse_U;
ExtY_cond_reuse_T cond_reuse_Y;
RT_MODEL_cond_reuse_T cond_reuse_M_;
RT_MODEL_cond_reuse_T *const cond_reuse_M = &cond_reuse_M_;
void cond_reuse_Subsystem(void)
{
    int32_T i;
    for (i = 0; i < 64; i++) {
        cond_reuse_Y.y[i] *= -3.0;
    }
}
void cond_reuse_step(void)
{
    int32_T rtb_PulseGenerator;
    int32_T i;
    rtb_PulseGenerator = ((cond_reuse_DW.clockTickCounter < 1) &&
        (cond_reuse_DW.clockTickCounter >= 0));
    if (cond_reuse_DW.clockTickCounter >= 19) {
        cond_reuse_DW.clockTickCounter = 0;
    } else {
        cond_reuse_DW.clockTickCounter++;
    }

    if (cond_reuse_U.u1[1] > 0.0) {
        for (i = 0; i < 64; i++) {
            cond_reuse_Y.y[i] = (real_T)rtb_PulseGenerator + cond_reuse_U.u1[i];
        }
    } else {
        for (i = 0; i < 64; i++) {
            cond_reuse_Y.y[i] = ((real_T)rtb_PulseGenerator + cond_reuse_U.u1[i]) *
                22.0 * -3.0;
        }
    }

    cond_reuse_Subsystem();
}

```

The temporary arrays `rtb_Add` and `cond_reuse_B.Merge1` and their associated data copies are not in the generated code. For the preceding model, you can also specify buffer reuse using `Simulink.Signal` objects. See [Specify Buffer Reuse for Multiple Signals in a Path](#).

More instances of buffer reuse for blocks and subsystems in a chain

In R2017a, the code generator can automatically reuse buffers for more modeling patterns involving blocks and subsystems in a chain. Specifically, the code generator can reuse buffers for these modeling patterns:

- A chain of blocks that includes reusable and nonreusable subsystems
- A chain of reusable subsystems
- A chain of blocks that includes a root-level Outputport block
- A chain of blocks that includes a mixture of signals with auto and reusable custom storage class specifications. However, the reusable custom storage class specification must be on a signal that leaves a root-level Inport block or enters a root-level Outputport block.

Note For buffer reuse to occur for these modeling patterns, in the Configuration Parameters dialog box, on the **All Parameters** tab, set the **Optimize global data access** parameter to `Use global` to hold temporary results. For models containing reusable subsystems, on the **Optimization > Signals and Parameters** tab, set the **Pass reusable subsystem outputs as** parameter to `Individual arguments`.

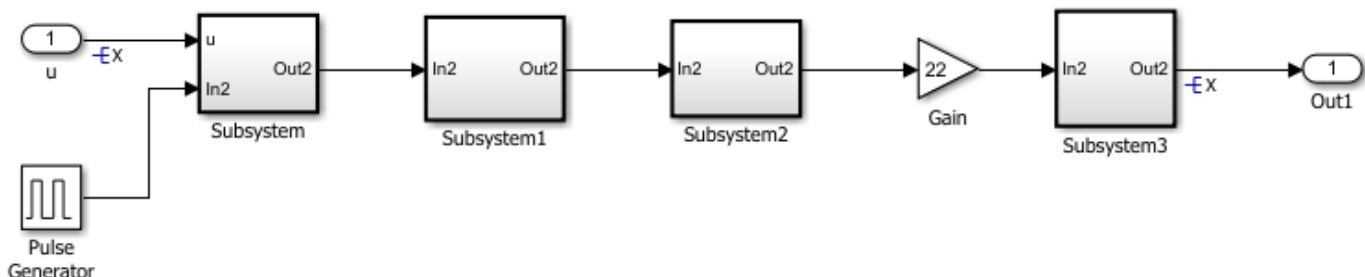
These optimizations reduce data copies in the generated code thereby conserving RAM and ROM consumption and improving code execution speed.

Buffer reuse for a chain of reusable and nonreusable subsystems

The code generator can now reuse buffers for a chain of reusable and nonreusable subsystems. This chain can include a root-level Outputport block. It can also contain a mixture of signals with auto and reusable custom storage class specifications. However, the reusable custom storage class specification must be on a signal that leaves a root-level Inport block or enters a root-level Outputport block.

For example, the model `Chainbuffer` contains the reusable subsystems `Subsystem`, `Subsystem1`, and `Subsystem2`. For a reusable subsystem, the generated code is a function with arguments.

The model also contains the nonreusable subsystem `Subsystem3`. For `Subsystem3`, the **Function interface** parameter has a value of `void-void`. The signal leaving `u` and entering `Out1` resolves to the `Simulink.signal X`. `X` has a reusable custom storage class.



In R2016b, the code generator produced this code.

```

real_T X[64];
B_Chainbuffer_T Chainbuffer_B;
...
void Chainbuffer_Subsystem3(void)
{
    int32_T i;
    for (i = 0; i < 64; i++) {
        X[i] = 22.0 * Chainbuffer_B.Gain[i] * 22.0;
    }
}
void Chainbuffer_step(void)
{
    int32_T rtb_PulseGenerator;
    real_T rtb_Gain1_o[64];
    real_T rtb_Gain1_a[64];
    rtb_PulseGenerator = ((Chainbuffer_DW.clockTickCounter < 1) &&
        (Chainbuffer_DW.clockTickCounter >= 0));
    if (Chainbuffer_DW.clockTickCounter >= 19) {
        Chainbuffer_DW.clockTickCounter = 0;
    } else {
        Chainbuffer_DW.clockTickCounter++;
    }

    Chainbuffer_Subsystem((&X[0])), (real_T)rtb_PulseGenerator, rtb_Gain1_o);
    Chainbuffer_Subsystem1(rtb_Gain1_o, rtb_Gain1_a);
    Chainbuffer_Subsystem1(rtb_Gain1_a, Chainbuffer_B.Gain);
    for (rtb_PulseGenerator = 0; rtb_PulseGenerator < 64; rtb_PulseGenerator++) {
        Chainbuffer_B.Gain[rtb_PulseGenerator] *= 22.0;
    }

    Chainbuffer_Subsystem3();
}

```

The generated code contained the global buffer `Chainbuffer_B.Gain` and the local buffers `rtb_Gain1_o` and `rtb_Gain1_a` for holding the inputs and outputs of `Subsystem`, `Subsystem1`, `Subsystem2`, and `Subsystem3`.

In R2017a, the code generator produces this code.

```

real_T X[64];
...
void Chainbuffer_Subsystem3(void)
{
    int32_T i;
    for (i = 0; i < 64; i++) {
        X[i] = 22.0 * X[i] * 22.0;
    }
}
void Chainbuffer_step(void)
{
    int32_T rtb_PulseGenerator;
    rtb_PulseGenerator = ((Chainbuffer_DW.clockTickCounter < 1) &&
        (Chainbuffer_DW.clockTickCounter >= 0));
    if (Chainbuffer_DW.clockTickCounter >= 19) {
        Chainbuffer_DW.clockTickCounter = 0;
    } else {

```

```

Chainbuffer_DW.clockTickCounter++;
}

Chainbuffer_Subsystem((&(X[0])), (real_T)rtb_PulseGenerator, (&(X[0]]));
Chainbuffer_Subsystem1((&(X[0])), (&(X[0]]));
Chainbuffer_Subsystem1((&(X[0])), (&(X[0]]));
for (rtb_PulseGenerator = 0; rtb_PulseGenerator < 64; rtb_PulseGenerator++) {
    X[rtb_PulseGenerator] = 22.0 * X[rtb_PulseGenerator];
}

Chainbuffer_Subsystem3();
}

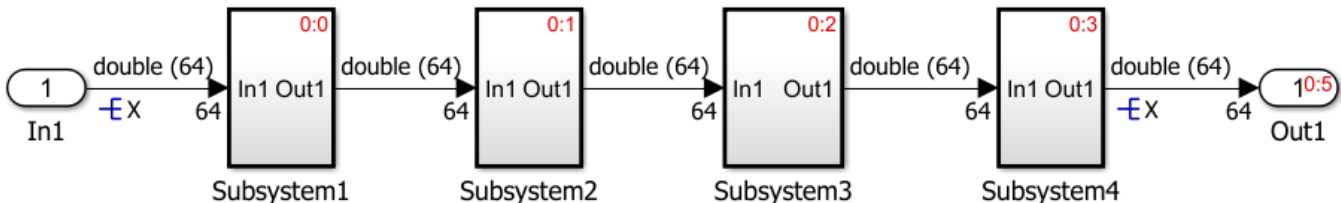
```

The generated code contains the global buffer X for holding the inputs and outputs of Subsystem, Subsystem1, Subsystem2, and Subsystem3.

Buffer reuse for a chain of reusable subsystems

The code generator can now reuse the arguments of reusable subsystems in a chain.

For example, the model `subsreuse` contains four subsystems. For the four subsystems, in the Subsystem Block Parameters dialog box, on the **Code Generation** tab, the **Function packaging** parameter is set to `Reusable` function. The input and output signals resolve to the Simulink.Signal X. This signal has a **Storage class** of `Reusable` (Custom).



In R2016b, the code generator produced this code:

```

void subsreuse_step(void)
{
    real_T rtb_Gain1[64];
    real_T rtb_Gain2[64];
    subsreuse_Subsystem1((&(X[0])), rtb_Gain1, (P_Subsystem1_subsreuse_T *)
        &subsreuse_P.Subsystem1);
    subsreuse_Subsystem2(rtb_Gain1, rtb_Gain2, (P_Subsystem2_subsreuse_T *)
        &subsreuse_P.Subsystem2);
    subsreuse_Subsystem3(rtb_Gain2, rtb_Gain1, (P_Subsystem3_subsreuse_T *)
        &subsreuse_P.Subsystem3);
    subsreuse_Subsystem4(rtb_Gain1, (&(X[0])), (P_Subsystem4_subsreuse_T *)
        &subsreuse_P.Subsystem4);
}

```

The code contained two temporary variables, `rtb_Gain1` and `rtb_Gain2`, for holding the input and output of each function.

In R2017a, the code generator produces this code:

```

void subsreuse_step(void)
{

```

```

subsreuse_Subsystem1((&(X[0])), (&(X[0])), &subsreuse_P.Subsystem1);
subsreuse_Subsystem2((&(X[0])), (&(X[0])), &subsreuse_P.Subsystem2);
subsreuse_Subsystem3((&(X[0])), (&(X[0])), &subsreuse_P.Subsystem3);
subsreuse_Subsystem4((&(X[0])), (&(X[0])), &subsreuse_P.Subsystem4);
}

```

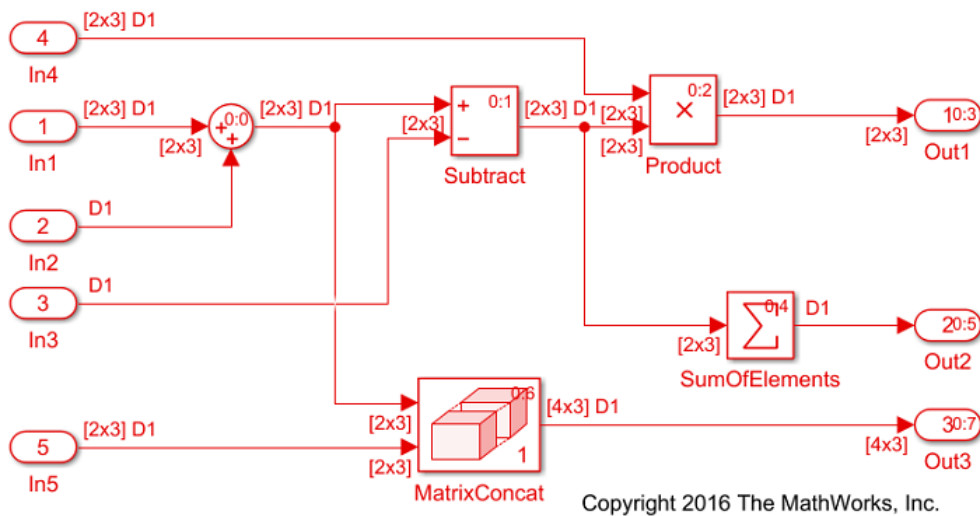
The generated code uses one global variable X for the input and output of each function.

Improved buffer reuse due to changes in block execution order

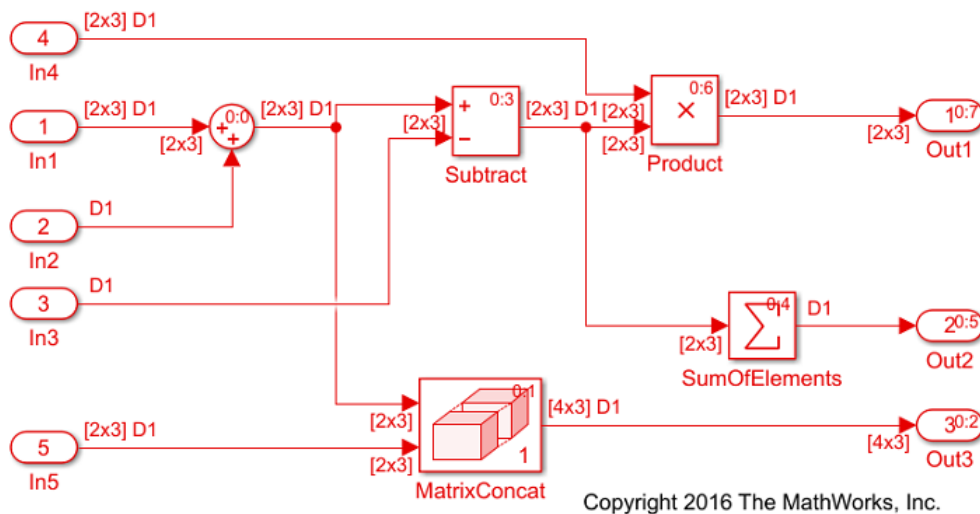
In R2016b, if you specified a signal for reuse, the code generator changed the block operation order so that buffer reuse occurred.

In R2017a, even if you do not specify a signal for reuse, the code generator can change the block operation order so that buffer reuse can occur. If the generated code contains extra buffers, you can try to eliminate them by setting the **Optimize block operation order in the generated code** parameter to **Improve Execution Speed**. In the Configuration Parameters dialog box, this parameter is on the **All Parameters** tab. Reusing buffers conserves RAM and ROM consumption and improves code execution speed.

For example, for the model `rtwdemo_optimizeblockorder`, the red numbers that follow the zeroes and colons represent the block execution order in R2016b. The Matrix Concatenate block executes after the Subtract block. The Sum of Elements block executes after the Product block. This block execution order prevents the same variables from being reused as the input and output to the Subtract and Product blocks in the generated code. As a result, there are two extra temporary arrays, two extra variables, and associated data copies for holding the inputs to these blocks.



In R2017a, the code generator can reorder the block execution order so that the Matrix Concatenate block executes before the Subtract block and the Sum of Elements block executes before the Product block. Reordering block operations eliminates the two temporary arrays, the two variables, and their associated data copies from the generated code. The blocks can use the same variable for the input and output.



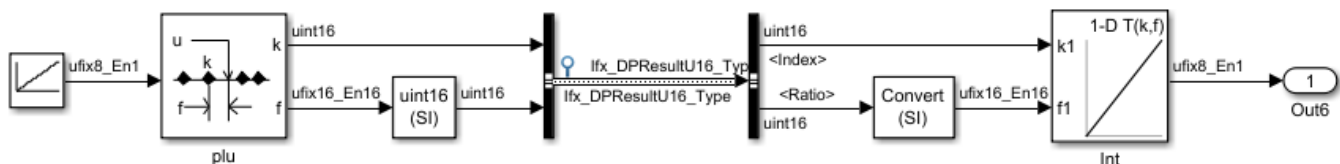
Note To implement buffer reuse, the code generator does not violate user-specified block priorities.

For more information, see [Remove Data Copies by Reordering Block Operations in the Generated Code](#).

More efficient code for Bus Creator blocks

In R2017a, the generated code contains additional optimizations for modeling patterns involving Bus Creator blocks. These optimizations include turning global variables into local variables, buffer elimination, data copy reduction, and expression folding. The optimizations improve ROM and RAM consumption and increase code execution speed.

For example, the model `bus_creator_ex` contains two Bus Creator blocks.



In R2016b, the `bus_creator.c` file contained this code:

```
void bus_creator_ex_step(void)
{
    Ifx_DPResultU16_Type dpResult;
    Ifx_DPResultU16_Type dpResult_0;
    Ifx_DPSearch_u8(&dpResult,
        bus_creator_ex_ConstP.Vector_Value[bus_creator_ex_DW.Output_DSTATE],
        2U, (*Rte_CData_BP1_SlopeBiasScaling_0_8_0p5_0()));
    bus_creator_ex_B.If_x_DPResultU16_Type_h.Index = dpResult.Index;
    bus_creator_ex_B.If_x_DPResultU16_Type_h.Ratio = dpResult.Ratio;
    dpResult_0.Index = bus_creator_ex_B.If_x_DPResultU16_Type_h.Index;
    dpResult_0.Ratio = bus_creator_ex_B.If_x_DPResultU16_Type_h.Ratio;
}
```

```

    bus_creator_ex_Y.Out6 = Ifx_IpoCur_u8(&dpResult_0,
        (*Rte_CData_TB1_SlopeBiasScaling_0_8_0p5_0()));
    bus_creator_ex_DW.Output_DSTATE++;
}

```

The code contained two local variables `dpResult` and `dpResult_0` for holding values prior to and from the Bus Creator blocks.

In R2017a, the `bus_creator.c` file contains this code:

```

void bus_creator_ex_step(void)
{
    Ifx_DPResultU16_Type dpResult;
    Ifx_DPSearch_u8(&dpResult,
        bus_creator_ex_ConstP.Vector_Value[bus_creator_ex_DW.Output_DSTATE],
        2U, (*Rte_CData_BP1_SlopeBiasScaling_0_8_0p5_0()));
    bus_creator_ex_B.Ifx_DPResultU16_Type_h.Index = dpResult.Index;
    bus_creator_ex_B.Ifx_DPResultU16_Type_h.Ratio = dpResult.Ratio;
    dpResult.Index = bus_creator_ex_B.Ifx_DPResultU16_Type_h.Index;
    dpResult.Ratio = bus_creator_ex_B.Ifx_DPResultU16_Type_h.Ratio;
    bus_creator_ex_Y.Out6 = Ifx_IpoCur_u8(&dpResult,
        (*Rte_CData_TB1_SlopeBiasScaling_0_8_0p5_0()));
    bus_creator_ex_DW.Output_DSTATE++;
}

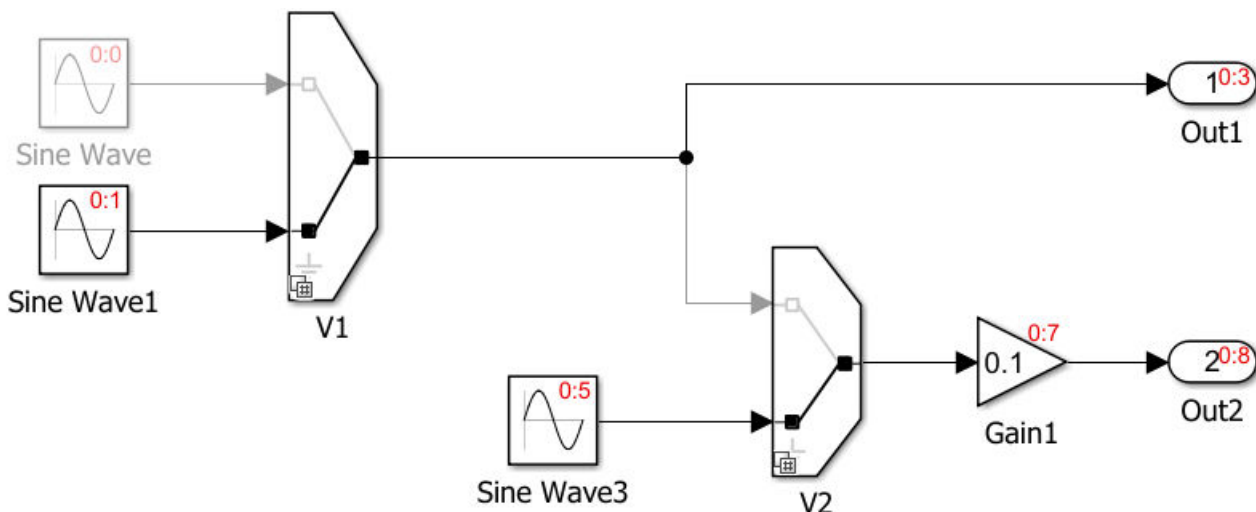
```

The generated code contains one less local variable.

Buffer reuse for Variant Source blocks

In R2017a, the code generator can reuse the buffer for Variant Source blocks.

For example, the model `VariantMergeReuse` contains two Variant Source blocks.



In R2016b, the code generator produced this code in the `VariantMergeReuse` step function:

```

#if V == 1 || V == 2
    real_T rtb_VariantMerge_For_Variant_So;

```



```
#endif                                     /* V == 1 || V == 2 */
#if (V == 1 && W == 1) || (V == 2 && W == 1) || W == 2
    real_T rtb_VariantMerge_For_Variant__k;
#endif
```

The code contained two buffers for holding intermediate values.

In R2017a, the code generator produces this code in the `VariantMergeReuse` step function:

```
#if V == 1 || V == 2 || W == 2
    real_T rtb_VariantMerge_For_Variant_So;
#endif                                     /* V == 1 || V == 2 || W == 2 */
```

The code contains one buffer for holding intermediate values.

Verification

SIL and PIL Testing: Log signals inside exported functions and stream signals to Simulation Data Inspector during simulation

To examine internal signals of a model component, you can enable internal signal logging for a top-model or Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. In R2017a, you can:

- Log signals inside export-function models.
- Stream the logged signals to the Simulation Data Inspector, where you can observe the signals during the SIL or PIL simulation.

For more information, see:

- <https://www.mathworks.com/help/releases/R2017a/ecoder/ug/configuring-a-sil-or-pil-simulation.html#br74o58-1>
- Export-Function Models (Simulink)
- General SIL and PIL Limitations

Verification of PIL target connectivity configuration

The `piltest` function provides additional tests for verifying your custom processor-in-the-loop (PIL) target connectivity configuration.

'Testpoint' Argument Value	Description
'verifyTopModelSILPILSwitching'	<p>New in R2017a.</p> <p>For a Simulink top model, the function:</p> <ul style="list-style-type: none"> • Verifies that production code is not regenerated when the simulation mode switches between software-in-the-loop (SIL) and PIL. The function compares timestamps of the production code used in each mode. • Compares results from SIL and PIL mode simulations to results from a normal mode simulation.

'Testpoint' Argument Value	Description
'verifyModelBlockSILPILSwitching'	<p>New in R2017a.</p> <p>For a Simulink Model block, the function:</p> <ul style="list-style-type: none"> • Verifies that production code is not regenerated when the Model block simulation mode switches between SIL and PIL. The function compares timestamps of the production code used in each mode. • Runs simulation loops with the Model block in SIL and PIL modes. The function varies the Code interface Model block parameter, setting this parameter to Top model or Model reference. <p>The function compares results from SIL and PIL mode simulations to results from a normal mode simulation.</p>
'verifyModelBlock'	<p>Updated in R2017a.</p> <p>The function runs simulation loops with a Simulink Model block in PIL mode. The function varies the Configuration Parameters > Code Generation > Language parameter, setting this parameter to C or C++. For C++, the function sets Code Generation > Interface > Code interface packaging to C++ class.</p> <p>Previously, Language was set to C .</p>

For more information, see Create PIL Target Connectivity Configuration.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.11

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Static code metrics report for C++ code

In R2016b, when you generate standalone C++ code, the HTML code generation report includes a static code metrics report. See [Generate a Static Code Metrics Report for MATLAB Code and Static Code Metrics](#).

Verification of `size_t` and `ptrdiff_t` hardware settings

In the project build settings, on the **Hardware** tab, R2016b provides values for the ANSI® C data types `size_t` and `ptrdiff_t`. At the start of a processor-in-the-loop (PIL) execution, the software verifies the values with reference to the target hardware.

Verification of PIL target connectivity configuration

Through the `piltest` function, you can use a test suite to verify your custom processor-in-the-loop (PIL) target connectivity configuration. Verify the target connectivity configuration early and independently of your algorithm development and code generation.

For more information, see:

- [Create PIL Target Connectivity Configuration](#)
- [PIL Execution of Code Generated for a Kalman Estimator](#)

Optimization for array indexing in loops

In R2016b, if you use Embedded Coder to generate C/C++ code from MATLAB code, you can enable an optimization that simplifies array indexing in loops in the generated code. When possible, for array indices in loops, this optimization replaces multiply operations with add operations. Multiply operations can be expensive. This optimization, referred to as strength reduction, is useful when the C/C++ compiler on the target platform does not optimize the array indexing.

Here is code generated without the optimization:

```
for (i = 0; i < 10; i++) {  
    z[5 * (1 + i) - 1] = x[5 * (1 + i)];  
}
```

Here is code generated with the optimization:

```
for (b_i = 0; b_i < 10; b_i++) {  
    z[i + 4] = x[i + 5];  
    i += 5;  
}
```

By default, the strength reduction optimization is disabled. To enable it:

- At the command line, set the configuration object parameter `EnableStrengthReduction` to `true`.
- In the MATLAB Coder app, project build settings, on the **All Settings** tab, set **Simplify array indexing** to Yes.

Even when the optimization replaces the multiply operations in the generated code, it is possible that the C/C++ compiler can generate multiply instructions.

Reduction of the Intel Performance Primitives (IPP) code replacement libraries (CRL)

The code replacement libraries (CRL) related to features, such as matrix multiple and dot product, that are no longer supported by the Intel Performance Primitives (IPP) library will be removed in a future release.

Model Architecture and Design

AUTOSAR Basic Software (BSW) Services: Simulate BSW including Diagnostic Event Manager (DEM) and NVRAM Manager (NvM)

The AUTOSAR standard defines important services as part of Basic Software (BSW) that runs in the AUTOSAR runtime environment (RTE). Examples include the NVRAM Manager (NvM) and the Diagnostic Event Manager (Dem). In the AUTOSAR RTE, AUTOSAR software components typically access BSW services using client-server or sender-receiver communication.

To support system-level modeling of AUTOSAR components and services, R2016b adds an AUTOSAR Basic Software block library. The library contains preconfigured Function Caller blocks for modeling component calls to AUTOSAR BSW services.

- Diagnostic Event Manager (Dem) blocks — Calls to Dem service interfaces, including `CallbackEventStatusChangeCaller`, `DiagnosticInfoCaller`, and `DiagnosticMonitorCaller`.
- NVRAM Manager (NvM) blocks — Calls to NvM service interfaces, including `NvMAdminCaller` and `NvMServiceCaller`.

To implement client calls to AUTOSAR BSW service interfaces in your AUTOSAR software component, you drag and drop Basic Software blocks into an AUTOSAR model and click a **Synchronize** icon. The software automatically configures the client calls in the AUTOSAR configuration. For more information, see [Model AUTOSAR Basic Software \(BSW\) Service Calls](#), [Configure Calls to AUTOSAR Diagnostic Event Manager \(Dem\) Service](#), and [Configure Calls to AUTOSAR NVRAM Manager \(NvM\) Service](#).

AUTOSAR Parameters: Model STD_AXIS and COM_AXIS lookup table parameters, export SwRecordLayouts, and apply SwAddrMethods

R2016b enhances AUTOSAR calibration parameter and data modeling with additional support for:

- “AUTOSAR STD_AXIS and COM_AXIS lookup tables” on page 13-4
- “AUTOSAR port-based and internal calibration parameters” on page 13-5
- “AUTOSAR SwRecordLayouts for lookup tables” on page 13-5
- “AUTOSAR SwAddrMethods for measurement and calibration tools” on page 13-5

AUTOSAR STD_AXIS and COM_AXIS lookup tables

AUTOSAR applications can use lookup tables in either or both of two ways:

- Implement high-performance search operations.
- Support tuning of the application with measurement and calibration tools.

To model lookup tables for automotive application tuning, use the new classes `Simulink.LookupTable` and `Simulink.Breakpoint` to store tunable table and breakpoint data. Simulink lookup table blocks have additional parameters to support the use of `Simulink.LookupTable` and `Simulink.Breakpoint` objects. AUTOSAR models can leverage the new classes to model STD_AXIS and COM_AXIS lookup tables. In Simulink, you can:

- Import `arxml` files that contain AUTOSAR lookup tables in STD_AXIS and COM_AXIS configurations.

- Create STD_AXIS and COM_AXIS lookup tables and map them to AUTOSAR parameters. In R2016b, you can create AUTOSAR parameters for lookup tables graphically, using the AUTOSAR Properties Explorer, or programmatically, using AUTOSAR property functions. For more information, see “AUTOSAR port-based and internal calibration parameters” on page 13-5.
- Generate arxml and C code with STD_AXIS and COM_AXIS lookup table content.

For more information, see [Configure STD_AXIS and COM_AXIS Lookup Tables for AUTOSAR Measurement and Calibration](#).

AUTOSAR port-based and internal calibration parameters

To support mapping a Simulink lookup table to an AUTOSAR parameter, you can now create AUTOSAR calibration parameters (`ParameterDataPrototypes`) using the AUTOSAR Properties Explorer or AUTOSAR property functions. You can create either internal AUTOSAR parameters, defined and accessed only within your software component, or port-based AUTOSAR parameters, associated with a port-based parameter interface.

The AUTOSAR parameters that you create subsequently are available for Simulink lookup table mapping, using the Simulink-AUTOSAR Mapping Explorer or AUTOSAR map functions.

For more information, see [Configure AUTOSAR Port-Based Calibration Parameters](#).

AUTOSAR SwRecordLayouts for lookup tables

AUTOSAR software components use software record layouts (`SwRecordLayouts`) to specify how to serialize data in the memory of an AUTOSAR ECU. The arxml importer imports and preserves the `SwRecordLayout` property for AUTOSAR data.

R2016b allows you to import `SwRecordLayouts` from arxml files in either of two ways:

- If you create your AUTOSAR model from arxml files using importer method `createComponentAsModel`, include an arxml file that contains `SwRecordLayout` definitions in the import. The imported `SwRecordLayouts` are preserved and subsequently exported in arxml code.
- If you create your AUTOSAR model in Simulink, you can import reference definitions of `SwRecordLayouts` from arxml files. When you generate model code, the exported arxml code contains references to the imported read-only `SwRecordLayout` elements, but not their definitions.

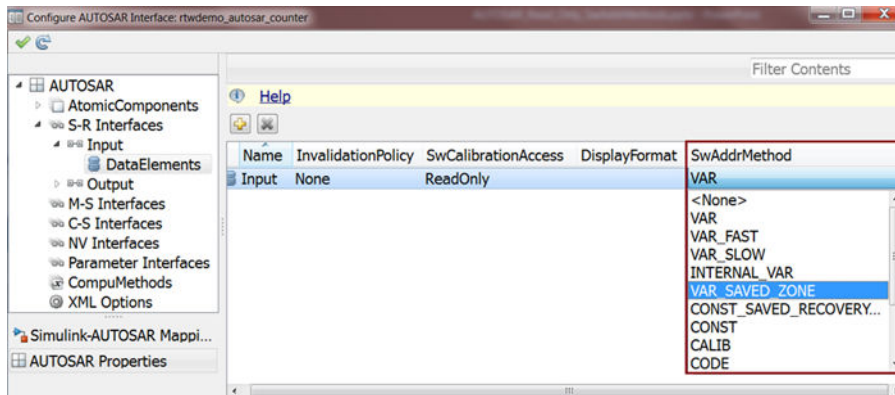
For more information, see [Configure AUTOSAR Data for Measurement and Calibration](#).

AUTOSAR SwAddrMethods for measurement and calibration tools

AUTOSAR software components use software address methods (`SwAddrMethods`) to group data in memory for access by measurement and calibration tools. In an AUTOSAR software component configuration, you assign common memory sections to data. When the runtime environment instantiates calibration parameters, calibration parameters that reference the same `SwAddrMethod` are placed within the same calibration parameter group.

The arxml importer imports and preserves the `SwAddrMethod` property for AUTOSAR data. In previous releases, in Simulink, you could assign memory sections to global constant and static memory, using AUTOSAR data objects. But you could not assign `SwAddrMethods` or memory sections to data accessed by RTE function calls, such as sender-receiver (S-R) interface data elements or inter-runnable variables (IRVs).

R2016b allows you to graphically or programmatically select imported SwAddrMethod values for AUTOSAR data accessed by RTE function calls.



When you build the model, the exported arxml code reflects the SwAddrMethod values you selected.

```
<SENDER-RECEIVER-INTERFACE>
  <SHORT-NAME>Input</SHORT-NAME>
  <IS-SERVICE>false</IS-SERVICE>
  <DATA-ELEMENTS>
    <VARIABLE-DATA-PROTOTYPE>
      <SHORT-NAME>Input</SHORT-NAME>
      <CATEGORY>VALUE</CATEGORY>
      ...
      <SW-DATA-DEF-PROPS>
        <SW-ADDR-METHOD-REF DEST="SW-ADDR-METHOD">
          /AUTOSAR/MemMap/SwAddrMethods_Blueprint/VAR_SAVED_ZONE</SW-ADDR-METHOD-REF>
        ...
      </SW-DATA-DEF-PROPS>
      ...
    </VARIABLE-DATA-PROTOTYPE>
  </DATA-ELEMENTS>
</SENDER-RECEIVER-INTERFACE>
```

For more information, see [Configure AUTOSAR Data for Measurement and Calibration](#).

AUTOSAR startup, reset, and shutdown modeling

AUTOSAR applications sometimes require complex logic to execute during system initialization, reset, and termination sequences. R2016b introduces the Simulink blocks Initialize Function and Terminate Function. You can use these blocks to control execution of a component in response to initialize, reset, or terminate events at any level of a model hierarchy. Each nonvirtual subsystem can have its own set of initialize, reset, and terminate functions. In a lower-level model, Simulink aggregates the content of the functions with corresponding instances in the parent model.

AUTOSAR models can leverage the new blocks to model potentially complex AUTOSAR startup, reset, and shutdown sequences. The subsystems work with any AUTOSAR component modeling style.

For more information, see [Startup, Reset, and Shutdown](#) and [Configure AUTOSAR Initialize, Reset, or Terminate Runnables](#).

AUTOSAR external trigger event communication

AUTOSAR Release 4.0 introduced external trigger event communication, in which an AUTOSAR component or service signals an external trigger occurred event

(ExternalTriggerOccurredEvent) to another component. The receiving component activates a runnable in response to the event.

Embedded Coder now supports modeling the receiver portion of AUTOSAR external trigger event communication. In a component that you want to react to an external trigger, you create a trigger interface, a trigger receiver port to receive an ExternalTriggerOccurredEvent, and a runnable that is activated by the event.

For more information, see [Configure Receiver for AUTOSAR External Trigger Event Communication](#).

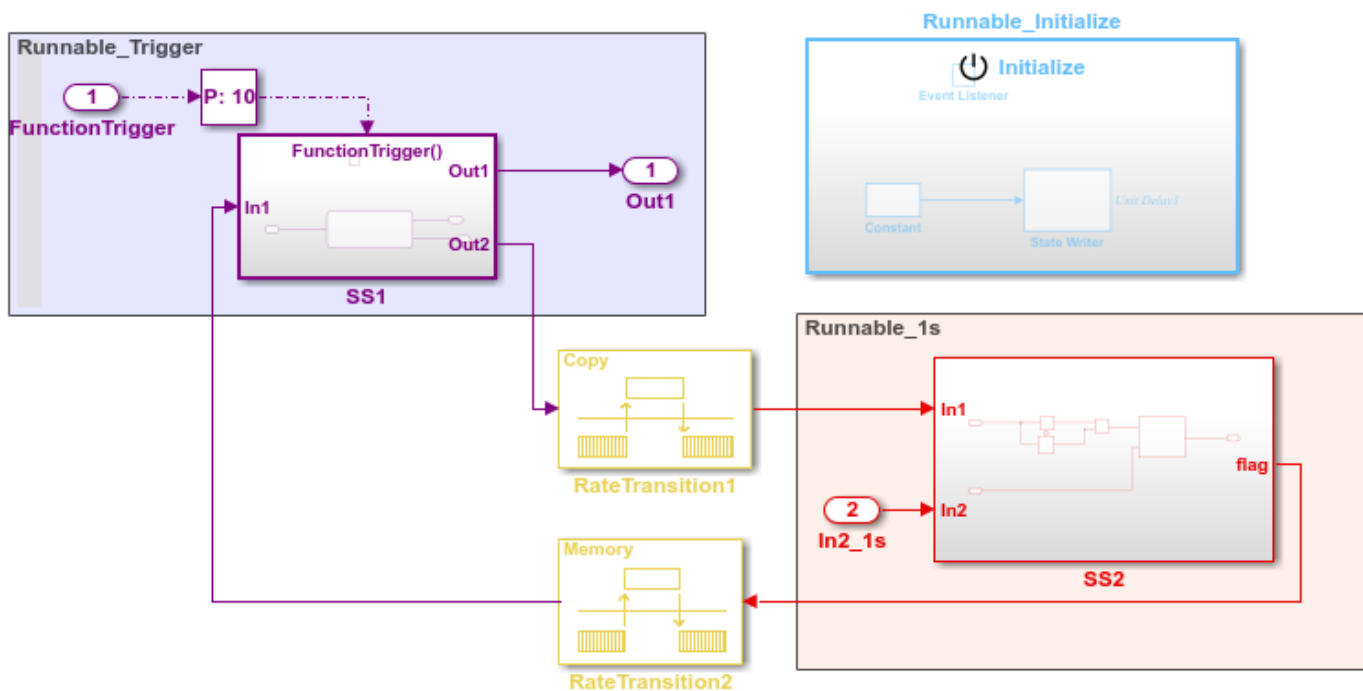
AUTOSAR support for JMAAB model architecture

Embedded Coder supports AUTOSAR code generation for the model architectures described in the Japan MBD Automotive Advisory Board (JMAAB) document *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow - Version 4.01*. The document is available from the MAAB Web page at <https://www.mathworks.com/solutions/automotive/standards/maab.html>.

The document describes three layouts for the top layer of a controller model:

- Simple control model — Represents a functions layer and a scheduling layer in one layer.
- Complex control model type alpha (α) — Places a scheduling layer above function layers.
- Complex control model type beta (β) — Places function layers above scheduling layers.

R2016b adds support for JMAAB type beta modeling in AUTOSAR models. For example, here is an AUTOSAR example model, `rtwdemo_autosar_swc_fcncalls`, in which an asynchronous function-call runnable at the top level of the model interacts with a periodic rate-based runnable. This type of component leverages periodic and asynchronous rates (sample times).



For more information about this component modeling style, see [.Add Top-Level Asynchronous Trigger to Periodic Rate-Based System](#).

AUTOSAR ExplicitReceiveByVal data access mode for receiver ports

R2016b adds support for modeling scalar explicit read by value access for AUTOSAR receiver ports, and generating the corresponding AUTOSAR API `Rte_DRead` in C code. Reading data by value can produce more efficient and readable C code and reduce RAM requirements.

In Simulink, you can model the data access in the following ways:

- Import an `arxml` file that uses `DATA-RECEIVE-POINT-BY-VALUES` variable access for a port. The importer creates a root inport with `ExplicitReceiveByVal` data access and maps it to an AUTOSAR receiver port.
- Create a root inport, select `ExplicitReceiveByVal` data access, and map it to an AUTOSAR receiver port.

When you build the model, the exported `arxml` code defines `DATA-RECEIVE-POINT-BY-VALUES` variable access for the AUTOSAR receiver port.

```
<RUNNABLE-ENTITY UUID="...">
...
  <SHORT-NAME>Runnable_Step</SHORT-NAME>
...
  <DATA-RECEIVE-POINT-BY-VALUES>
    <VARIABLE-ACCESS UUID="...">
      <SHORT-NAME>IN_Input_Input</SHORT-NAME>
      <ACCESSED-VARIABLE>
        <AUTOSAR-VARIABLE-IREF>
          <PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">
            /pkg/swc/rtwdemo_autosar_counter/Inport</PORT-PROTOTYPE-REF>
          <TARGET-DATA-PROTOTYPE-REF DEST="VARIABLE-DATA-PROTOTYPE">
            /pkg/if/Input/Input</TARGET-DATA-PROTOTYPE-REF>
          </AUTOSAR-VARIABLE-IREF>
        </ACCESSED-VARIABLE>
      </VARIABLE-ACCESS>
    </DATA-RECEIVE-POINT-BY-VALUES>
  ...
</RUNNABLE-ENTITY>
```

The generated C code uses `Rte_DRead` API calls to receive the port data by value.

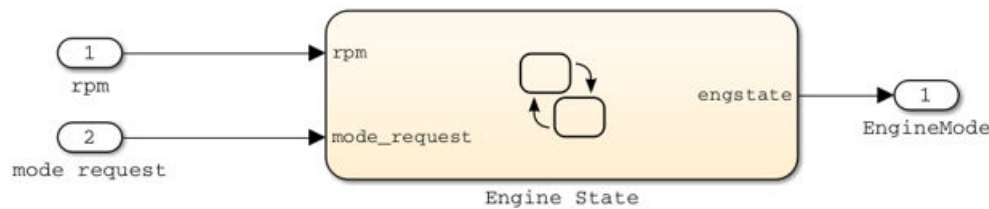
```
void Runnable_Step(void)
{
  ...
  /* Gain: '<S1>/Gain' incorporates:
   * Inport: '<Root>/Input'
   *
   * Block description for '<S1>/Gain':
   * This block references an AUTOSAR calibration parameter, which is
   * accessed using the AUTOSAR Rte_Calprm function signature.
   */
  rtwdemo_autosar_counter_B.Gain = Rte_Prm_rCounter_K() *
    Rte_DRead_Input_Input();
  ...
}
```

AUTOSAR ModeSenderPorts and ModeSwitchPoints for application mode management

AUTOSAR mode-switch (M-S) communication relies on a mode manager and connected mode users. The mode manager is an authoritative source for software components to query the current mode and to receive notification when the mode changes. A mode manager can be provided by AUTOSAR Basic Software (BSW) or implemented as an AUTOSAR software component. A mode manager implemented

as a software component is called an application mode manager. A software component that queries the mode manager and receives notifications of mode changes is a mode user.

R2016b enhances Simulink modeling of AUTOSAR M-S communication by adding the ability to model application mode manager components, including AUTOSAR mode sender ports (as defined in AUTOSAR Release 4). Mode sender ports output a mode switch to connected mode user components. For example, here is an application mode manager, modeled in Simulink, that uses a mode sender port to output the current value of EngineMode.



For more information, see Mode-Switch Interface and Configure AUTOSAR Mode-Switch Communication.

AUTOSAR reference element definitions for sharing among components and services

R2016b supports a new workflow for importing external AUTOSAR element definitions, defined in `arxml` files, for sharing among multiple AUTOSAR components and services. Benefits of sharing and reusing AUTOSAR element definitions include lower risk of definition conflicts and easier code integration. You can manage shared definitions in a centralized way.

Suppose that you have a large number of AUTOSAR software components that use similar packageable AUTOSAR elements in similar ways. You can define sets of *reference elements* in `arxml` files, and your software components can share them on a read-only basis. Each software component can import the element definitions it requires and reference them. When you build the model, exported `arxml` code contains references to the shared elements, but not their definitions. Their definitions remain in the reference element `arxml` source files.

If definitions of reference elements change, you modify them in the `arxml` files, and then import the updated definitions into the affected software components.

AUTOSAR elements that are supported for reference use in Simulink include:

- CompuMethod, Unit, and PhysicalDimension
- ImplementationDataType and SwBaseType
- SwSystemConst, SwSystemConstValueSet, and PredefinedVariant
- SwRecordLayout
- SwAddrMethod

For more information, see Import or Update Shared AUTOSAR Reference Element Definitions.

ERT Target Code Generation: Remove unreachable reset and disable functions to reduce dead code

In some model referencing contexts for ERT targets, generating code models can contain `reset` and `disable` functions that are dead code. You can use two new configuration parameters to remove the

generated `disable` and `reset` functions that cannot be reached from anywhere in the generated code. Avoiding dead code is essential in safety-critical applications.

The new configuration parameters are:

- Remove reset function (`RemoveResetFunc`)
- Remove disable function (`RemoveDisableFunc`)

See `Remove Reset and Disable Functions from the Generated Code`.

Compatibility Considerations

The **Remove reset function** configuration parameter replaces the **Optimize initialization code for model reference** parameter.

- In R2016b, if you load a legacy model from an earlier release that has the **Optimize initialization code for model reference** parameter set, the new **Remove reset function** parameter is set to produce the same behavior as the **Optimize initialization code for model reference** parameter produced.
- If you save a model created in R2016b to an earlier release, the **Optimize initialization code for model reference** parameter is updated appropriately. The model saved to a previous version reflects the behavior that was specified with the **Remove reset function** parameter.

Conditional compile time check for imported macros with ImportedDefine custom storage class

In R2016a, for a model that contained variant blocks and a `Simulink.Parameter` with an `ImportedDefine` custom storage class, the compile-time check for the `Simulink.Parameter` was unconditional. If the `Simulink.Parameter` was undefined, there was an error even if the `Simulink.Parameter` was in the inactive variant.

In R2016b, the compile-time check is conditional, so the error occurs only if the `Simulink.Parameter` is undefined and in the active variant.

Suppose that a model contains two `Variant Subsystem` blocks, `Variant A` and `Variant B`. `Variant B` contains a `Constant` block in which the **Constant value** parameter is the `Simulink.Parameter` `myvar`. `myvar` has an `ImportedDefine` custom storage class.

In R2016a, the `model.h` file contained this code:

```
#ifndef myvar
#error The variable for the parameter "myvar" is not defined
#endif
```

The code for `myvar` was not conditionally compiled. If you did not define `myvar` in a user-provided header file, there was an error.

In R2016b, the `model.h` file contains this code:

```
#if Variant B
#ifndef myvar
#error The variable for the parameter "myvar" is not defined
#endif
```

There is an error only if `myvar` is undefined and `Variant B` is the active variant because the code for `myvar` is conditionally compiled. See [Variant Systems](#).

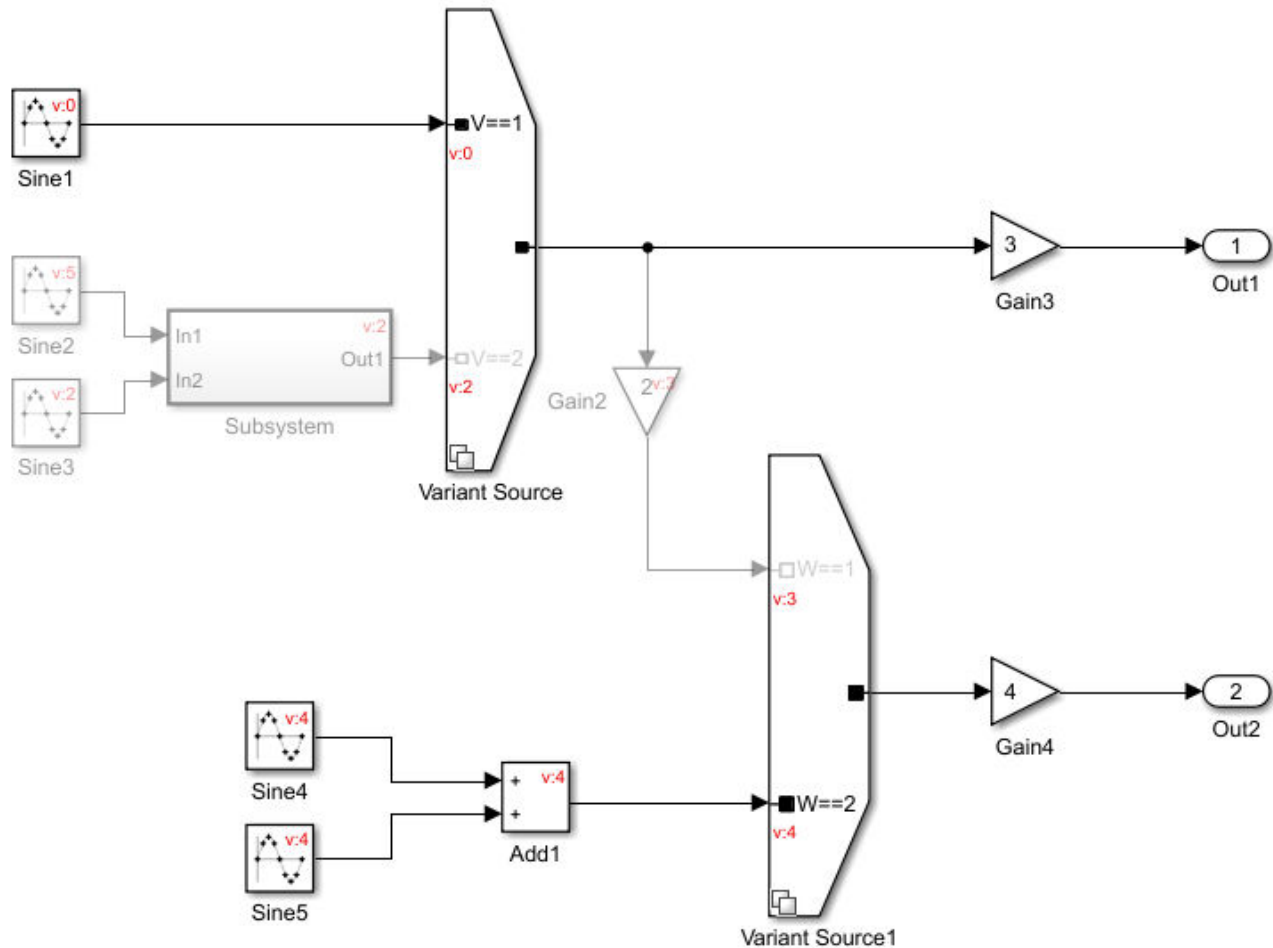
Additional guarding of global data for variant systems

In R2016a, for models that contained `Variant Source` or `Variant Sink` blocks, preprocessor conditionals surrounded global variable declarations for root inports and root outports.

In R2016b, preprocessor conditionals also surround most global variable declarations for `Dwork` vectors, signals, and states. The inclusion of preprocessor conditionals around these global variable declarations conserves RAM because the code is not compiled unless these global variables are part of the active variant.

For example, the model `inline_variants_example` contains three `Variant Source` blocks. `Variant Source` is in the top model and in `Subsystem`. `Variant Source1` is in the top model. `Subsystem` contains a `Unit Delay` block and a signal with a **Signal Name** of `sig1`.

For `Variant Source`, if the `Simulink.Parameter V` equals 1, the top port is active. If `V` equals 2, the bottom port is active. For `Variant Source1`, if the `Simulink.Parameter W` equals 1, the top port is active. If `W` equals 2, the bottom port is active.



Copyright 2015 The MathWorks Inc.
MathWorks Confidential

In R2016a, in the `inline_variants_example.h` file, for block signals and states, the code generator produced this code.

```

/* Block signals (auto storage) */
typedef struct {
    real_T VariantMerge_For_Variant_Source;
    real_T Sine3; /* '<Root>/Sine3' */
    real_T sig1; /* '<S1>/Unit Delay' */
} B_inline_variants_example_T;

/* Block states (auto storage) for system '<Root>' */
typedef struct {
    real_T delay1; /* '<S1>/Unit Delay' */
    int32_T counter; /* '<Root>/Sine1' */
    int32_T counter_f; /* '<Root>/Sine4' */
    int32_T counter_e; /* '<Root>/Sine5' */
    int32_T counter_fl; /* '<Root>/Sine3' */
}

```



```
} DW_inline_variants_example_T;
```

Preprocessor conditionals do not surround the global variable declarations.

In R2016b, in the `inline_variants_example.h` file, the code generator produces this code.

```
/* Block signals (auto storage) */
typedef struct {
    real_T VariantMerge_For_Variant_Source;
    real_T Sine3; /* '<Root>/Sine3' */

#if V == 2

    real_T sig1; /* '<S1>/Unit Delay' */

#define B_INLINE_VARIANTS_EXAMPLE_T_VARIANT_EXISTS
#endif /* V == 2 */

} B_inline_variants_example_T;

/* Block states (auto storage) for system '<Root>' */
typedef struct {

#if V == 2

    real_T delay1; /* '<S1>/Unit Delay' */

#define DW_INLINE_VARIANTS_EXAMPLE_T_VARIANT_EXISTS
#endif /* V == 2 */

    int32_T counter; /* '<Root>/Sine1' */
    int32_T counter_f; /* '<Root>/Sine4' */
    int32_T counter_e; /* '<Root>/Sine5' */
    int32_T counter_fl; /* '<Root>/Sine3' */
} DW_inline_variants_example_T;
```

For block signals and states, preprocessor conditionals do surround the global variable declarations. See Represent Variant Source and Sink Blocks in Generated Code.

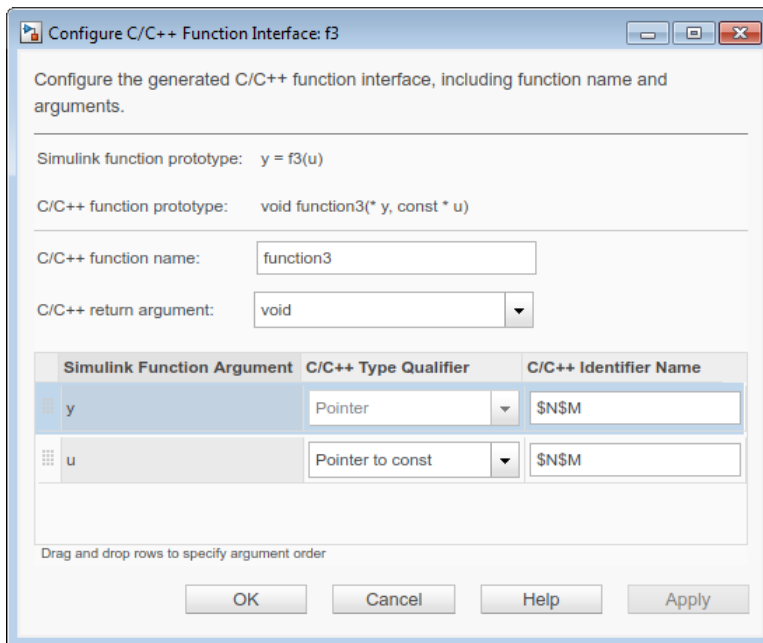
Data, Function, and File Definition

Simulink Function Code Interface: Configure generated C/C++ function interfaces for Simulink Function and Function Caller blocks

With Embedded Coder, you can customize generated C/C++ function interfaces. Function code interface configuration supports easier integration of generated code with functions or function calls in external code and customizations for coding standards or design requirements.

R2016b extends function code interface configuration to Simulink Function and Function Caller blocks. By opening a dialog box from a selected Simulink Function or Function Caller block, you can customize the C/C++ function prototype generated for that block. Your changes for the selected block also update other corresponding Simulink Function and Function Caller blocks in the model. You can change the generated C/C++ function name, and the names, type qualifiers, and order of function arguments. Your changes do not graphically alter the model and do not affect the Simulink function prototype defined in the block.

For example, you can configure a Simulink function prototype $y = f3(u)$ to generate a C/C++ function prototype such as `void function3(* y, const * u)`.



For more information, see [Configure Simulink Function Code Interface](#).

ERT default value for configuration parameter `ParameterTunabilityLossMsg`

In R2016b, the default value for the configuration parameter **Diagnostics > Data Validity > Detect loss of tunability** (programmatic name `ParameterTunabilityLossMsg`) for ERT-based targets is error. 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, **Detect loss of tunability** is set to error. If necessary, you can then change the value of **Detect loss of tunability**.

Compatibility Considerations

Your scripts that change code generation targets can unintentionally change the setting for **Detect loss of tunability**, causing unexpected errors during code generation.

Code Generation

Cross-Release Code Integration: Reuse code generated from earlier releases

Integrate R2016b generated code with existing:

- Shared code that is custom code or code that you generated from previous releases.
- Model code that you generated from previous releases (R2010a and later).

You avoid the cost of reverification because you reuse the existing code without modification.

You can use the `sharedCodeUpdate` function to collocate shared code from multiple source folders in an existing shared code folder. R2016b also provides the following configuration parameters on the **Configuration Parameters > All Parameters** tab:

- **Existing shared code** (`ExistingSharedCode`) — Specifies the folder that contains the shared code.
- **Use only existing shared code** (`UseOnlyExistingSharedCode`) — A diagnostic setting that determines whether the build process is permitted to generate new shared code that is not available from the specified folder.

You can use the `crossReleaseExport` and `crossReleaseImport` functions to integrate model code from previous releases when:

- The source models are single-rate, and set to generate nonreusable code with function prototype control (root-level Inport and Outport blocks are mapped to step function arguments).
- The model code has been produced by top-model and subsystem build processes.

Follow this workflow:

- 1 From a previous release, use the `crossReleaseExport` function to export model components. Add the function to the search path for that release with the following command:

```
addpath(fullfile(matlabRootForR2016b, 'toolbox', 'coder', 'xrelexport'));
```

- 2 With the `crossReleaseImport` function, import components from previous releases via software-in-the-loop (SIL) or processor-in-the-loop (PIL) blocks.
- 3 Insert the SIL or PIL blocks into your R2016b model.

When you run a model simulation, the simulation runs the previous release code through the SIL or PIL blocks.

When you build your model, new code is not generated for the components represented by the SIL or PIL blocks. The model code calls code generated by a previous release.

For more information, see:

- Cross-Release Shared Code Reuse
- Cross-Release Code Integration

Compound Operation Code Replacement: Replace "Multiply Shift Right Arithmetic" and "Multiply Divide" in generated code with a single custom operation

R2016b supports replacement of code for these compound operations with a single custom operation:

- Integer replacement of real, scalar multiplication followed by a shift right arithmetic operation (RTW_OP_MUL_SRA)
- Integer replacement of real, scalar multiplication followed by a division operation (RTW_OP_MULDIV)

ARXML import/export and C code generation for latest AUTOSAR 4.2 and 3.2 standard revisions

R2016b extends support of AUTOSAR schema versions 4.2 and 3.2 to include schema revisions 4.2.2 and 3.2.2. Embedded Coder supports the new schema revisions for import and export of arxml files and generation of AUTOSAR-compatible C code.

If you import schema 4.2.2 or 3.2.2 arxml code into Simulink, the arxml importer detects and uses the schema version and revision, and sets the schema version parameter in the model. For more information on schema import and export, see [Select an AUTOSAR Schema](#).

If you are developing an AUTOSAR software component based on AUTOSAR schema version 3.2, schema revision 3.2.2 allows you to include sender-receiver port end-to-end (E2E) protection, receiver port IsUpdated service, and port-based nonvolatile (NV) data communication in your component design.

Note This support is available to R2015b and R2016a Embedded Coder customers by installing the latest AUTOSAR support package for your release:

- R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.4 or later
 - R2016a Embedded Coder Support Package for AUTOSAR Standard, Version 16.1.1 or later
-

AUTOSAR code replacement library enhancements

R2016b improves the AUTOSAR code replacement library (CRL) by adding support for:

- Functions that perform multiplication followed by a shift right arithmetic operation.
- Arguments of type `struct` for the lookup table functions that perform prelookup and interpolation operations.

For more information, see [AUTOSAR Code Replacement Library](#).

Static code metrics report for C++ code

In R2016b, for a Simulink model with the target language set to C++, you can generate a **Static Code Metrics Report**. For more information, see [Generate Static Code Metrics Report for Simulink Model and Static Code Metrics](#).

Static code metrics data produced by Polyspace

In R2016b, for a Simulink model, Polyspace produces the data in the Static Code Metrics Report. The report contains the same information types in R2016b as it contained in R2016a. For a model, in the **Function Information** section of the Static Code Metrics Report, there can be differences between the Stack Size and Complexity in R2016b and R2016a.

Streamlined report pane for easier model configuration

In the Configuration Parameters dialog box, a streamlined **Code Generation > Report** pane displays only configuration parameters that you are most likely to use when configuring your model for code generation.

Compatibility Considerations

Following are the configuration parameters on the **Code Generation > Report** pane that are now only available on the **All Parameters** tab.

- **Code-to-model**
- **Model-to-code**
- **Eliminated / virtual blocks**
- **Traceable Simulink blocks**
- **Traceable Stateflow blocks**
- **Traceable MATLAB blocks**
- **Summarize which blocks triggered code replacements**

Improved traceability between model and code

In R2016b, these features enhance traceability between the model and generated code:

- Line-level traceability
- Highlighted code for multiple blocks or Stateflow objects

Previously, traceability between model and code depended on block comments in the generated code. If these comments were disabled, traceability was not available. In R2016b, Embedded Coder provides more precise model-to-code and code-to-model navigation with traceability to lines of code. Line-level traceability is enabled by default and is not dependent on block comments in the code.

From the code generation report, click a linked line of code to navigate to corresponding blocks in the model. From a block or blocks in your model, right-click the block and select **C/C++ Code > Navigate To C/C++ Code**. Highlighted lines of code in the code generation report correspond to your selected model blocks. Line-level traceability supports Simulink blocks, MATLAB function blocks, and Stateflow objects. The HTML traceability report and Microsoft® Excel® traceability matrix include line-level traceability information.

Note Line-level traceability is not available for some TLC-generated code and for code in header files.

In R2016b, you can select multiple blocks or Stateflow objects for model-to-code navigation. To highlight code for multiple objects:

- 1 To select contiguous blocks to trace, click and drag the cursor over the contiguous blocks. Alternatively, **Shift** + click to select the individual blocks.
- 2 From the selected blocks, right-click the blocks and select **C/C++ Code > Navigate To C/C++ Code**. The code generation report highlights lines of code that correspond to the selected blocks.

Code replacement enhancements

R2016b supports these code replacement enhancements:

- Integer replacement of real, scalar multiplication followed by a shift right arithmetic or division operation.
- When generating code for models that contain fixed-point calculations, improved integer code replacements for these saturating, real, scalar operations:
 - Addition, RTW_OP_ADD
 - Subtraction, RTW_OP_MINUS
 - Multiplication, RTW_OP_MUL
 - Division, RTW_OP_DIV
 - Data type conversion (cast), RTW_OP_CAST
- Improved detection of identity operations to avoid unnecessary replacements.

For more information, see [Code Replacement](#) and [Code Replacement Customization](#).

\$I macro changed for argument names used as input and output

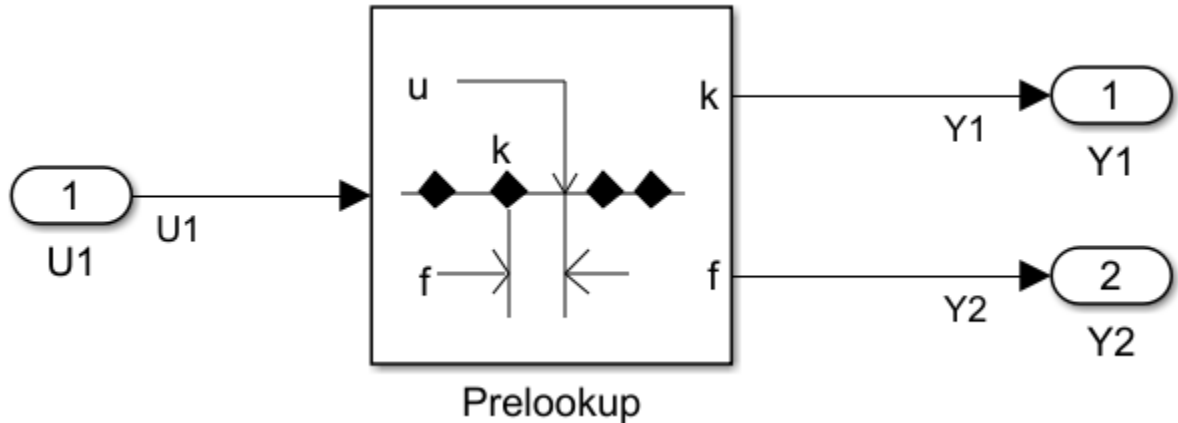
Previously, when you specified custom function argument names for a Simulink function by using the Subsystem method arguments parameter, for arguments that were input and output, the generated code inserted a `y` for the `$I` macro. In R2016b, the generated code inserts a `uy`.

Improved compliance with MISRA C:2012 Rules 10.1, 10.5, and 10.8

In R2016b, in the Configuration Parameters dialog box, on the **Code Generation > Code Style** tab, when you set the **Casting Modes** parameter to **Standards Compliant**, for more modeling patterns, the code generator produces code that is compliant with the Essential Type Model: Rules 10.1-10.8. See [MISRA C:2012 Directives and Rules](#).

MISRA C:2012 Rule 10.1

In R2016b, for operations involving Prelookup blocks, the code generator can produce code that is compliant with MISRA C:2012 Rule 10.1. For example, in the model `misra1`, in the Prelookup block parameters dialog box, the **Value** parameter is a vector with 256 elements.



In R2016a, in the `misra1.c` file, for the Prelookup block, the code generator produced this code:

```
if (u < (((int32_T)bp[0U]) << 16)) {
    bpIndex = 0U;
    *fraction = 0U;
}
```

This code is not compliant with MISRA C:2012 Rule 10.1 because the left operand of the `<<` operator is a signed integer, which is an inappropriate essential type.

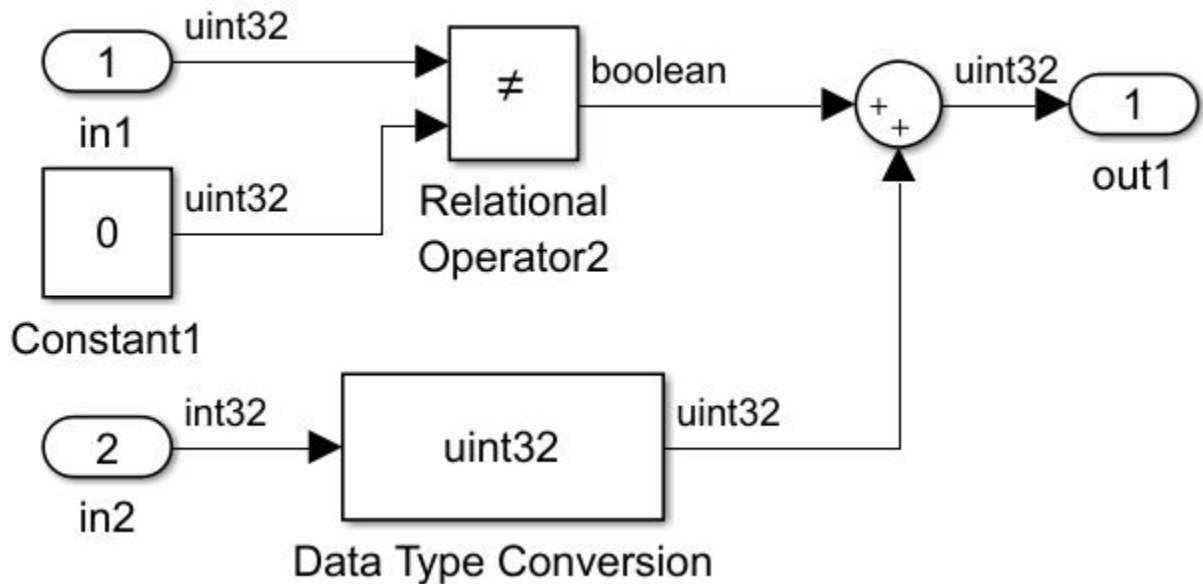
In R2016b, in the `misra1.c` file, for the Prelookup block, the code generator produces this code:

```
if (u < (((uint32_T)((uint32_T)((uint32_T)bp[0U]) << 16)))) {
    bpIndex = 0U;
    *fraction = 0U;
}
```

This code is compliant with MISRA C:2012 Rule 10.1 because the left operand is cast to an unsigned type.

MISRA C:2012 Rules 10.5 and 10.8

In R2016b, for more modeling patterns containing type conversions between different essential type categories, the code generator produces code that is compliant with MISRA C:2012 Rules 10.5 and 10.8. For example, in the model `misra2`, signals with data types Boolean and unsigned integer feed into a Sum block. The Sum block outputs a signal with a data type of unsigned integer.



In R2016a, in the `misra2.c` file, the code generator produced this code:

```
misra2_Y.out1 = ((uint32_T)(misra2_U.in1 != 0U)) + ((uint32_T)misra2_U.in2);
```

This code is not compliant with MISRA C:2012 Rules 10.5 and 10.8 because a Boolean, which is the output of the relational operator, is cast to an unsigned integer.

In R2016b, in the `misra2.c` file, the code generator produces this code:

```
misra2_Y.out1 = ((uint32_T)((misra2_U.in1 != 0U) ? 1 : 0)) + ((uint32_T)
    misra2_U.in2);
```

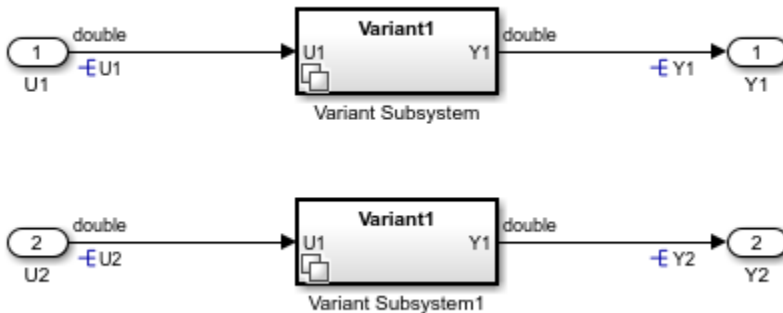
This code is compliant with MISRA C:2012 Rules 10.5 and 10.8 because the ternary operator prevents a cast from a Boolean to an unsigned integer.

Improved compliance with MISRA AC AGC Rule 12.6

In R2016a, for Variant Subsystem, Variant Source, and Variant Sink blocks, the preprocessor conditional that checked for only one active variant was not compliant with MISRA AC AGC Rule 12.6. In R2016b, this preprocessor conditional check is compliant with this rule. MISRA AC AGC Rule 12.6 states

Operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, or !).

For example, the model `misra_check` contains two Variant Subsystems, `Variant1` and `Variant2`. For `Variant Subsystem`, if the `Simulink.Parameter VC` equals 1, `Variant1` is active. If `VC` equals 2, `Variant2` is active. For `Variant Subsystem1`, if the `Simulink.Variant V1` evaluates to true, `Variant1` is active. If the `Simulink.Variant V2` evaluates to true, `Variant2` is active.



In R2016a, in the `preprocessor_check_types.h` file, the preprocessor conditionals that checked for just one active variant per subsystem were

```
/* Exactly one variant for '<Root>/Variant Subsystem' should be active */
#if (VC == 1) + (VC == 2) != 1
#error Exactly one variant for '<Root>/Variant Subsystem' should be active
#endif

/* Exactly one variant for '<Root>/Variant Subsystem1' should be active */
#if (V1) + (V2) != 1
#error Exactly one variant for '<Root>/Variant Subsystem1' should be active
#endif
#endif
```

According to the second sentence of Rule 12.6, `VC==1` and `VC==2` and `V1` and `V2` should not be added together because they are effectively Boolean expressions.

In R2016b, in the `preprocessor_check_types.h` file, the preprocessor conditionals that check for one active variant per subsystem are

```
/* Exactly one variant for '<Root>/Variant Subsystem' should be active */
#if ((VC == 1) ? 1 : 0) + ((VC == 2) ? 1 : 0) != 1
#error Exactly one variant for '<Root>/Variant Subsystem' should be active
#endif

/* Exactly one variant for '<Root>/Variant Subsystem1' should be active */
#if ((V1) ? 1 : 0) + ((V2) ? 1 : 0) != 1
#error Exactly one variant for '<Root>/Variant Subsystem1' should be active
#endif
#endif
```

The conditional checks contain ternary Boolean operators that do not violate MISRA Rule 12.6. See MISRA C:2004 and MISRA AC AGC Coding Rules.

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 for Folder Names with Spaces](#).

Deployment

Cortex-M7 Target Support Package: Generate code for STM32F746G-Discovery Board

You can use the Embedded Coder Support Package for STMicroelectronics Discovery Boards to generate code on the Cortex-M7 based STM32F746G-Discovery board.

To build your model for the STM32F746G-Discovery board, you can use the following blocks from the support package library:

- Audio Input
- Audio Output
- Analog Input
- Digital Read
- Digital Write
- I2C Master Read
- I2C Master Write
- PWM Output
- SPI Master Transfer
- SPI Register Read
- SPI Register Write

For more information, see Embedded Coder Support Package for STMicroelectronics Discovery Boards.

Added Embedded Coder Support Package for ARM Cortex-R Processors

You can use the Embedded Coder Support Package for ARM Cortex-R Processors to:

- Run executables with FreeRTOS on a Texas Instruments Hercules RM57Lx Launchpad, which uses a lockstep cached 330Mhz ARM Cortex-R5F based RM series MCU.
- Tune parameters on, and monitor data from, an executable running on the Texas Instruments Hercules RM57Lx Launchpad (External mode).
- Verify numeric accuracy and profile execution times using processor-in-the-loop (PIL) on the Texas Instruments Hercules RM57Lx Launchpad.
- Profile task and function execution times of executables running in real time on the Texas Instruments Hercules RM57Lx Launchpad.

To download and install this feature, perform the steps described in <https://www.mathworks.com/help/releases/R2016b/supportpkg/armcortexr/ug/install-support-for-arm-cortex-a-processors.html>.

For more information, see <https://www.mathworks.com/help/releases/R2016b/supportpkg/armcortexr/index.html>.

Improved External mode over serial communication

The external mode in Embedded Coder Support Package for Texas Instruments C2000 Processors feature is now improved with a faster serial communication protocol. The new protocol reduces data drop during data logging. With this change, increasing the baud rate also increases the data logging performance.

New blocks added to TI's C2000 support package

You can use eCAP, eQEP, CLA, and DAC blocks on TI's C2000™ F2837xS, F2837xD, and F2807x processors.

Use the eCAP block to capture input pin transitions or configure auxiliary pulse width modulator.

Use the eQEP block to interface with a linear or rotary incremental encoder.

Use CLA Trigger block to run code on Control Law Accelerator (CLA) co-processor available on F2803x, F2806x, F2837xS, F2837xD, and F2807x processors.

Use the DAC block to convert digital data to analog signal.

Change in name and the base product for the FRDM-K64F and the FRDM-KL25Z support packages

The base product for FRDM-K64F and FRDM-KL25Z support packages is changed from Embedded Coder to Simulink Coder. The two support package are now named as Simulink Coder Support Package for NXP™ FRDM-K64F Board and Simulink Coder Support Package for NXP FRDM-KL25Z Board respectively. For more information, see Simulink Coder Target Support Packages: Generate code for NXP Freedom boards and STMicroelectronics Nucleo boards.

Support for TI's C5000 DSPs has been removed

Embedded Coder support for TI's C5000 has been removed in R2016b. However, you can still generate code using Embedded Coder® by selecting TI's C5000 as the device vendor on the Hardware Implementation pane for ANSI-C. You can also create your own target optimizations using code replacement libraries. For more information, see Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®.

Support for TI's C6000 has been removed

Embedded Coder support for TI C6000™ has been removed in R2016b. However, you can still generate code using Embedded Coder by selecting TI's C6000™ as the device vendor on the Hardware Implementation pane for ANSI-C. You can also create your own target optimizations using code replacement libraries. For more information, see Optimize Generated Code By Developing and Using Code Replacement Libraries - Simulink®.

Support for Wind River VxWorks RTOS will be removed

Embedded Coder support for Wind River VxWorks RTOS will be removed in a future release. You will still be able to use Embedded Coder for Wind River VxWorks RTOS, but will need to manually integrate the generated code with hand written scheduler and drivers.

Support for idelink_ert.tlc will be removed

Support for idelink_ert.tlc will be removed in R2017a. C2000 processors will be supported only on ert.tlc workflow.

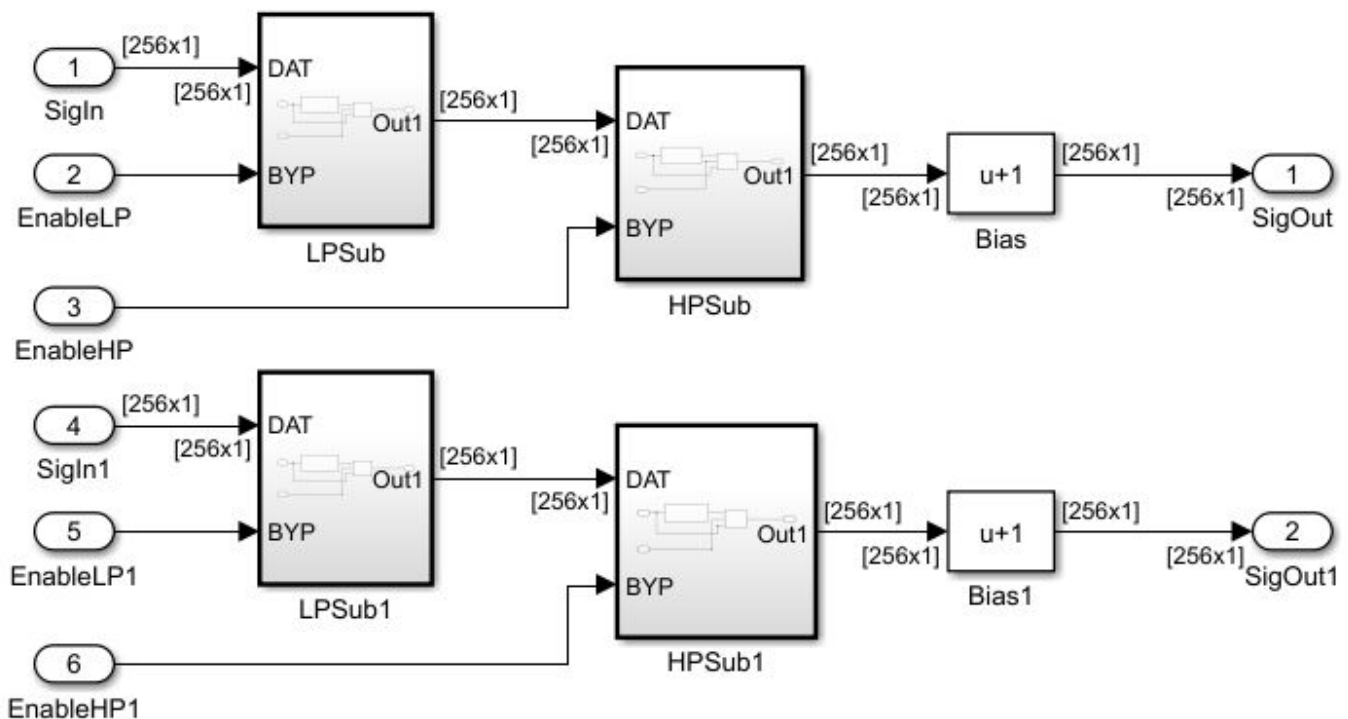
Performance

Data Reuse and Memory Reduction: Reuse global data for nonreusable subsystems and reduce data copies with user-specified buffers

Buffer reuse across nonreusable subsystems

In R2016b, for a model containing multiple nonreusable subsystems, the code generator can reuse a single global buffer. In the subsystem block parameters dialog box, on the **Code Generation** tab, a nonreusable subsystem has the **Function packaging** parameter set to **Nonreusable function**. The **Function interface** parameter is set to `void_void`. This optimization decreases data copies and memory consumption and increases code execution speed.

For example, the model `rtwdemo_automatic_global_reuse` contains four nonreusable subsystems. The inputs to each subsystem are arrays of size 256.



In R2016a, the `rtwdemo_automatic_global_reuse.h` file contained this code:

```
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
    DW_LowpassFilter LowpassFilter_pn; /* '<S4>/Lowpass Filter' */
    DW_LowpassFilter LowpassFilter_p; /* '<S3>/Lowpass Filter' */
    DW_HighpassFilter HighpassFilter_pn; /* '<S2>/Highpass Filter' */
    DW_HighpassFilter HighpassFilter_p; /* '<S1>/Highpass Filter' */
    real_T Switch[256]; /* '<S4>/Switch' */
    real_T Switch_i[256]; /* '<S3>/Switch' */
}
```

```

    real_T Switch_k[256];          /* '<S2>/Switch' */
    real_T Switch_f[256];          /* '<S1>/Switch' */
} DW;

```

For each nonreusable subsystem, the global structure DW contained an array. The array names were Switch, Switch_i, Switch_k, and Switch_f.

In R2016b, the `rtwdemo_automatic_global_reuse.h` file contains this code:

```

/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
    DW_LowpassFilter LowpassFilter_pn; /* '<S4>/Lowpass Filter' */
    DW_LowpassFilter LowpassFilter_p; /* '<S3>/Lowpass Filter' */
    DW_HighpassFilter HighpassFilter_pn; /* '<S2>/Highpass Filter' */
    DW_HighpassFilter HighpassFilter_p; /* '<S1>/Highpass Filter' */
    real_T Switch[256]; /* '<S1>/Switch' */
} DW;

```

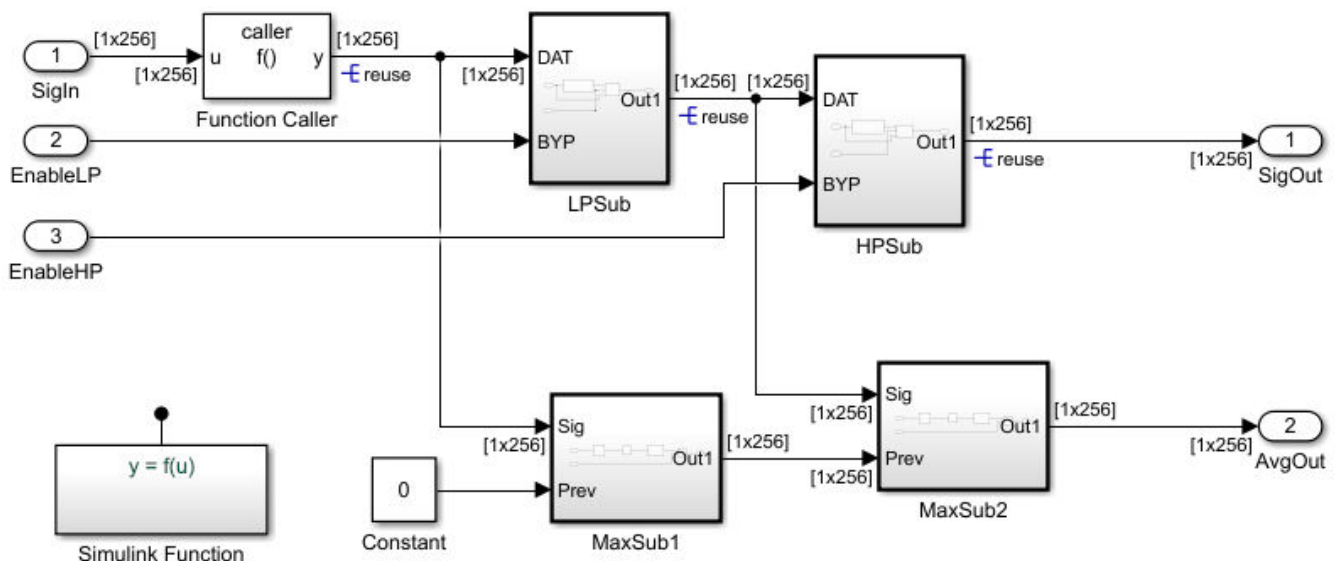
The global structure DW contains one array Switch for buffer reuse. Each nonreusable subsystem uses this array.

Buffer reuse for multiple signals in a path

For blocks and subsystems that form a path, if the input and output signals to these blocks and subsystems have the same reusable storage class specification, the code generator tries to reuse the signals in the generated code. This optimization decreases data copies and memory consumption and increases code execution speed.

For user-specified buffer reuse, blocks that modify a signal specified for reuse must execute before blocks that use the original signal value. In R2016a, sometimes the code generator changed the block operation order so that buffer reuse occurred.

In R2016b, the code generator performs better reordering of block operations, so that more instances of user-specified buffer reuse can occur. For example, in the model `rtwdemo_reusable_csc_scheduling`, the `Simulink.Signal` reuse is for buffer reuse. The four subsystems have nonreusable function packaging.



In R2016a, the `rtwdemo_reusable_csc_scheduling.c` file contained this code.

```
real_T reuse_1[256];
real_T reuse_0[256];
real_T reuse[256];
...
void rtwdemo_reusable_csc_scheduling_step(void)
{
    real_T rtb_MinMax_d[256];
    f(rtU.SigIn, reuse_1);
    LPSub();
    HPSub();
    MaxSub1(reuse_1, 0.0, rtb_MinMax_d);
    MaxSub2(reuse_0, rtb_MinMax_d, rtY.SigOut1);
}
```

For the Simulink.Signal `reuse`, there were three global variables: `reuse`, `reuse_0`, and `reuse_1`. The generated code could not use the same global variable in the four functions. `LPSub` and `HPSub` modified the signal value before `MaxSub1` and `MaxSub2` used it, and `MaxSub1` and `MaxSub2` had to use the original signal value.

In R2016b, the `rtwdemo_reusable_csc_scheduling.c` file contains this code:

```
real_T reuse[256];
...
void rtwdemo_reusable_csc_scheduling_step(void)
{
    f(rtU.SigIn, (&(reuse[0])));
    MaxSub1();
    LPSub();
    MaxSub2();
    HPSub();
}
```

For the Simulink.Signal `reuse`, there is one global variable `reuse`. The code generator can reuse this variable because calls to functions `MaxSub1` and `MaxSub2` happen before calls to functions `LPSub` and `HPSub`, respectively.

For more information, see [Specify Buffer Reuse for Multiple Signals in a Path](#).

Code Optimizations: Generate more efficient code with select-assign-iterator pattern and matrix padding operations

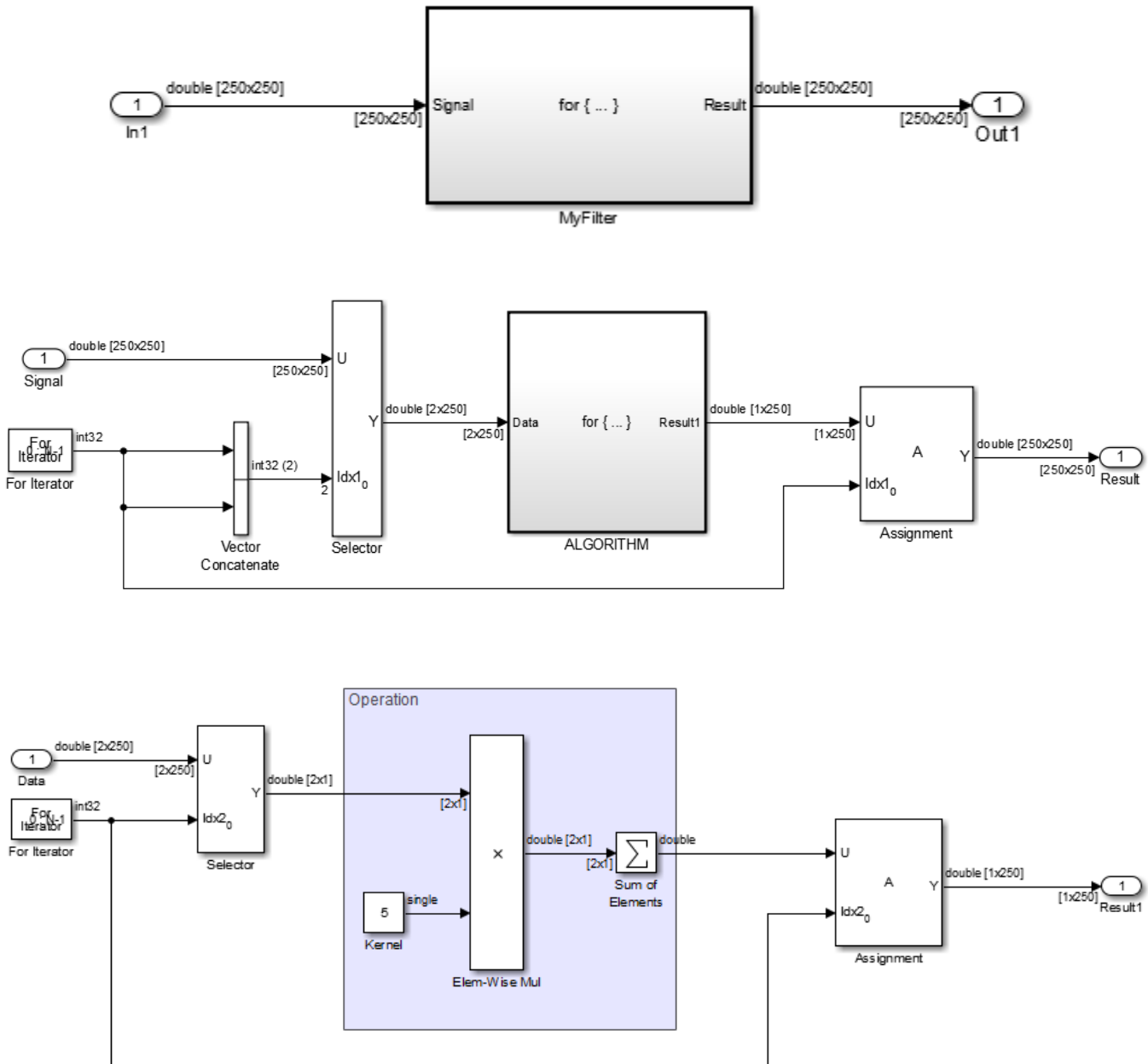
In R2016b, for iterative select assignment modeling patterns and matrix padding operations, there are buffer reductions in the generated code. This optimization reduces memory usage and increases code execution speed. Iterative select-assignment modeling patterns and matrix padding operations are useful in image processing.

Data copy reduction for select-assign-iterator modeling pattern

In R2016a, for a model that iteratively selects values from an input signal and assigns them to an output signal, there was an extra buffer in the generated code. In R2016b, the code generator eliminates this buffer.

For example, the `rtwdemo_optimize_nestedloops` model contains two select-assignment modeling patterns. One pattern is in the subsystem `MyFilter`. The other pattern is in the subsystem

ALGORITHM, which is nested in MyFilter. Both subsystems contain a for Iterator block, a Selector block, and an Assignment block.



In R2016a, the code generator produced this code.

```
void rtwdemo_optimize_nestedloops_step(void)
{
    int32_T s1_iter;
    real_T rtb_Selector[500];
```

```

int32_T s2_iter;
for (s1_iter = 0; s1_iter < 250; s1_iter++) {
    for (s2_iter = 0; s2_iter < 250; s2_iter++) {
        rtb_Selector[s2_iter << 1] = rtwdemo_optimize_nestedloops_U.In1[250 *
            s2_iter + s1_iter];
        rtb_Selector[1 + (s2_iter << 1)] = rtwdemo_optimize_nestedloops_U.In1[250 *
            s2_iter + s1_iter];
        rtwdemo_optimize_nestedloops_Y.Out1[s1_iter + 250 * s2_iter] =
            rtb_Selector[(s2_iter << 1) + 1] * 5.0 + rtb_Selector[s2_iter << 1] *
            5.0;
    }
}
}
}

```

The generated code contains the buffer `rtb_Selector[500]`.

In R2016b, the code generator produces this code.

```

void rtwdemo_optimize_nestedloops_step(void)
{
    int32_T s1_iter;
    int32_T s2_iter;
    for (s1_iter = 0; s1_iter < 250; s1_iter++) {
        for (s2_iter = 0; s2_iter < 250; s2_iter++) {
            rtwdemo_optimize_nestedloops_Y.Out1[s1_iter + 250 * s2_iter] =
                rtwdemo_optimize_nestedloops_U.In1[250 * s2_iter + s1_iter] * 5.0 +
                rtwdemo_optimize_nestedloops_U.In1[250 * s2_iter + s1_iter] * 5.0;
        }
    }
}
}

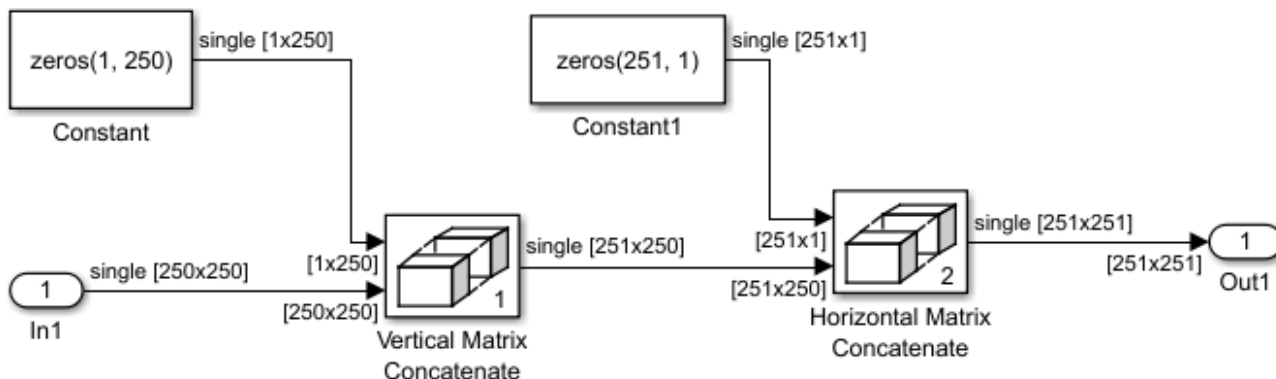
```

The generated code does not contain the `rtb_Selector[500]` buffer or the associated data copies.

Data copy reduction for matrix padding operations

In R2016a, for a model that uses Matrix Concatenate blocks to add rows and columns to a multidimensional input signal, there was an extra buffer in the generated code. In R2016b, the code generator eliminates this buffer.

For example, in the model `pattern_grow_matrix`, the Vertical Matrix Concatenate block adds a row of 250 zeros and the Horizontal Matrix Concatenate blocks adds a column of 250 zeros to a multidimensional input signal.



In R2016a, the code generator produced this code.

```
/* Model step function */
void pattern_grow_matrix_step(void)
{
    int32_T i;

    /* SignalConversion: '<Root>/ConcatBufferAtHorizontal Matrix ConcatenateIn1' */
    memset(&Y.Out1[0], 0, 251U * sizeof(real32_T));

    /* Concatenate: '<Root>/Vertical Matrix Concatenate' incorporates:
     * Constant: '<Root>/Constant'
     * Inport: '<Root>/In1'
     */
    for (i = 0; i < 250; i++) {
        B.fv0[251 * i] = 0.0F;
    }

    for (i = 0; i < 250; i++) {
        memcpy(&B.fv0[i * 251 + 1], &U.In1[i * 250], 250U * sizeof(real32_T));
    }

    memcpy(&Y.Out1[251], &B.fv0[0], 62750U * sizeof(real32_T));

    /* End of Concatenate: '<Root>/Vertical Matrix Concatenate' */
}
```

The code contained the buffer B.fv0.

In R2016b, the code generator produces this code.

```
/* Model step function */
void pattern_grow_matrix_step(void)
{
    int32_T i;

    /* SignalConversion: '<Root>/ConcatBufferAtHorizontal Matrix ConcatenateIn1' */
    memset(&Y.Out1[0], 0, 251U * sizeof(real32_T));

    /* Concatenate: '<Root>/Vertical Matrix Concatenate' incorporates:
     * Constant: '<Root>/Constant'
     * Inport: '<Roo>/In1'
     */
    for (i = 0; i < 250; i++) {
        Y.Out1[i * 251 + 251] = 0.0F;
    }

    for (i = 0; i < 250; i++) {
        memcpy(&Y.Out1[i * 251 + 252], &U.In1[i * 250], 250U * sizeof(real32_T));
    }

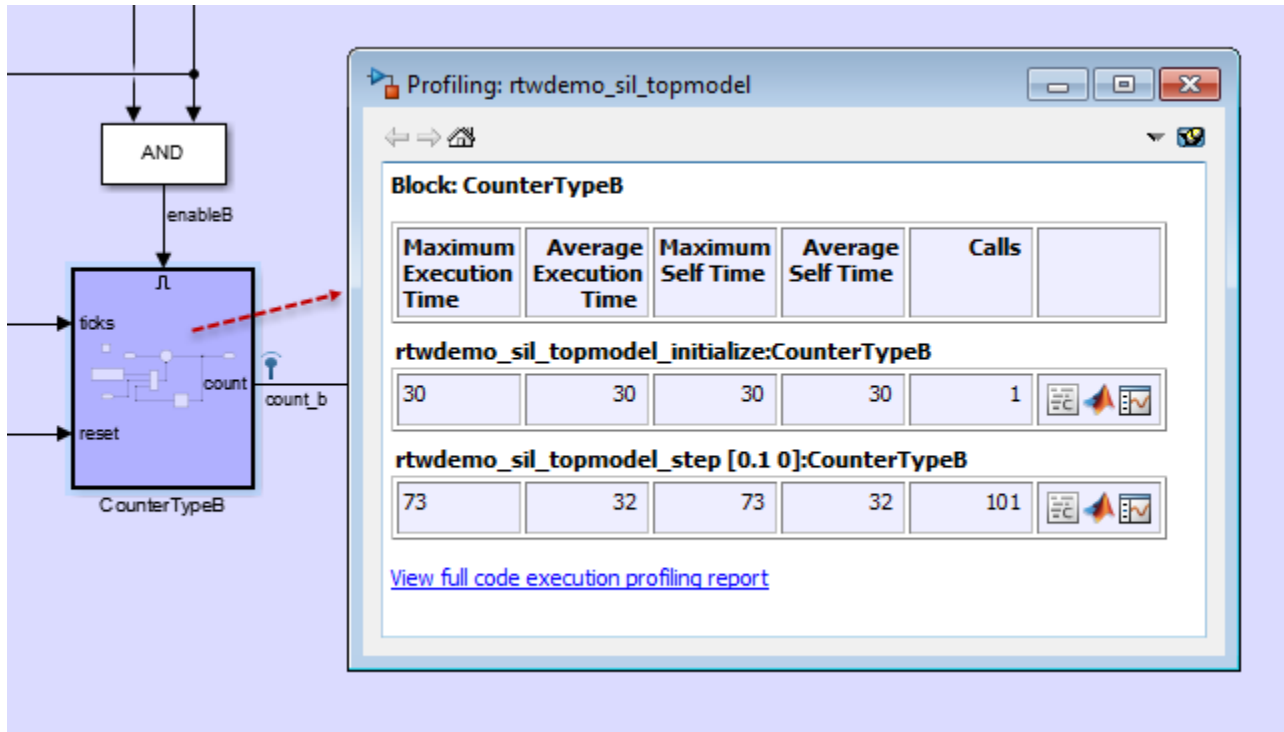
    /* End of Concatenate: '<Root>/Vertical Matrix Concatenate' */
}
```

The buffer, B.fv0, and the extra memcpy to B.fv0 are not in the generated code.

Display of code execution times for model component

R2016b provides improved viewing and analysis of code execution-time measurements that software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations produce. For example, at the end of a top-model SIL or PIL simulation, you can view execution-time metrics in a display window:

- For execution-time metrics from the top-level tasks, click the blue Simulink Editor background.
- For execution-time metrics from a profiled block, click the blue block.



The display window also provides access to:

- The complete profiling report, which provides execution-time metrics for profiled code sections.
- The profiled code section in the code generation report.
- The Simulation Data Inspector, which enables you to plot and compare execution-time measurements for the profiled code section.

For more information, see [View and Compare Code Execution Times](#).

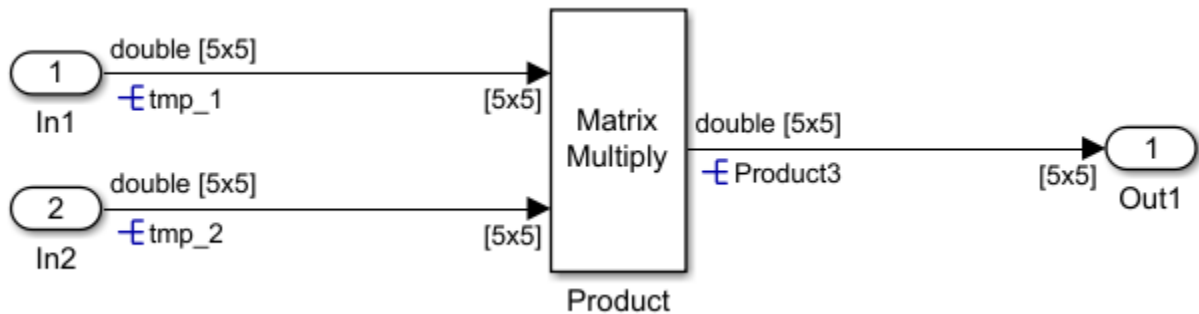
More efficient code for array element assignments

In R2016a, for a model that contained a Product block that performed matrix multiplication, the code generator assigned values to the product matrix one column at a time. In the generated code, the array element assignments occurred out of order.

In R2016b, the code generator performs a loop exchange, so that these assignments occur one row at a time. During a loop exchange, the code generator switches the order of iteration variables. In the generated code, the array element assignments occur in contiguous order.

When array element assignments occur in contiguous order, the CPU stores and accesses data in continuous memory. This optimization increases code execution speed because it improves cache efficiency.

For example, in the `matrix_multiply` model, the Product block processes the input vectors as matrices.



In R2016a, the code generator produced this code.

```

/* Product: '<Root>/Product' incorporates:
 * Inport: '<Root>/In1'
 * Inport: '<Root>/In2'
 */
for (i = 0; i < 5; i++) {
    for (i_0 = 0; i_0 < 5; i_0++) {
        matrix_multiply_B.Product3[i + 5 * i_0] = 0.0;
        for (i_1 = 0; i_1 < 5; i_1++) {
            matrix_multiply_B.Product3[i + 5 * i_0] += tmp_1[5 * i_1 + i] * tmp_2[5 *
                i_0 + i_1];
        }
    }
}

/* End of Product: '<Root>/Product' */
  
```

In the `matrix_multiply_step` function, element assignments to the array `matrix_multiply.Product3` occur in intervals of 5 (that is, one column at a time).

In R2016b, the code generator produces this code.

```

/* Product: '<Root>/Product' incorporates:
 * Inport: '<Root>/In1'
 * Inport: '<Root>/In2'
 */
for (i_0 = 0; i_0 < 5; i_0++) {
    for (i = 0; i < 5; i++) {
        matrix_multiply_B.Product3[i + 5 * i_0] = 0.0;
        for (i_1 = 0; i_1 < 5; i_1++) {
            matrix_multiply_B.Product3[i + 5 * i_0] += tmp_1[5 * i_1 + i] * tmp_2[5 *
                i_0 + i_1];
        }
    }
}
  
```

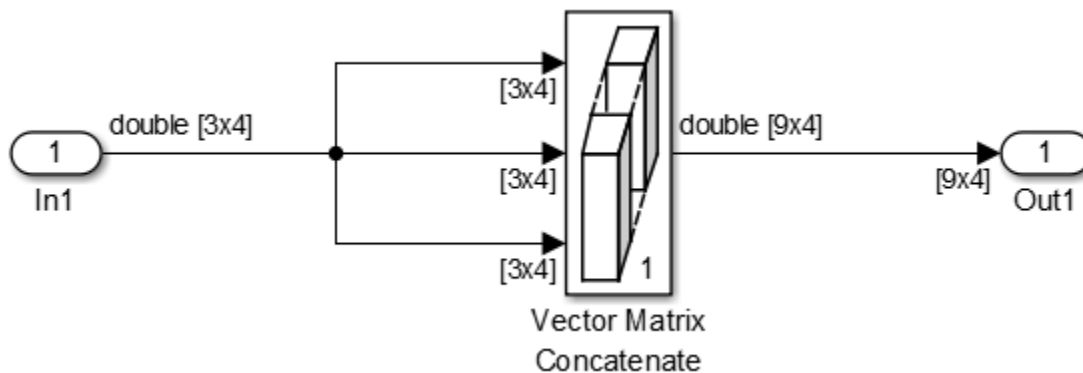
```
/* End of Product: '<Root>/Product' */
```

In the `matrix_multiply_step` function, element assignments to the array `matrix_multiply_B.Product3` occur in contiguous order (that is, one row at a time). The assignments occur in contiguous order because the code generator interchanges the iteration variables `i_0` and `i`. In R2016a, `i_0` is the iteration variable for the inner for loop, and `i` is the iteration variable for the outer for loop. In R2016b, `i_0` is the iteration variable for the outer for loop, and `i` is the iteration variable for the inner for loop.

Loop fusion for nested for loops

In R2016a, for a model that used a Concatenate block to concatenate input signals into a continuous multidimensional signal, the generated code contained separate nested for loops for each signal. In R2016b, the code generator combines these for loops. This optimization conserves ROM consumption and increases code execution speed.

For example, in the model `loop_fusion`, the Vertical Matrix Concatenate block concatenates three signals that each have dimensions 3x4 into one signal with dimensions 9x4.



In R2016a, the code generator produced this code.

```
void loop_fusion_step(void)
{
    int32_T i;
    int32_T i_0;
    for (i = 0; i < 4; i++) {
        for (i_0 = 0; i_0 < 3; i_0++) {
            loop_fusion_Y.Out1[i_0 + 9 * i] = loop_fusion_U.In1[3 * i + i_0];
        }
    }

    for (i = 0; i < 4; i++) {
        for (i_0 = 0; i_0 < 3; i_0++) {
            loop_fusion_Y.Out1[(i_0 + 9 * i) + 3] = loop_fusion_U.In1[3 * i + i_0];
        }
    }

    for (i = 0; i < 4; i++) {
        for (i_0 = 0; i_0 < 3; i_0++) {
            loop_fusion_Y.Out1[(i_0 + 9 * i) + 6] = loop_fusion_U.In1[3 * i + i_0];
        }
    }
}
```

```

    }
  }
}

```

There are three nested for loops.

In R2016b, the code generator produces this code.

```

void loop_fusion_step(void)
{
  int32_T i;
  int32_T i_0;
  for (i_0 = 0; i_0 < 4; i_0++) {
    for (i = 0; i < 3; i++) {
      loop_fusion_Y.Out1[i + 9 * i_0] = loop_fusion_U.In1[3 * i_0 + i];
      loop_fusion_Y.Out1[(i + 9 * i_0) + 3] = loop_fusion_U.In1[3 * i_0 + i];
      loop_fusion_Y.Out1[(i + 9 * i_0) + 6] = loop_fusion_U.In1[3 * i_0 + i];
    }
  }
}

```

There is one nested for loop.

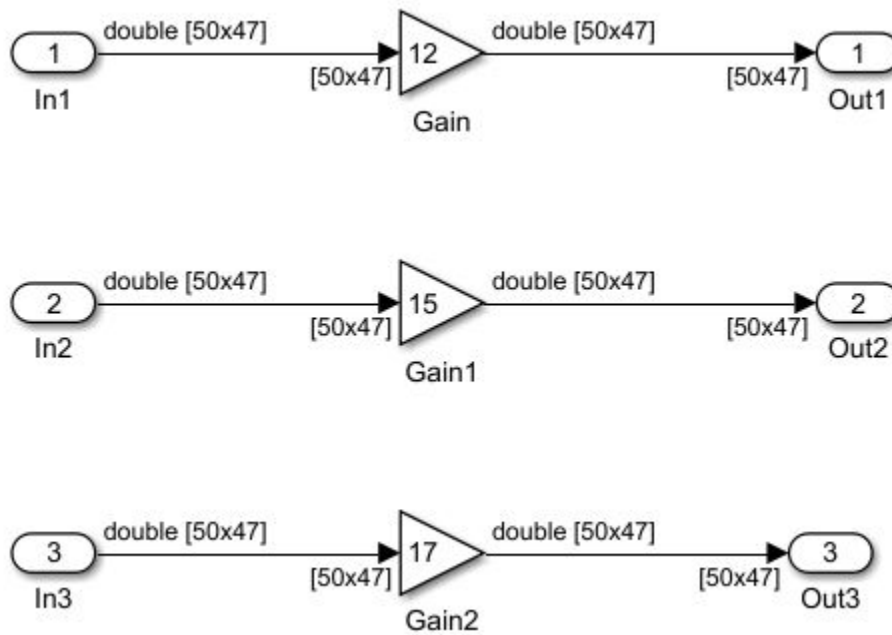
More efficient initialization code for root-level inports

Loop fusion in model_initialize function

In R2016a, the code generator did not fuse for loops that initialized data for root-level inports.

In R2016b, the code generator can fuse for loops that have the same upper bound value.

For example, in `loopfusionex`, for each Inport block, the **Port dimensions** parameter is [50 47].



In R2016a, in the `loopfusionex_initialize` function, the code generator produced this code.


```

/* external inputs */
{
  int32_T i;
  for (i = 0; i < 2350; i++) {
    loopfusionex_U.In1[i] = 0.0;
  }
}

{
  int32_T i;
  for (i = 0; i < 2350; i++) {
    loopfusionex_U.In2[i] = 0.0;
  }
}

{
  int32_T i;
  for (i = 0; i < 2350; i++) {
    loopfusionex_U.In3[i] = 0.0;
  }
}

```

For each Inport block, the generated code contained a separate `for` loop. The code generator generated code that initialized root inports to zero because in the Configuration Parameters dialog box, on the **Optimization** pane, the **Remove root level I/O zero initialization** parameter is not selected.

In R2016b, the code generator produces this code.

```

/* external inputs */
{
  int32_T i;
  for (i = 0; i < 2350; i++) {
    loopfusionex_U.In1[i] = 0.0;
    loopfusionex_U.In2[i] = 0.0;
    loopfusionex_U.In3[i] = 0.0;
  }
}

```

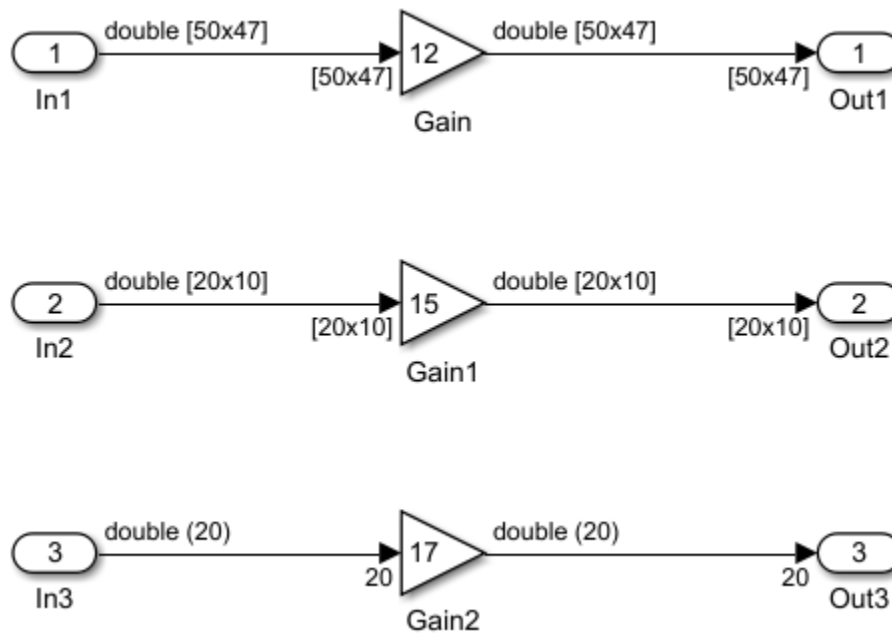
The generated code contains one `for` loop to initialize data for the three root-level inports.

One iteration variable for multiple for loops

In R2016a, in the `model_initialize` function, for `for` loops that initialized data for root-level Inport blocks, there was an iteration variable for each `for` loop.

In R2016b, for `for` loops that initialize data for root-level Inport blocks, there is one iteration variable for these `for` loops.

For example, in `for_loop_iterator`, for In1, In2, and In3, the **Port dimension** parameter is [50 47], [20 10], and 20, respectively. In the Configuration Parameters dialog box, on the **Optimization** pane, the **Remove root level I/O zero initialization** parameter is not selected.



In R2016a, in the `for_loop_iterator_initialize` function, the code generator produced this code:

```
/* external inputs */
{
    int32_T i;
    for (i = 0; i < 2350; i++) {
        for_loop_iterator_U.In1[i] = 0.0;
    }
}

{
    int32_T i;
    for (i = 0; i < 200; i++) {
        for_loop_iterator_U.In2[i] = 0.0;
    }
}

{
    int32_T i;
    for (i = 0; i < 20; i++) {
        for_loop_iterator_U.In3[i] = 0.0;
    }
}
```

The iteration variable `i` was declared three times—once for each for loop.

In R2016b, in the `for_loop_iterator_initialize` function, the code generator produces this code:

```
/* external inputs */
{
    int32_T i;
    for (i = 0; i < 2350; i++) {
```

```

    for_loop_iterator_U.In1[i] = 0.0;
}

for (i = 0; i < 200; i++) {
    for_loop_iterator_U.In2[i] = 0.0;
}

for (i = 0; i < 20; i++) {
    for_loop_iterator_U.In3[i] = 0.0;
}
}

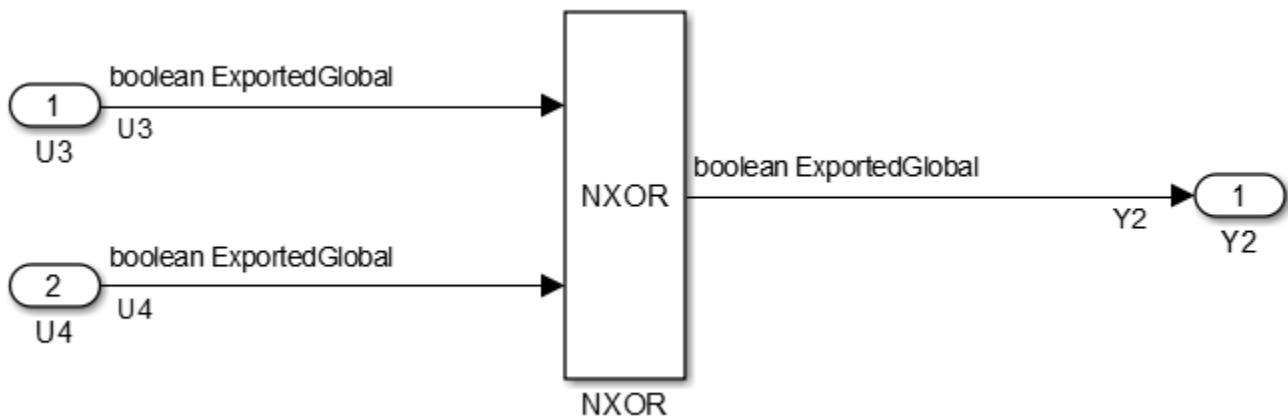
```

The generated code declares the iteration variable, *i*, once.

More efficient code for Boolean expressions

In R2016b, for a model containing a Logic block with the **Operator** parameter set to NXOR, the code generator removes an equality operator. Removing this operator makes the generated code less complex and more efficient.

For example, the model `nxor_example` contains two Inport blocks that connect to a Logic block. In the Inport block parameters dialog box, on the **Signal Attributes** tab, the **Data type** parameter is set to Boolean.



In R2016a, in the `nxor_example_step` function, the code generator produced this code:

```

void nxor_example_step(void)
{
    Y2 = !(U3 != U4);
}

```

The generated code contained two equality operators.

In R2016b, the code generator produces this code:

```

void nxor_example_step(void)
{
    Y2 = (U3 == U4);
}

```

The generated code contains one equality operator.

Verification

Verification of `size_t` and `ptrdiff_t` hardware settings

In R2016b, the **Configuration Parameters > Hardware Implementation** pane provides settings for the ANSI C data types `size_t` and `ptrdiff_t`. At the start of a processor-in-the-loop (PIL) simulation, the software verifies the settings with reference to the target hardware.

Verification of PIL target connectivity configuration

Through the `piltest` function, you can use a test suite to verify your custom processor-in-the-loop (PIL) target connectivity configuration. Verify the target connectivity configuration early and independently of your model development and code generation.

For more information, see [Create PIL Target Connectivity Configuration](#).

Signal range checking in SIL and PIL simulations

Top-model and Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations support the **Configuration Parameters > Diagnostics > Data Validity > Simulation range checking** (`SignalRangeChecking`) diagnostic. With this diagnostic, you can detect mismatches between model and generated code simulations that arise when you specify the code optimization configuration parameter, `UseSpecifiedMinMax`. The range checking applies to only root-level I/O signals of the SIL or PIL component.

SIL and PIL block support for Simulink Function and Function Caller blocks

You can run simulations with SIL and PIL blocks that you create from subsystems containing Simulink Function or Function Caller blocks. Function calls across the SIL or PIL block boundary are not supported.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.10

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Export data by using ExportedDefine storage class

In R2016a, when you generate C/C++ code from MATLAB code, you can use an `ExportedDefine` storage class to declare global variables with `#define` directives. The code generator emits these directives to the `entryfcn.h` header file. `entryfcn.h` is the name of the first entry-point function.

To assign the `ExportedDefine` storage class to a global variable, in your MATLAB code, use the `coder.storageClass` function. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generation software recognize `coder.storageClass` calls.

The syntax for `coder.storageClass` is:

```
coder.storageClass(var_name, storage_class)
```

`var_name` is the name of a global variable. Specify `var_name` as a constant string. Specify `storage_class` as `'ExportedDefine'`. For example, `coder.StorageClass('g', 'ExportedDefine')` assigns the `ExportedDefine` storage class to the global variable `g`. To assign the `ExportedDefine` storage class to a global variable, the global variable must be only read and not written to in the code.

SIL execution returns standard output and standard error streams

During a SIL execution, the SIL application redirects the `stdout` and `stderr` streams. When the application terminates, the MATLAB Command Window now displays the information from the redirected streams.

The SIL application also provides a basic signal handler, which captures the POSIX® signals `SIGFPE`, `SIGILL`, `SIGABRT`, and `SIGSEV`. The SIL application includes the file `signal.h` for the signal handler.

The information from the redirected streams can help you to debug SIL applications that fail before the execution is complete. For example, you can view:

- Output from `printf` statements in your code.
- If you enable run-time error detection, messages sent to `stderr`.
- Some low-level system messages.

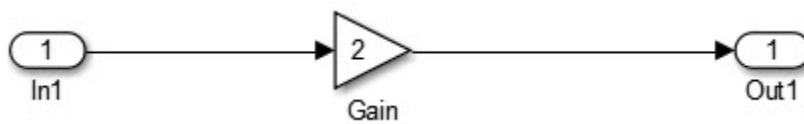
For more information, see [Debug SIL Execution](#).

Model Architecture and Design

Compile-Time Dimensions: Generate compiler directives (#define) for implementing signal dimensions

Previously, Simulink treated signal and parameter dimension specifications as numeric constants. In R2016a, you can use a `Simulink.Parameter` object as a symbol in a MATLAB expression to represent a dimension value. During simulation, Simulink propagates dimension symbols throughout the model and preserves these symbols in the propagated signal dimensions.

For example, in the model `sym_dim_ex`, the **Port Dimensions** parameter of `In1` is the `Simulink.Parameter D`.



The `sym_dim_ex.c` file contains the following code:

```

for (i = 0; i < D; i++) {
    sym_dim_ex_Y.Out1[i] = 2.0 * sym_dim_ex_U.In1[i];
}
  
```

In a header file, a macro defines the symbol `D`:

```
#define D 100
```

For the same model, if you change the value of `D`, the generated code remains the same except for the definition of `D`:

```
#define D 200
```

When you use symbols instead of numeric constants for dimension specifications, you can compile the same generated code into multiple applications of different sizes. When you simulate the model, you validate the behavior of the generated code for a set of symbolic dimension values. Change the values of the `Simulink.Parameter` objects that define the dimension symbols and simulate the model with the new values to check that the new values are valid.

For more information on how to specify dimensions with `Simulink.Parameters`, see [Implement Dimension Variants for Array Sizes in Generated Code](#)

The dimension variants feature is on by default. You can turn off this feature by clearing the `Allow symbolic dimension specification` parameter on the **All Parameters** tab of the Configuration Parameters dialog box.

Compile-Time Variants: Generate compiler directives (#if) for variant choices specified with Variant Source and Variant Sink blocks

Previously, you used model variants and variant subsystems to make parts of a model conditional. Preprocessor conditionals controlled which child subsystem of the variant subsystem or which child model of the model variant was active in the generated code.

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 can generate code for only the active variant choice or generate preprocessor conditionals that defer the choice of active variant until compilation time. You can generate preprocessor conditionals that allow for no active variant choice. For more information, see [Represent Variant Source and Sink Blocks in Generated Code](#)

C++ Code Generation: Use referenced models with multitasking, export-functions, and virtual buses

Previously, code generation for the C++ model class interface was limited to single tasking mode for model reference targets and non-virtual buses for crossing model boundaries. Also, the Export Function feature could not generate code for the C++ model class interface.

The C++ model class interface support in this release provides multitasking mode for model reference target, provides virtual bus for crossing model boundaries, and supports export function-call subsystems. For more information about using exported functions with a C++ model class interface, see [Export Function-Call Subsystems](#). For more information about multitasking and virtual buses with C++ code generation, see “Default style C++ interface replaces the void-void style C++ interface” on page 14-13.

MISRA C:2012 Compliance: Check block names and Assignment blocks by using the Model Advisor

To improve MISRA C:2012 compliance, in the Model Advisor **By Task > Modeling Guidelines for MISRA C:2012** folder, you can run the following new checks.

Check	Description	Addresses MISRA C:2012 Rule
Check for unsupported block names	Identifies block names that contain a / character.	3.1
Check usage of Assignment blocks	Identifies Assignment blocks with incomplete array initialization that do not have block parameter Action if any output element is not assigned set to Error or Warning.	9.1

For information about MISRA C versions and updates, see [MISRA C Guidelines](#)

AUTOSAR Round Trip: Automate model additions for update and merge of ARXML files

Simulink provides the ability to merge AUTOSAR authoring tool changes into a model as part of round-trip iterations. R2016a adds more automation and better reporting to the merge process. The software:

- Automates Simulink block additions. In the updated model, green highlighting identifies the added blocks.
- Lists required Simulink block deletions. In the updated model, red highlighting identifies the blocks to delete.

For more information, see Import AUTOSAR Software Component Updates.

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.2 or later.

R2016a provides many other enhancements to Simulink modeling of AUTOSAR elements and AUTOSAR code generation. For more information, see:

- Under Model Architecture and Design:
 - “Variants in AUTOSAR component modeling” on page 14-5
 - “AUTOSAR DataReceivedEvents for receiver ports in ImplicitReceive data access mode” on page 14-7
 - “AUTOSAR LiteralPrefix for enumerations in IncludedDataTypeSets” on page 14-7
 - “Programmatic validation and synchronization of AUTOSAR model configurations” on page 14-7
- Under Code Generation:
 - “AUTOSAR arxml round trip” on page 14-14
 - “Improved AUTOSAR library support for Mfx functions” on page 14-15
 - “AUTOSAR target no longer supports building wrapper subsystem as AUTOSAR SW-Component” on page 14-15

Comment change in generated code

In R2016a, for models containing hierarchical model elements such as a conditionally executed subsystem and either a reusable subsystem, a Stateflow Chart, or a model reference, there is a comment change in the generated code.

In R2015b, for the code that sets the initial conditions of block states inside these hierarchical model elements, the comment states `Initial Conditions` or `InitializeConditions`.

In R2016a, the comment states `System initialize` or `SystemInitialize`.

Variants in AUTOSAR component modeling

R2016a enhances AUTOSAR component modeling with modeling support for:

- AUTOSAR variants in ports and runnables
- AUTOSAR variants in array sizes

AUTOSAR variants in ports and runnables

AUTOSAR software components can use variants to enable or disable AUTOSAR elements, such as ports and runnables, based on defined conditions. Embedded Coder now supports modeling AUTOSAR variants in ports and runnables.

In Simulink, you can:

- Import AUTOSAR ports and runnables with variation points.

The `arxml` importer creates the required model elements, including workspace variables for modeling with variants, `Variant Sink` blocks, and `Variant Source` blocks to propagate variant conditions.

- Model AUTOSAR ports and runnables with variation points.
 - To define variant condition logic, use `Simulink.Variant` data objects.
 - To represent AUTOSAR system constants, use `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
 - To propagate variant conditions for the AUTOSAR elements, use `Variant Sink` and `Variant Source` blocks.
- Run validation on the AUTOSAR configuration. The validation software checks that variant conditions on Simulink blocks match the designed behavior from the imported `arxml` code.
- Export previously imported AUTOSAR ports and runnables with variation points.

For more information, see [Model AUTOSAR Variants and Configure AUTOSAR Variants in Ports and Runnables](#).

AUTOSAR variants in array sizes

AUTOSAR software components can flexibly specify the dimensions of an AUTOSAR element, such as a port, by using a symbolic reference to a system constant. The system constant defines the array size of the port data type.

Embedded Coder now supports modeling AUTOSAR variants in array sizes.

In Simulink, you can:

- Import AUTOSAR elements with variant array sizes.
 - The `arxml` importer creates the required model elements, including `AUTOSAR.Parameter` data objects with storage class `SystemConstant`, to represent the array size values.
 - Each block created by the importer to represent an AUTOSAR element with variant array sizes references `AUTOSAR.Parameter` data objects to define its dimensions.
- Model AUTOSAR elements with variant array sizes.
 - Create blocks that represent AUTOSAR elements.
 - To represent array size values, add `AUTOSAR.Parameter` data objects with storage class `SystemConstant`.
 - To specify array size for an AUTOSAR element, reference an `AUTOSAR.Parameter` data object.
- Modify array size values in system constants and simulate the model, without regenerating code for simulation.

- Generate C and arxml code with symbols corresponding to variant array sizes.

For more information, see Variants in Array Sizes and Configure AUTOSAR Variants in Array Sizes.

AUTOSAR DataReceivedEvents for receiver ports in ImplicitReceive data access mode

R2016a enhances AUTOSAR sender-receiver modeling with support for DataReceivedEvents for receiver ports in ImplicitReceive data access mode. Previously, the software supported DataReceivedEvents for receiver ports only in ExplicitReceive, QueuedExplicitReceive, and EndToEndRead data access modes.

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.0 or later.

AUTOSAR LiteralPrefix for enumerations in IncludedDataTypeSets

The arxml importer can now import AUTOSAR LiteralPrefixes defined in IncludedDataTypeSets. The software adds LiteralPrefixes to Simulink enumerated data types generated by the importer.

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.2 or later.

Programmatic validation and synchronization of AUTOSAR model configurations

R2016a adds MATLAB functions for validating and synchronizing AUTOSAR model configurations:

- `autosar.api.validateModel` — Validate AUTOSAR properties and Simulink to AUTOSAR mapping of specified model.
- `autosar.api.syncModel` — Synchronize Simulink to AUTOSAR mapping of specified model with Simulink block modifications.

The functions are equivalent to using the **Validate**  and **Synchronize**  icons in the graphical views of an AUTOSAR configuration.

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.1 or later.

Data, Function, and File Definition

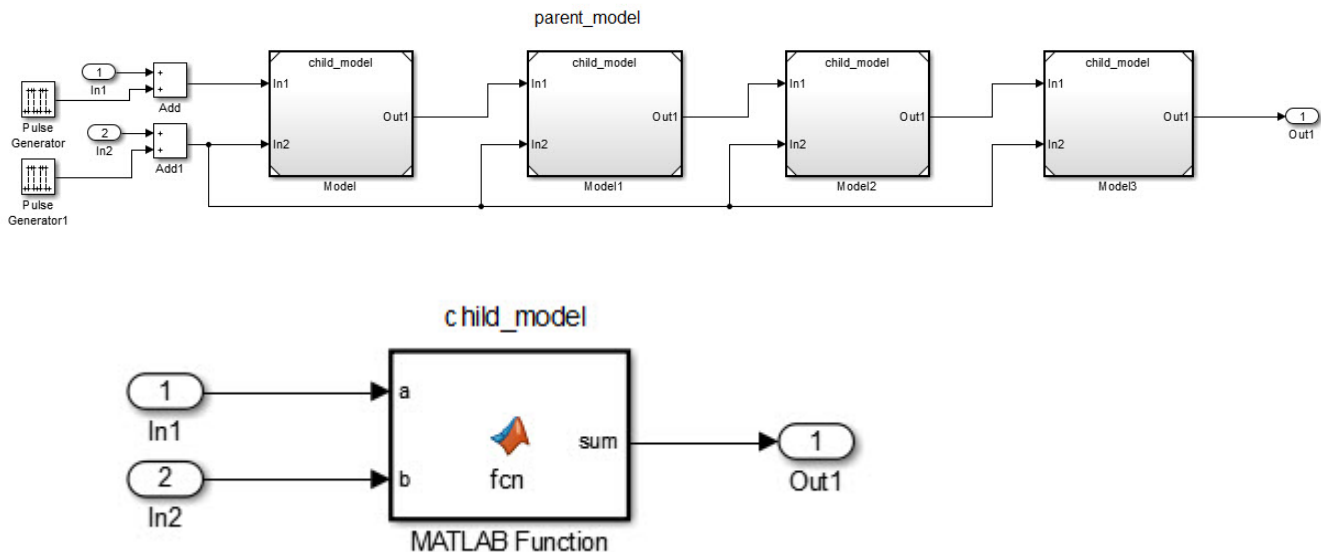
In/Out Arguments: Specify same variable name for in/out arguments of MATLAB Function and Model blocks

Buffer reuse across Model blocks

Previously, the code generator tried to reuse buffers for a pair of model step function input and output ports that were assigned the same argument name using function prototype control. This optimization decreases RAM/ROM consumption because there are less data copies and global variables in the generated code.

In R2016a, the code generator tries to reuse the input and output buffers of Model blocks.

For example, the model `parent_model` contains three copies of the model `child_model`.



In R2015b and R2016a, the code generator produces the following code in `child_model.cpp`:

```
void bottommodel::step(real_T arg_Inout1[9], const real_T arg_In2[9])
{
    int32_T i;
    for (i = 0; i < 9; i++) {
        arg_Inout1[i] += arg_In2[i];
    }
}
```

The generated code uses the same buffer `arg_Inout1` for the input `In1` and the output `Out1`.

In R2015b, the code generator produces this code in `parent_model.cpp`:

```
void topmodel::step()
{
    real_T rtb_Model[9];
    real_T rtb_Model1[9];
```

```

real_T rtb_Model2[9];
int32_T rtb_PulseGenerator1;
real_T rtb_Add[9];
real_T rtb_Add1[9];
int32_T i;
...
for (i = 0; i < 9; i++) {
    rtb_Add[i] = parent_model_U.In1[i] + (real_T)rtb_PulseGenerator1;
}

...

for (i = 0; i < 9; i++) {
    rtb_Add1[i] = parent_model_U.In2[i] + (real_T)rtb_PulseGenerator1;
}

(void) memcpy(&rtb_Model[0], &rtb_Add[0],
              9*sizeof(real_T));
ModelMDL0BJ1.step(&rtb_Model[0], &rtb_Add1[0]);
(void) memcpy(&rtb_Model1[0], &rtb_Model[0],
              9*sizeof(real_T));
Model1MDL0BJ2.step(&rtb_Model1[0], &rtb_Add1[0]);
(void) memcpy(&rtb_Model2[0], &rtb_Model1[0],
              9*sizeof(real_T));
Model2MDL0BJ3.step(&rtb_Model2[0], &rtb_Add1[0]);
(void) memcpy(&parent_model_Y.Out1[0], &rtb_Model2[0],
              9*sizeof(real_T));
Model3MDL0BJ4.step(&parent_model_Y.Out1[0], &rtb_Add1[0]);
}

```

The code generator does not reuse the output of one child model as the input to the next child model. Instead, there is a full array data copy prior to each call to the model step function.

In R2016a, the code generator produces the following code:

```

void topmodel::step()
{
    int32_T rtb_PulseGenerator1;
    real_T rtb_Model2[9];
    real_T rtb_Add1[9];
    int32_T i;

    for (i = 0; i < 9; i++) {
        rtb_Model2[i] = parent_model_U.In1[i] + (real_T)rtb_PulseGenerator1;
    }

    ...

    for (i = 0; i < 9; i++) {
        rtb_Add1[i] = parent_model_U.In2[i] + (real_T)rtb_PulseGenerator1;
    }

    ModelMDL0BJ1.step(&rtb_Model2[0], &rtb_Add1[0]);
    Model1MDL0BJ2.step(&rtb_Model2[0], &rtb_Add1[0]);
    Model2MDL0BJ3.step(&rtb_Model2[0], &rtb_Add1[0]);
    memcpy(&parent_model_Y.Out1[0], &rtb_Model2[0], (uint32_T)(9U * sizeof(real_T)));
    Model3MDL0BJ4.step(&parent_model_Y.Out1[0], &rtb_Add1[0]);
}

```

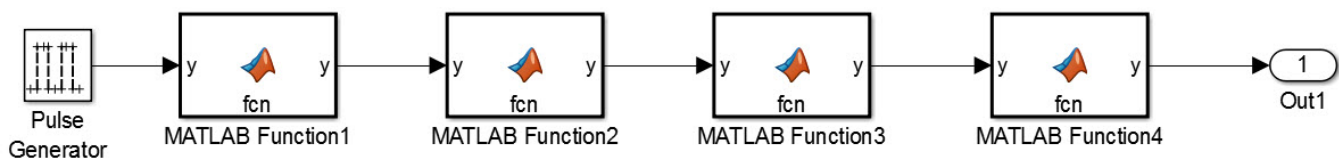
The code generator reuses the output of each child model as the input to the next child model. As a result, there are three less local arrays and four less full array data copies in the generated code.

To configure model step function I/O arguments to allow buffer reuse, use either C function prototype control or C++ class interface control. When generating C code, there can be only one instance of the same Model Reference block in the parent model. When generating C++ code, the same Model Reference block can occur multiple times in the parent model. For more information, see [Combine I/O Arguments in Model Step Interface](#)

Buffer reuse across MATLAB Function blocks

In R2016a, you can specify the same variable name for the input and output of a MATLAB Function block. If you connect multiple MATLAB Function blocks with the same variable name for the input and output arguments, the code generator tries to reuse the output of one MATLAB Function block as the input to the next MATLAB Function block. This optimization conserves RAM/ROM consumption by reducing the number of local variables and data copies in the generated code.

For example, the model named `mb_reuse` contains four MATLAB Function blocks.



Each MATLAB Function block contains the following code:

```
function y = fcn(y)
y = y+4;
```

In R2016a, the code generator produces this code:

```
void mb_reuse_MATLABFunction1(real_T *rty_y)
{
    *rty_y += 4.0;
}
void mb_reuse_step(void)
{
    real_T rtb_y_p;
    rtb_y_p = (mb_reuse_DW.clockTickCounter < mb_reuse_P.PulseGenerator_Duty) &&
        (mb_reuse_DW.clockTickCounter >= 0) ? mb_reuse_P.PulseGenerator_Amp : 0.0;
    if (mb_reuse_DW.clockTickCounter >= mb_reuse_P.PulseGenerator_Period - 1.0) {
        mb_reuse_DW.clockTickCounter = 0;
    } else {
        mb_reuse_DW.clockTickCounter++;
    }

    mb_reuse_MATLABFunction1(&rtb_y_p);
    mb_reuse_MATLABFunction1(&rtb_y_p);
    mb_reuse_MATLABFunction1(&rtb_y_p);
    mb_reuse_Y.Out1 = rtb_y_p;
    mb_reuse_MATLABFunction1(&mb_reuse_Y.Out1);
}
```


The code generator reuses the variable `rtb_y_p` for the input and output arguments of each MATLAB Function block.

On the **Code Generation** tab in the subsystem Block Parameters dialog box, if **Function packaging** is set to `Nonreusable function` and **Function interface** is set to `Allows arguments`, the code generator cannot reuse the input and output buffers.

Custom Storage Class Type AccessFunction

In R2016a, you can use the Custom Storage Class Designer to create custom storage classes of the new type `AccessFunction`. These custom storage classes access data in the generated code through functions whose custom definitions you provide. The built-in custom storage class `GetSet` from the package `Simulink` now uses this type.

You can configure these attributes as instance-specific or as common to all data items that use the custom storage class:

- For your `get` and `set` functions, a naming scheme that uses the name of each data item
- The name of the header file in which you provide the function prototypes

When you copy the `+SimulinkDemos` package to create your own data class package, you can modify the definition of the custom storage class `GetSet` by using the Custom Storage Class Designer.

For more information about the built-in custom storage class `GetSet`, see [Access Data Through Functions with Custom Storage Class GetSet](#). To create custom storage classes, see [Design Custom Storage Classes and Memory Sections](#).

Creation of custom storage classes for macros defined by compiler options

Previously, the built-in custom storage class `CompilerFlag` used the type `Other`. In R2016a, `CompilerFlag` uses the type `Unstructured` and represents an imported macro that does not require a header file.

To import macros that you define by configuring compiler options, you can use `CompilerFlag` or create your own custom storage class. In the Custom Storage Class Designer:

- Set **Data initialization** to `Macro`.
- Set **Data scope** to `Imported`.
- Set **Header file** to `Specify`. Omit the header file name.

For more information, see [Design Custom Storage Classes and Memory Sections](#).

Generation of ERT S-functions that represent variant controls as preprocessor conditionals

Previously, when you generated an ERT S-function from a subsystem, `Simulink.Parameter` objects that you selected as tunable appeared in the S-function code as tunable global variables. You could change the values of these parameters in the S-Function block dialog box during simulation.

In R2016a, if a `Simulink.Parameter` object uses a custom storage class that treats the parameter as a macro in the generated code, you cannot select the parameter as a tunable parameter for the generated S-function. Instead, the S-function code generator applies the custom storage class to the parameter object. This generation of macros in the S-function code allows you to generate S-functions from subsystems that contain variant elements, such as Variant Subsystem blocks, that you configure to produce preprocessor conditionals in the generated code. However, you cannot change the value of the parameter during simulation of the S-function.

For more information about generating S-functions from subsystems, see [Macro Parameters](#).

Compatibility Considerations

If you apply macro custom storage classes to `Simulink.Parameter` objects, you can no longer select the parameter objects as tunable parameters when you generate an ERT S-function. To select these parameter objects as tunable parameters, apply a different storage class or custom storage class.

Code Generation

Default style C++ interface replaces the void-void style C++ interface

In C++ class interface support, the **Default step method** replaces the **Void-void step method**. The default style interface adds support for:

- Multitasking mode for model reference target
- Virtual bus for crossing model boundaries

When the **Code Generation** pane selection for **System target file** is `ert.tlc` (or is an ERT-derived target), the **Code Generation** pane selection for **Language** is C++, and the **Code Generation > Interface** pane selection for **Code interface packaging** is C++ class, you click the **Configure C++ Class Interface** button to configure the step method for your model.

For models configured to use the **Void-void step method**, the code generator treats this replaced configuration as the **Default step method**. No incompatibility occurs for the model configuration.

`RTW.ModelCPPDefaultClass` replaces `RTW.ModelCPPVoidClass`. Where code uses the replaced `RTW.ModelCPPVoidClass` class, update the code to use the `RTW.ModelCPPDefaultClass`, otherwise potential incompatibility can occur.

For information about the step methods, see Control Generation of C++ Class Interfaces. For information about using an ERT-derived target with C++ support, see Support C++ Class Interface Control.

Compiler warning limitation removed for portable word sizes in SIL simulations

Prior to R2016a, compilation warnings occurred for code generated by using portable word sizes if all of the following conditions existed:

- The combination of word sizes for the host and target computers caused `rtwtypes.h` to redefine the word sizes by using preprocessor macros. For example, when the target computer had a 16-bit `int` data type and the host computer had a 16-bit `short` data type, `int16_T` was redefined to be `short` on the host computer and `int` on the target computer. The data types were used for pointer arguments to function calls. The called functions resided on the host computer and were precompiled (not compiled using `rtwtypes.h`).
- The data types were used in pointer arguments to function calls.
- The called functions resided on the host computer and were precompiled (not compiled by using `rtwtypes.h`).

Under these conditions, the compiler typically issued a warning similar to the following warning:

```
warning: passing argument 2 of 'frexp' from incompatible pointer type
```

Executing the generated code on the host computer led to memory corruption. For example, the function `double frexp (double value, int *exp);` expected `int *` as the second argument. However, `int16_T *` is passed in the generated code. On the host computer, `int16_T` was redefined to `short`. During SIL simulation, `frexp` attempted to write four bytes to a 2-byte location.

The suggested workaround for this limitation was to develop a custom code replacement library for functions that wrote to address locations obtained through pointer arguments.

As of R2016a, this limitation does not apply. When you select portable word sizes, if possible, the code generator handles unsized arguments for standard library functions registered in libraries that MathWorks provides. For unhandled cases, the code generator reports an error.

If user-defined code replacements use arguments of word sizes that map to settings of hardware implementation model configuration parameters, and you select portable word sizes, the code generator issues a warning.

If you use portable word sizes, when possible, define the size of arguments.

For more information, see [Configure Hardware Implementation Settings and Enable portable word sizes](#).

AUTOSAR arxml round trip

R2016a enhances the AUTOSAR arxml round-trip workflow with support for:

- CompuMethods with LINEAR and TEXTTABLE COMPU-SCALES
- PredefinedVariants import and export
- Enhanced control of AUTOSAR package path specification

CompuMethods with LINEAR and TEXTTABLE COMPU-SCALES

In R2016a, you can import and export a CompuMethod that uses LINEAR and TEXTTABLE scaling. Importing application data types that reference CompuMethods of category SCALE_LINEAR_AND_TEXTTABLE creates Simulink.NumericType or Simulink.AliasType data objects in the Simulink workspace. In Simulink, you can modify the LINEAR scaling for the CompuMethods. The TEXTTABLE scaling is read-only.

For more information, see [CompuMethod Categories for Data Types and Modify Linear Scaling for SCALE_LINEAR_AND_TEXTTABLE CompuMethod](#).

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.1 or later.

PredefinedVariants import and export

For an AUTOSAR software component that contains variation points, to define the values that control variation points, you can use the following AUTOSAR elements:

- SwSystemconst — Defines a system constant that serves as an input to control a variation point.
- SwSystemconstantValueSet — Specifies a set of system constant values.
- PredefinedVariant — Describes a combination of system constant values, among potentially multiple valid combinations, to apply to an AUTOSAR software component.

Previously, when creating a model from arxml code, the arxml importer did not provide a way to specify a PredefinedVariant or SwSystemconstantValueSets as a basis for resolving variation points in the model.

In R2016a, you can resolve variation points in an AUTOSAR software component at model creation time. Specify a `PredefinedVariant` or `SwSystemconstantValueSets` with which the importer can initialize `SwSystemconst` data.

After model creation, you can run simulations and generate code based on the combination of variation point input values that you specified.

For more information, see [Model AUTOSAR Variants and Control AUTOSAR Variants with Predefined Value Combinations](#).

Enhanced control of AUTOSAR package path specification

In R2016a, if you modify an AUTOSAR package path, and if packageable elements of that category are affected, you can:

- Move the elements from the existing package to the new package.
- Set the new package path without moving the elements.

If you modify a package path in the `Configure AUTOSAR Interface` dialog box, and if packageable elements of that category are affected, a dialog box opens with control options. If you programmatically modify a package path, you can use the `MoveElements` property to specify handling of affected elements.

For more information, see [Control AUTOSAR Elements Affected by Package Path Modifications](#).

Note This capability is available to R2015b Embedded Coder customers by installing the R2015b Embedded Coder Support Package for AUTOSAR Standard, Version 15.2.0 or later.

Improved AUTOSAR library support for Mfx functions

As of R2016a, the AUTOSAR 4.0 code replacement library (CRL) replaces `abs`, `saturate`, `min`, and `max` function calls that involve operands with equal slope and bias with calls to corresponding `Mfx` functions.

Calls To	Replace
<code>Mfx_Abs</code>	<code>abs</code> with operands that have equal slope
<code>Mfx_Limit</code>	<code>saturate</code> with operands that have equal slope and bias
<code>Mfx_Max</code>	<code>max</code> with operands that have equal slope and bias
<code>Mfx_Min</code>	<code>min</code> with operands that have equal slope and bias

For more information about using the AUTOSAR 4.0 CRL, see [Code Generation with AUTOSAR Library](#).

AUTOSAR target no longer supports building wrapper subsystem as AUTOSAR SW-Component

In R2016a, the AUTOSAR target removes support for using right-click builds to build a wrapper subsystem that models an AUTOSAR SW-Component. In R2013b, a top model approach to modeling multirunnable AUTOSAR SW-Components replaced the wrapper subsystem approach. For more information, see [Multi-Runnable Software Components and Configure AUTOSAR Multiple Runnables](#).

Compatibility Considerations

In R2016a, if you try to configure and build an AUTOSAR SW-Component by using a wrapper subsystem, the software issues an error message. The message states that configuring a subsystem as an AUTOSAR SW-Component is not supported.

To convert subsystem multirunnables to top model multirunnables, use the subsystem-to-model conversion techniques described in [Convert a Subsystem to a Referenced Model](#). After the basic conversion, you must manually reestablish some AUTOSAR configuration information from the subsystem configuration in the new configuration.

Root model name in generated identifier for shared utility files

In R2016a, you can add the root model name to the generated identifier for shared utility files. When you merge code for multiple models, including the root model name in the generated identifier avoids name clashes. Name clashes arise due to identical shared utility file names.

To specify that the code generator add the root model name, in the Configuration Parameters dialog box, on the **Code Generation > Symbols** pane, add the \$R token to the **Shared Utilities** field.

Improved configuration parameter defaults for Embedded Coder targets

Improved configuration parameter defaults for Embedded Coder targets enable more optimizations and traceability options. These parameter defaults make it easier to develop your model for production code generation. In R2016a, when you switch your system target file to `ert.tlc`, the following configuration parameters are enabled by default:

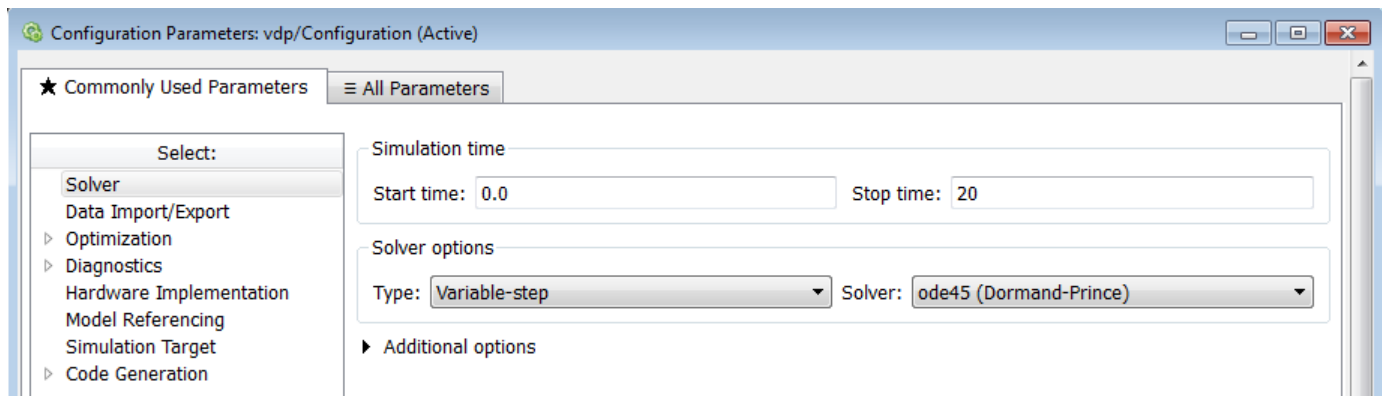
- **Convert if-elseif-else patterns to switch-case statements** (ConvertIfToSwitch)
- **Suppress generation of default cases for Stateflow switch statements if unreachable** (SuppressUnreachableDefaultCases)
- **Simulink data object descriptions** (SimulinkDataObjDesc)
- **Simulink block descriptions** (InsertBlockDesc)
- **Verbose comments for SimulinkGlobal storage class** (ForceParamTrailComments)
- **Open report automatically** (LaunchReport)
- **Create code generation report** (GenerateReport)
- **Stateflow object descriptions** (SFDataObjDesc)
- **Show eliminated blocks** (ShowEliminatedStatement)
- **Operator annotations** (OperatorAnnotations)
- **MATLAB function help text** (MATLABFcnDesc)
- **MATLAB source code as comments** (MATLABSourceComments)
- **Traceable Simulink blocks** (GenerateTraceReportSl)
- **Traceable Stateflow objects** (GenerateTraceReportSf)
- **Traceable MATLAB functions** (GenerateTraceReportEml)
- **Model-to-code** (GenerateTraceInfo)
- **Code-to-model** (IncludeHyperlinkInReport)

- **Eliminated / virtual blocks** (GenerateTraceReport)

Streamlined code generation panes for easier model configuration

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**

Code Generation > Verification Pane

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

- **Create block**

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:

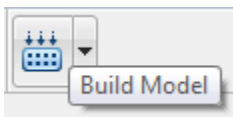
- **Reserved names**

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.



Improved web view for code generation report

In R2016a, significant updates improve the model Web view in the code generation report. Updates include:

- Graphics and navigation similar to the Simulink Editor.
- Block parameter and signal property value inspection using the **Object Inspector** pane.
- Model search for locating Simulink blocks and Stateflow objects.
- Tab support for displaying individual block diagrams.

For more information, see the Simulink Report Generator™ documentation.

Dependent parameters not added to custom code generation objective

Previously, when you added a parameter to a custom code generation objective using the `addParam` function, the software included the parameter dependencies in the list of parameter values that the Code Generation Advisor reviews. In R2016a, these dependent parameters are not added.

Removal of leading underscore character in macro type definitions

In R2015b, generated type definition macros began with an underscore character (`_`). In R2016a, the code generator does not include the underscore character at the beginning of these macros. This change in the generated code addresses MISRA C:2012 Rule 21.1.

For example, in R2015b, the code generator produced this code for an enumeration type definition:

```
#ifndef _DEFINED_TYPEDEF_FOR_EnumErrorType_
#define _DEFINED_TYPEDEF_FOR_EnumErrorType_

typedef enum {
    NoError = 0,                /* Default value */
    MeasuredVelocityError
} EnumErrorType;

#endif
```

The code contained an underscore character at the beginning of the name `_DEFINED_TYPEDEF_FOR_EnumErrorType_`.

In R2016a, the code generator produces this code for the same type definition:

```
#ifndef DEFINED_TYPEDEF_FOR_EnumErrorType_
#define DEFINED_TYPEDEF_FOR_EnumErrorType_

typedef enum {
    NoError = 0,                /* Default value */
    MeasuredVelocityError
} EnumErrorType;

#endif
```

The code does not contain an underscore character at the beginning of the name
DEFINED_TYPEDEF_FOR_EnumErrorType_.

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.

MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with Altera SoC and Xilinx Zynq hardware

In R2016a, you can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware using a MATLAB Coder workflow with an Embedded Coder license. By using PIL with hardware, you can more effectively generate customized code for your hardware by profiling speed and algorithm performance. You have the option of using the command-line workflow or the MATLAB Coder app to configure your target hardware for PIL executions.

This PIL execution is available with the following hardware support packages:

- Embedded Coder Support Package for Altera SoC Platform
- Embedded Coder Support Package for Xilinx Zynq-7000 Platform

To use this PIL execution, you must install one of these support packages. For more information, see:

- PIL Execution with ARM Cortex-A at the Command Line
- PIL Execution with ARM Cortex-A by Using the MATLAB Coder App

Updates to support package for Texas Instruments C2000 processors

The updated Embedded Coder Support Package for Texas Instruments C2000™ Processors, adds the code generation support for Texas Instruments Delfino F2837xD, F2837xS and Texas Instruments Piccolo F2807x processors. You must install the Embedded Coder Support Package for Texas Instruments C2000 Processors to use this support.

To install or update this support package, perform the steps described in [Install Support for TI's C2000 Processors](#).

For more information, see [Texas Instruments C2000 Processors](#).

Support package for Freescale FRDM-K64F board

You can use the Embedded Coder Support Package for Freescale™ FRDM-K64F Board to generate, build, and deploy code to the Freescale FRDM-K64F board. See [Install Support for Freescale FRDM-K64F Board](#). For more information, see [Embedded Coder Support Package for Freescale FRDMK64F Board](#).

Support for TI's C5000 DSPs will be removed

Support for TI's C5000™ DSPs will be removed. You can still use Embedded Coder for TI's C5000 processors, but need to manually integrate the generated code with hand written schedulers and drivers.

Support for TI's C6000 DSPs will be removed

Support for TI's C6000 DSPs will be removed in a future release. You will still be able to use Embedded Coder for TI's C6000 processors, but will need to manually integrate the generated code with hand written schedulers and drivers.

Change in base product for ARM Cortex-Based VEX Microcontroller support package

The base product for ARM Cortex-Based VEX Microcontroller support package is changed from Embedded Coder to Simulink Coder. However, you can use this support package with Embedded Coder to use some of the Embedded Coder features. For more information on Simulink Coder Support Package for ARM Cortex-based VEX® Microcontroller, see Simulink Coder Support Package for ARM Cortex-Based VEX Microcontroller.

Performance

Data Buffer Reuse: Use same variable for multiple signals in a path by using the same Reusable storage class specification

Previously, if the input and output signals at a block or subsystem boundary shared the same `Reusable` storage class specification, the code generator tried to reuse the signals in the generated code.

In R2016a, this optimization extends to blocks or subsystems that are in a path. The optimization decreases RAM/ROM consumption by reducing the number of global variables and data copies in the generated code.

For more information, see [Buffer Reuse Around a Block or Subsystem Boundary](#).

Reuse input, output, and state of Delay block

Previously, the code generator reused the input, output, and state of a Unit Delay block. In R2016a, the code generator tries to reuse the input, output, and state of a Delay block if in the Delay block parameters dialog box, the following conditions exist:

- The **Delay length** parameter has a value of 1.
- The **Initial condition > Source parameter** is set to Dialog.

For more information, see [Buffer Reuse for Model Block Boundary and Unit Delay](#).

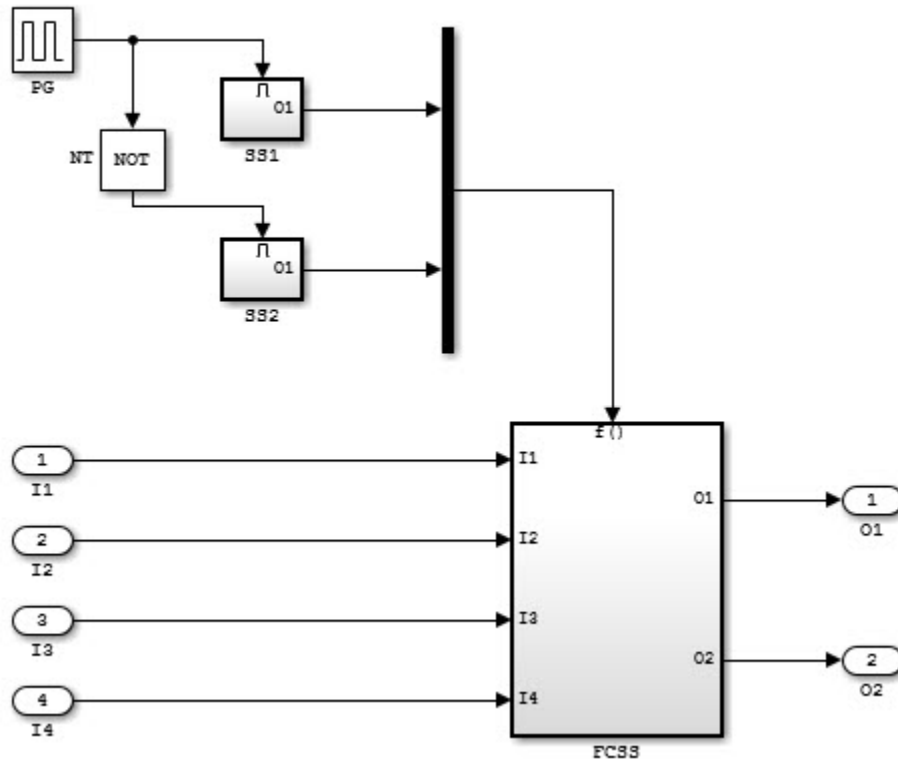
Initialization code occurs once after start code in `model_initialize` function

In R2016a, for conditionally executed subsystems, there are the following changes in the generated code for the `model_initialize` function:

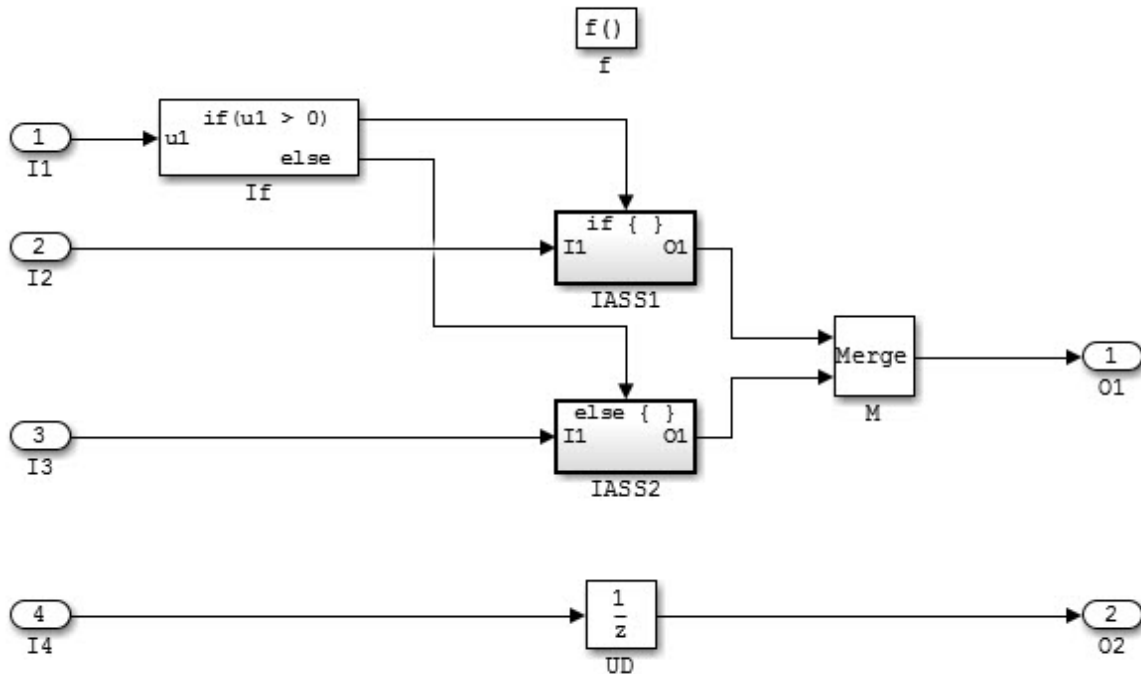
- In R2015b, the code generator called the `model_Subsystem_Init` function possibly before and after the `model_Subsystem_Start` function. In R2016a, the generated code contains one call to the `model_Subsystem_Init` function. This call occurs after the `model_Subsystem_Start` function. One call reduces code size and improves ROM consumption.
- In R2015b, the `model_Subsystem_Start` function and the `model_Subsystem_Init` function initialized the states of blocks. In R2016a, the `model_Subsystem_Init` function initializes the states of blocks. The `model_Subsystem_Start` function still performs other tasks involving a small selection of blocks.

For example, in the model `cond_sub`, a Mux block combines the signals from two enabled subsystems into one signal. This signal feeds into a function-call subsystem. One of the outputs from the function-call subsystem is the combination of signals from two subsystems. The other output is the signal from the Unit Delay block.

cond_sub



Contents of FcnCall SS



In R2015b, the code generator produced this code in the `cond_sub.c` file:

```

void cond_sub_IASS1_Init(void)
{
    cond_sub_DW.UD_DSTATE_c = 0.0;
}
void cond_sub_IASS1_Start(void)
{
    cond_sub_IASS1_Init();
}
...

void cond_sub_FCSS_Init(void)
{
    cond_sub_DW.UD_DSTATE = 2.0;
}
...

void cond_sub_FCSS_Start(void)
{
    cond_sub_DW.If_ActiveSubsystem = -1;
    cond_sub_IASS1_Start();
    cond_sub_B.M = 3.0;
    cond_sub_Y.O1 = 4.0;
}

void cond_sub_initialize(void)
{

```

```

(void) memset((void *)&cond_sub_U, 0,
              sizeof(ExtU_cond_sub_T));
(void) memset((void *)&cond_sub_Y, 0,
              sizeof(ExtY_cond_sub_T));
cond_sub_FCSS_Init();
cond_sub_FCSS_Start();
cond_sub_FCSS_Init();
}

```

The `cond_sub_initialize` function calls the `cond_sub_FCSS_Init` function before and after the `cond_sub_FCSS_Start` function. The `cond_sub_FCSS_Init` function sets the initial condition of the Unit Delay block. The `cond_sub_FCSS_Start` function sets the initial conditions of the Merge block and the Outputport block, 01.

In R2016a, the code generator produces this code in the `cond_sub.c` file:

```

void cond_sub_FCSS_Init(void)
{
    cond_sub_DW.UD_DSTATE = 2.0;
    cond_sub_B.M = 3.0;
    cond_sub_Y.01 = 4.0;
}
...
void cond_sub_FCSS_Start(void)
{
    cond_sub_DW.If_ActiveSubsystem = -1;
}
...
void cond_sub_initialize(void)
{
    (void) memset((void *)&cond_sub_U, 0,
                  sizeof(ExtU_cond_sub_T));
    (void) memset((void *)&cond_sub_Y, 0,
                  sizeof(ExtY_cond_sub_T));
    cond_sub_FCSS_Start();
    cond_sub_FCSS_Init();
}

```

In R2016a, the `cond_sub_FCSS_Start` function occurs once before the `cond_sub_FCSS_Init` function. The `cond_sub_FCSS_Init` function sets the initial condition of the Merge block, the Outputport block, 01, and the Unit Delay block. The `cond_sub_FCSS_Start` function does not set the initial conditions of blocks.

Reset function improves initialization code optimization

In R2016a, for models containing a conditionally executed subsystem and a reusable subsystem or model reference, the initialization code contains a new function called `model_Reset` or `subsystem_Reset`. The `model_Reset` or `subsystem_Reset` function sets the states of blocks inside a subsystem or model reference back to their initial conditions. The `subsystem_Init` function sets the states of blocks inside a model reference or subsystem to their initial conditions for the first time.

In the Configuration Parameters dialog box, when you select **Optimization > Remove internal data zero initialization**, the code generator does not generate code that initializes internal work structures to zero. This optimization reduces code size and increases execution speed.

For example, in the `cond_sub` model (shown in this release note: “Initialization code occurs once after start code in `model_initialize` function” on page 14-25), the function-call subsystem contains two Unit Delay blocks. One Unit Delay block connects to the output, `o2` and has an initial condition of 2. The other Unit Delay block is inside the subsystem `IASS1` and has an initial condition of 0.

In R2015b, the code generator produced the following code in the `cond_sub.c` file:

```
void cond_sub_IASS1_Init(void)
{
    /* InitializeConditions for UnitDelay: '<S4>/UD' */
    cond_sub_DW.UD_DSTATE_c = 0.0;
}
...
void cond_sub_FCSS_Init(void)
{
    /* InitializeConditions for UnitDelay: '<S1>/UD' */
    cond_sub_DW.UD_DSTATE = 2.0;
}
```

In R2015b, the code generator creates the `cond_sub_FCSS_Init` and the `cond_sub_IASS1_init` functions to initialize and reset the state of each Unit Delay block.

In R2016a, the code generator produces the following code inside of the `cond_sub.c` file:

```
/* System reset for action system: '<S1>/IASS1' */
void cond_sub_IASS1_Reset(void)
{
    /* InitializeConditions for UnitDelay: '<S4>/UD' */
    cond_sub_DW.UD_DSTATE_c = 0.0;
}
...
/* System initialize for function-call system: '<Root>/FCSS' */
void cond_sub_FCSS_Init(void)
{
    /* InitializeConditions for UnitDelay: '<S1>/UD' */
    cond_sub_DW.UD_DSTATE = 2.0;

    /* SystemInitialize for Merge: '<S1>/M' */
    cond_sub_B.M = 3.0;

    /* SystemInitialize for Output: '<Root>/01' incorporates:
     * SystemInitialize for Output: '<S1>/01'
     */
    cond_sub_Y.01 = 4.0;
}

/* System reset for function-call system: '<Root>/FCSS' */
void cond_sub_FCSS_Reset(void)
{
    /* InitializeConditions for UnitDelay: '<S1>/UD' */
    cond_sub_DW.UD_DSTATE = 2.0;
}
```

In R2016a, the `cond_sub_FCSS_init` function initializes the state of one Unit Delay block. The code generator does not generate a `cond_sub_IASS1_Init` function to initialize the state of the other Unit Delay block to zero because the **Remove internal data zero initialization** parameter is selected.

The `void cond_sub_IASS1_Reset` and the `void cond_sub_FCSS_Reset` functions reset the states of the Unit Delay blocks.

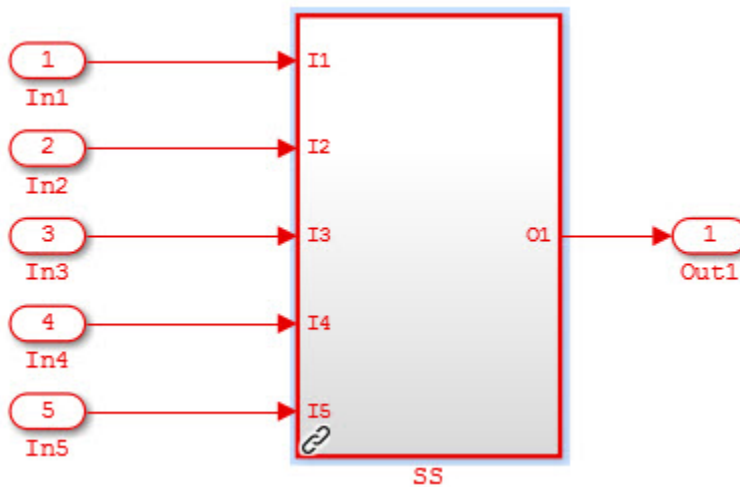
If you know that a parent model does not have to reset the states of blocks inside a model reference, you can remove the `model_Reset` function from the generated code. In the Configuration Parameters dialog box, select **Optimization > Optimize initialization code for model reference** to remove the `model_Reset` function.

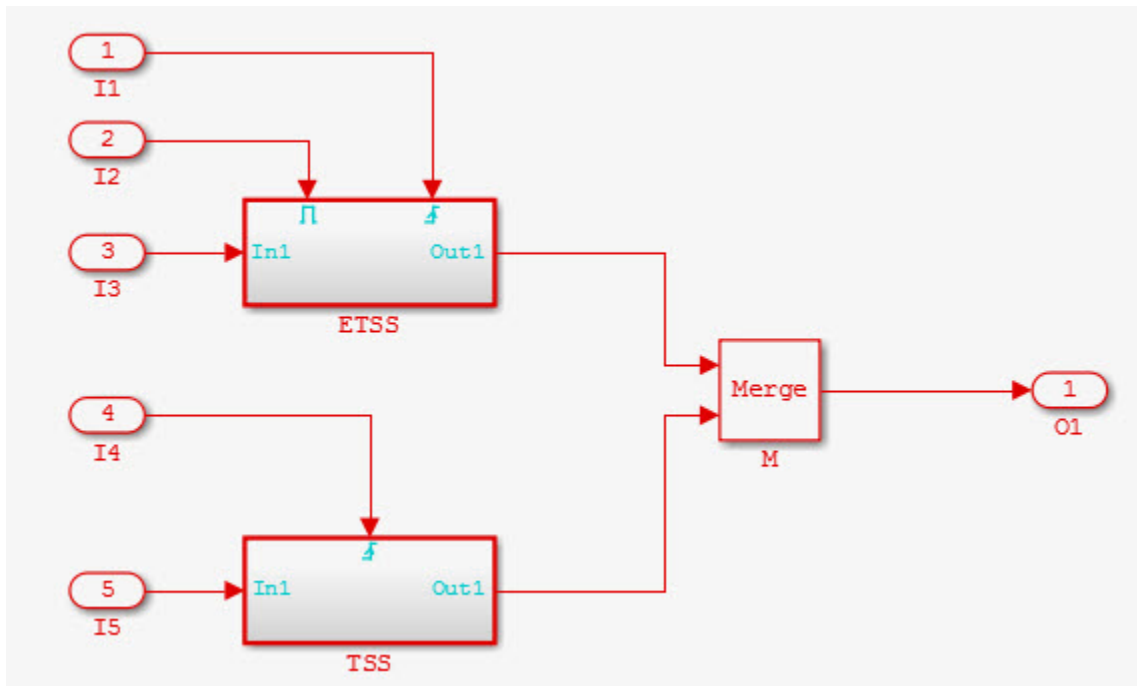
Removal of unnecessary `rtmIsFirstInitCond` flag

In R2015b, for modeling patterns involving conditionally executed subsystems, the code generator created an `rtmIsFirstInitCond` flag in the `model_initialize` function and in the `model_step` function.

In R2016a, the code generator does not generate the `rtmIsFirstInitCond` flag, except for S-Function blocks. This enhancement reduces code size and ROM consumption and enables code reuse and a Simulink Code Inspector™ verification.

For example, the model `removeflag` contains a subsystem. This subsystem contains an enabled and triggered subsystem and a triggered subsystem that feed into a Merge block.





In R2015b, in the `removeflag.c` file, the code generator produced this code in the `removeflag_initialize` function:

```
/* InitializeConditions for Atomic SubSystem: '<Root>/SS' */
removeflag_SS_Init(removeflag_M, &removeflag_B.SS);

/* End of InitializeConditions for SubSystem: '<Root>/SS' */
```

The code for the `removeflag_SS_Init` function was as follows:

```
/* Initial conditions for atomic system: '<Root>/SS' */
void removeflag_SS_Init(RT_MODEL_removeflag_T * const removeflag_M,
    B_SS_removeflag_T *localB)
{
    /* InitializeConditions for Merge: '<S1>/M' */
    if (rtmIsFirstInitCond(removeflag_M)) {
        localB->M = 3.0;
    }

    /* End of InitializeConditions for Merge: '<S1>/M' */
}
```

In R2015b, for the `removeflag_SS_Init` function, the generated code contained the `rtmIsFirstInitCond` flag.

In R2016a, in the `_sharedutils` folder, the code generator produces this reusable code:

```
/* System initialize for atomic system: 'SS' ('removeflagLib:1') */
void SS_bbDo8UEo_Init(B_SS_bbDo8UEo_T *localB)
{
    /* SystemInitialize for Merge: 'M' ('removeflagLib:11') */
    localB->M = 3.0;
}
```

The `removeflag.c` file contains a call to the `SS_bbDo8UEo_T_Init` function inside the `removeflag_initialize` function:

```
/* SystemInitialize for Atomic SubSystem: '<Root>/SS' */
  SS_bbDo8UEo_Init(&removeflag_B.SS);

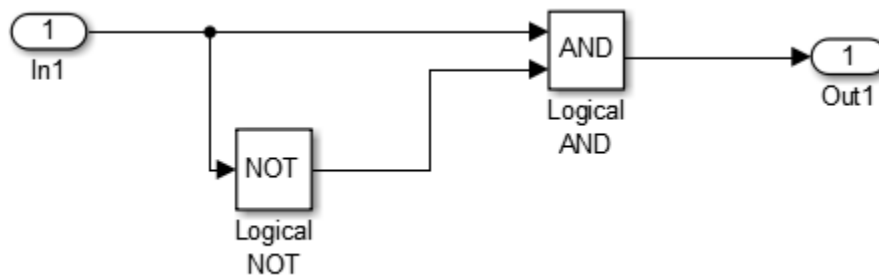
  /* End of SystemInitialize for SubSystem: '<Root>/SS' */
```

The generated code does not contain the `rtmIsFirstInitCond` flag. Instead, the code generator generates reusable code for the `SS_bbDo8UEo_T_Init` function. The `rtmIsFirstInitCond` flag is not needed because the `model_Subsystem_Init` function initializes the states of blocks while the new reset function sets the states of all blocks back to their initial conditions.

Optimized code for models containing logical operator blocks

In R2015b, for a model where an input signal fed into a Logical NOT block and either a Logical AND block or a Logical OR block, the generated code contained an expression for the Logical NOT and Logical AND or Logical OR blocks. In R2016a, the generated code sets the output equal to either true or false. This optimization simplifies the code and improves code efficiency.

For example, in the model `andornotself`, the input signal feeds into the Logical NOT block and the Logical AND block.



In R2015b, the generated code contained this code:

```
/* Model step function */
void andornotself_step(void)
{
  /* Output: '<Root>/Out1' incorporates:
   * Inport: '<Root>/In1'
   * Logic: '<Root>/Logical AND'
   * Logic: '<Root>/Logical NOT'
   */
  andornotself_Y.Out1 = (andornotself_U.In1 && (!andornotself_U.In1));
}
```

In R2016a, the generated code contains this code:

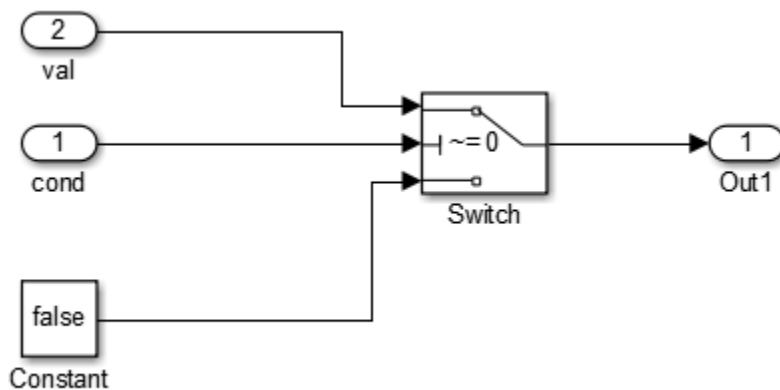
```
/* Model step function */
void andornotself_step(void)
{
  /* Output: '<Root>/Out1' */
  andornotself_Y.Out1 = false;
}
```

The optimized code sets `andornotself_Y.Out1` equal to `false` because the condition `andornotself_Y.Out1 = (andornotself_U.In1 && (!andornotself_U.In1))` is false.

Improved code for conditional expressions involving Boolean expressions

In R2015b, for a model in which the generated code contained a conditional expression involving Boolean expressions, the generated code contained an `if-else` statement. In R2016a, the generated code uses `&&` and `||` operators to enable short-circuit evaluation. This optimization simplifies the code and improves code efficiency.

For example, the model `booleanConditionalExpr` contains two Inport blocks, a Switch block, a Constant block, and an Outport block. The Constant block has a value of `false`.



In R2015b, the code generator generated this code:

```
/* Model step function */
void booleanConditionalExpr_step(void)
{
    /* Switch: '<Root>/Switch' incorporates:
     * Inport: '<Root>/cond'
     */
    if (booleanConditionalExpr_U.cond) {
        /* Outport: '<Root>/Out1' incorporates:
         * Inport: '<Root>/val'
         */
        booleanConditionalExpr_Y.Out1 = booleanConditionalExpr_U.val;
    } else {
        /* Outport: '<Root>/Out1' incorporates:
         * Constant: '<Root>/Constant'
         */
        booleanConditionalExpr_Y.Out1 = false;
    }

    /* End of Switch: '<Root>/Switch' */
}
```

The generated code contained an `if-else` statement.

In R2016a, the code generator generates this code:

```

/* Model step function */
void booleanConditionalExpr_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/cond'
     * Inport: '<Root>/val'
     */
    booleanConditionalExpr_Y.Out1 = (booleanConditionalExpr_U.cond &&
        booleanConditionalExpr_U.val);
}

```

The generated code contains an && expression. If `booleanConditionalExpr_U.cond` is false, the && expression short-circuits and `booleanConditionalExpr_Y.Out1` is equal to false. Otherwise, `booleanConditionalExpr_Y.Out1` is equal to `booleanConditionalExpr_U.val`.

memset Optimization for more scenarios

- “memset optimization for assigning a constant value to fields of a structure array” on page 14-34
- “memset optimization for array element assignments” on page 14-36
- “memset optimization for consecutive assignments that define a continuous write” on page 14-37

In R2015b, the code generator tried to replace a `for` loop that assigned a literal constant to consecutive array elements with a `memset` function call. A `memset` function call can be more efficient than `for`-loop controlled array element assignments.

In R2016a, the code generator attempts to invoke the `memset` optimization when assigning a constant value to all fields of a structure array. The code generator attempts to invoke the `memset` optimization for a loop with one or more array element assignments and for consecutive statements that define a continuous write.

Note The minimum array size for which `memset` function calls can replace `for` loops depends on the setting of the **Memcpy threshold (bytes)** parameter. By default, this parameter specifies 64 bytes as the minimum array size for which `memset` function calls can replace `for` loops in the generated code.

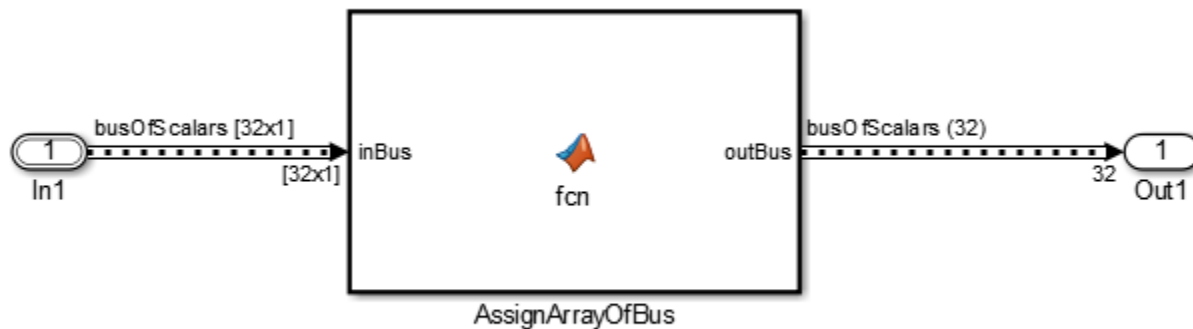
memset optimization for assigning a constant value to fields of a structure array

The following Simulink modeling pattern produces C code with a constant value assignment to fields of a structure array:

- The input to a MATLAB Function block is an array of buses. The bus elements are scalars.
- The MATLAB Function block contains a structure that writes the same value to each bus element.

In R2015b, for this modeling pattern, the generated code contained `for` loop controlled array element assignments. In R2016a, the code generator can replace these `for` loop controlled array element assignments with `memset` function calls. This optimization improves execution speed.

For example, in the following model, the input signal is an array of busses. The bus elements are the three scalars, `f1`, `f2`, and `f3`.



The MATLAB Function block contains this code:

```
function outBus = fcn(inBus)
%#codegen

for k = 1:24
    inBus(k).f1 = int32(0);
    inBus(k).f2 = int32(0);
    inBus(k).f3 = int32(0);
end

outBus=inBus;
end
```

In R2015b, the code generator produced this code:

```
/* MATLAB Function 'AssignArrayOfBus': '<S1>:1' */
/* '<S1>:1:4' for k = 1:24 */
for (k = 0; k < 24; k++) {
    /* '<S1>:1:5' inBus(k).f1 = int32(0); */
    inBus[k].f1 = 0;

    /* '<S1>:1:6' inBus(k).f2 = int32(0); */
    inBus[k].f2 = 0;

    /* '<S1>:1:7' inBus(k).f3 = int32(0); */
    inBus[k].f3 = 0;
}

/* '<S1>:1:10' outbus = inBus; */
memcpy(&localB->outBus[0], &inBus[0], sizeof(busOfScalars) << 5U);
```

The generated code contained a for loop for assigning a value of `int32(0)` to the structure fields, `f1`, `f2`, and `f3`.

In R2016a, the code generator produces this code:

```
/* MATLAB Function 'AssignArrayOfBus': '<S1>:1' */
/* '<S1>:1:4' for k = 1:24 */
/* '<S1>:1:5' inBus(k).f1 = int32(0); */
```

```

/* '<S1>:1:6' inBus(k).f2 = int32(0); */
/* '<S1>:1:7' inBus(k).f3 = int32(0); */
memset(&inBus[0], 0, 24U * sizeof(busOfScalars));

/* '<S1>:1:9' outBus=inBus; */
memcpy(&localB->outBus[0], &inBus[0], sizeof(busOfScalars) << 5U); }

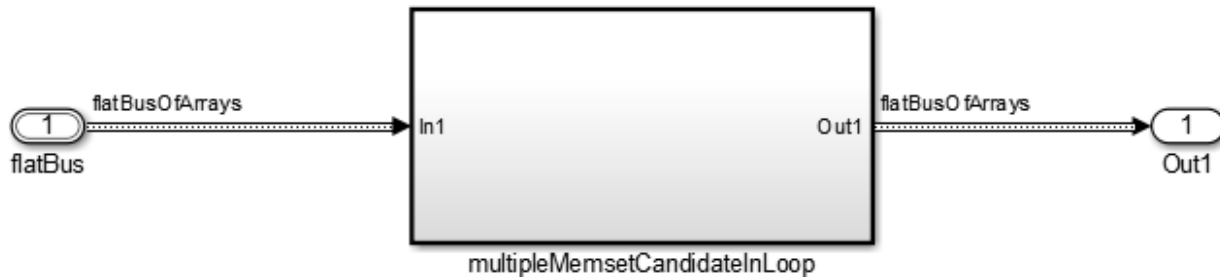
```

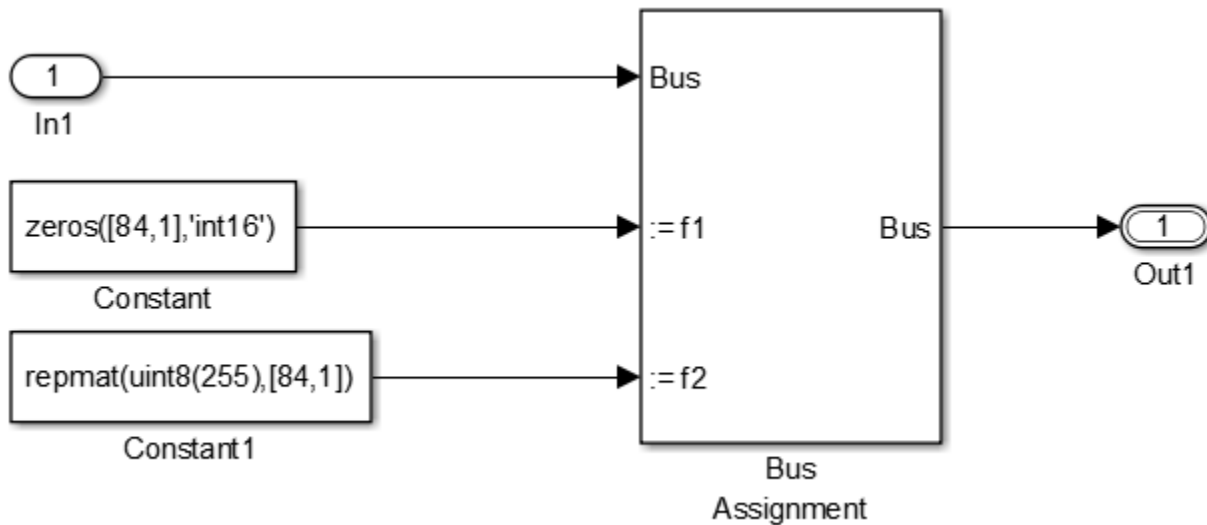
The generated code contains a `memset` function call for assigning a value of `int32(0)` to the structure fields `f1`, `f2`, and `f3`.

memset optimization for array element assignments

For a Simulink model containing a Bus Assignment block that accepts a bus signal consisting of arrays, the code generator produces C code with one or more array element assignments. If the Bus Assignment block assigns values to a single array of the bus signal, the generated code contains one array element assignment. If the Bus Assignment block assigns values to arrays in the bus signal, there are multiple array element assignments. In R2015b, the generated code contained `for` loop controlled array element assignments. In R2016a, the code generator can replace these `for` loop controlled array element assignments with `memset` function calls. This optimization improves execution speed.

For example, in following model, the input signal is a `Simulink.Bus` object consisting of two arrays, `f1` and `f2`. The Bus Assignment block assigns a value of `0` to every element in `f1` and a value of `255 (MAX_uint8_T)` to every element in `f2`.





In R2015b, the code generator produced this code:

```
/* Model step function */
void memsetexample_step(void)
{
    int32_T i;

    /* Outputport: '<Root>/Out1' */
    for (i = 0; i < 84; i++) {
        memsetexample_Y.Out1.f1[i] = 0;
        memsetexample_Y.Out1.f2[i] = MAX_uint8_T;
    }

    /* End of Outputport: '<Root>/Out1' */
}
```

The generated code contained a `for` loop for assigning values to the arrays `f1` and `f2`.

In R2016a, the code generator produces this code:

```
/* Model step function */
void memsetexample_step(void)
{
    /* Outputport: '<Root>/Out1' */
    memset(&memsetexample_Y.Out1.f1[0], 0, 84U * sizeof(int16_T));
    memset(&memsetexample_Y.Out1.f2[0], 255, 84U * sizeof(uint8_T));
}
```

The generated code contains `memset` function calls for assigning values to `f1` and `f2`.

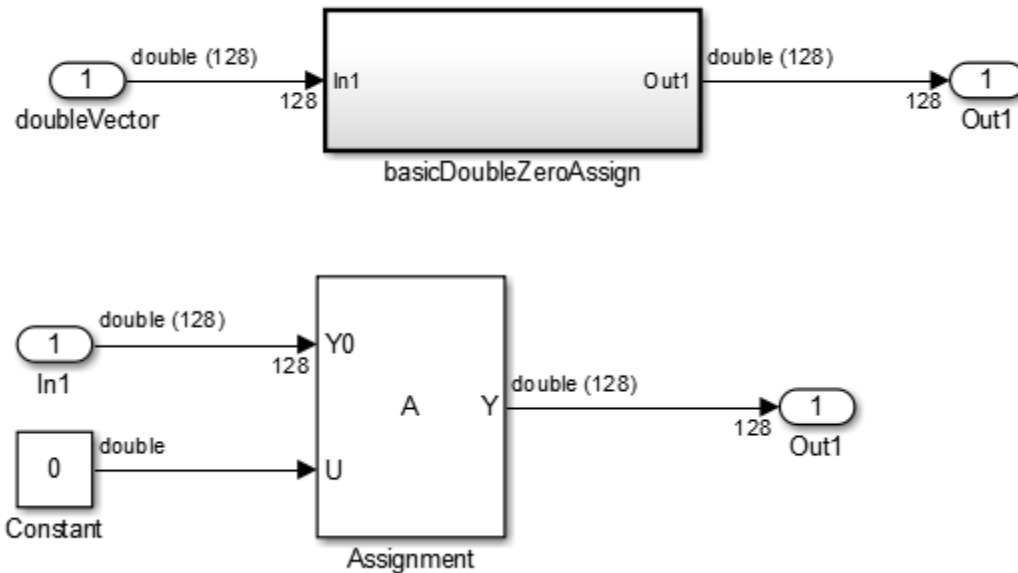
memset optimization for consecutive assignments that define a continuous write

For a Simulink model containing a 1-D, 2-D, or multidimensional signal that feeds into an Assignment block, the code generator produces C code with consecutive array element assignments. In R2015b, if the following modeling conditions were met, the generated code contained multiple assignment statements:

- The Assignment block assigned a value of 0 to multiply elements of an output signal.
- In the generated code, the array size was below the value of the loop unrolling threshold parameter.

In R2016a, regardless of the value you set for the **Loop unrolling threshold** parameter, the code generator can replace these assignment statements with a `memset` function call. This optimization improves execution speed.

For example, in the Inport block parameters dialog box, the **Port Dimensions** parameter has a value of 128. The Assignment block assigns a value of 0 to the first 10 elements of this signal.



In R2015b, with a the code generator produced this code:

```
void memsetEx_basicDoubleZeroAssign(const real_T rtu_In1[128],
  B_basicDoubleZeroAssign_memse_T *localB)
{
  int32_T i;

  /* Assignment: '<S1>/Assignment' incorporates:
   * Constant: '<S1>/Constant'
   */
  memcpy(&localB->Assignment[0], &rtu_In1[0], sizeof(real_T) << 7U);
  localB->Assignment[0] = 0.0;
  localB->Assignment[1] = 0.0;
  localB->Assignment[2] = 0.0;
  localB->Assignment[3] = 0.0;
  localB->Assignment[4] = 0.0;
  localB->Assignment[5] = 0.0;
  localB->Assignment[6] = 0.0;
  localB->Assignment[7] = 0.0;
  localB->Assignment[8] = 0.0;
  localB->Assignment[9] = 0.0;
  /* End of Assignment: '<S1>/Assignment' */
}
```

The generated code contained individual write statements for assigning a value of 0 to the first 10 elements of the Assignment array.

In R2016a, the code generator produces this code:

```
void memsetEx_basicDoubleZeroAssign(const real_T rtu_In1[128],
    B_basicDoubleZeroAssign_memse_T *localB)
{
    /* Assignment: '<S1>/Assignment' incorporates:
     * Constant: '<S1>/Constant'
     */
    memcpy(&localB->Assignment[0], &rtu_In1[0], sizeof(real_T) << 7U);
    memset(&localB->Assignment[0], 0, 10U * sizeof(real_T));
}
```

The generated code contains a memset function call for assigning a value of 0 to the first 10 elements of the Assignment array.

Changes to meaning of createCRLEntry wildcard syntax for fixed-point data

The meaning of wildcard symbols tilde (~) and asterisk (*) in conceptual argument syntax specifications that you specify with the createCRLEntry function have changed.

Modified Syntax	Meaning Prior to R2016a	Meaning Starting with R2016a
Tilde symbol	Slopes must be the same across data types	Based on the position of the symbol, slopes or bias must be the same across data types
fixdt(1,16,*) y1 = sin(fixdt(1,16,*) u1) conceptual specification	Specify fixed-point data types and wildcard	Specify fixed-point data types and set CheckSlope to false and CheckBias to true
fixdt(1,16,~) y1 = sin(fixdt(1,16,~) u1) conceptual specification	Not applicable	Specify fixed-point data types and set SlopesMustBeTheSame to true, CheckSlope to false, and CheckBias to true
fixdt(1,16,~,~) y1 = sin(fixdt(1,16,~,~) u1) conceptual specification	Not applicable	Specify fixed-point data types and set SlopesMustBeTheSame to true, BiasMustBeTheSame to true, CheckSlope to false, and CheckBias to false
fixdt(1,16,*) y1 = fixdt(1,16,*) u1 + fixdt(1,16,*) u2 conceptual specification	Specify fixed-point data types and wildcard	Specify fixed-point data types and set CheckSlope to false and CheckBias to true

For more information, see the description of the createCRLEntry function.

Code replacements involving root-level I/O variables and data alignment

The code generator does not replace functions that use root-level I/O variables or AUTOSAR inter-runnable access functions when it generates function code with C function prototype control, C++ class I/O arguments step method, or the AUTOSAR system target file.

If the following conditions exist, the code generator includes data alignment directives for root-level I/O variables in the example main program file (`ert_main.c` or `ert_main.cpp`) that it produces:

- Compiler supports global variable alignment
- Generate an example main program (select **Configuration Parameters > All Parameters > Generate an example main program**)
- Generate a reusable function interface for the model (set **Configuration Parameters > Code Generation > Interface > Code interface packaging** to Reusable function)
- Function uses root-level I/O variables that are passed in as individual arguments (set **Configuration Parameters > Code Generation > Interface > Pass root-level I/O as to Individual arguments**)
- Replaced function uses a root-level I/O variable
- Replaced function imposes alignment requirements

If you discard the generated example main program, align used root-level I/O variables correctly.

If you choose not to generate an example main program in this case, the code generator does not replace the function.

For more information, see Code Replacement Customization Limitations.

Verification

SIL/PIL Data Access: Use vector Get/Set custom storage class and C++ parameter access methods

R2016a adds SIL and PIL support for data access capabilities:

- GetSet custom storage class support for vector signals and parameters. Previously, GetSet SIL and PIL support was available for scalar signals, parameters, and global data stores. For more information, see Access Data Through Functions with Custom Storage Class GetSet.
- Simulation support for the Method and Inlined method options for the **Configuration Parameters > Code Generation > Interface > Parameter access** parameter. For more information, see Control Generation of C++ Class Interfaces.

SIL/PIL support for variant condition propagation

Model block SIL/PIL simulations support variant condition propagation with Variant Source and Variant Sink blocks.

Top-model SIL/PIL and SIL/PIL block simulations do not support the propagation of variant conditions across component boundaries.

SIL simulation returns standard output and standard error streams

During a SIL simulation, the SIL application redirects the `stdout` and `stderr` streams. When the application terminates, the Diagnostic Viewer now displays the information from the redirected streams.

The SIL application also provides a basic signal handler, which captures the POSIX signals SIGFPE, SIGILL, SIGABRT, and SIGSEV. The SIL application includes the file `signal.h` for the signal handler.

The information from the redirected streams can help you to debug SIL applications that fail before the simulation is complete. For example, you can view:

- Output from `printf` statements in your code.
- Messages sent to `stderr`.
- Some low-level system messages.

For more information, see Debug SIL Simulation.

Linux SIL/PIL support for LDRA Testbed

For SIL and PIL simulations on Linux systems, you can collect code coverage metrics by using LDRA Testbed® from LDRA Technology. For more information, see Code Coverage Tool Support.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.8.1

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.9

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

MATLAB Coder Storage Classes: Easily import and export data by using storage classes

In R2015b, when you generate C/C++ code from MATLAB code, you can use a storage class to control the declaration and definition of a global variable in the generated code. Use of storage classes requires an Embedded Coder license.

In the context of code generation, a storage class is a specification that determines the declaration and definition of a variable in the generated code. For code generation, the term storage class is not the same as the C language term storage class specifier.

Storage classes help you to integrate generated code with external code. You can make a generated variable visible to external code. You can also make variables declared in the external code visible to the generated code. For code generation from MATLAB code, you can use storage classes with global variables only. The storage class determines:

- The file placement of a global variable declaration and definition.
- Whether the global variable is imported from external code or exported for use by external code.

To assign a storage class to a global variable, in your MATLAB code, use the `coder.storageClass` function. Only when you use an Embedded Coder project or configuration object for generation of C/C++ libraries or executables does the code generation software recognize `coder.storageClass` calls.

The syntax for `coder.storageClass` is:

```
coder.storageClass(var_name, storage_class)
```

`var_name` is the name of a global variable. Specify `var_name` as a constant string.

`storage_class` can be one of the following values:

- 'ExportedGlobal'
- 'ImportedExtern'
- 'ImportedExternPointer'

For descriptions of these storage classes, see [Storage Classes for Code Generation from MATLAB Code](#).

For example, `coder.StorageClass('g','ExportedGlobal')` assigns the exported global storage class to the global variable `g`.

For a detailed example, see [Control Declarations and Definitions of Global Variables in Code Generated from MATLAB Code](#).

If you do not assign a storage class to a global variable, the code generated for the variable is the same as the code generated in previous releases.

MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with BeagleBone Black hardware

In R2015b, you can use processor-in-the-loop (PIL) executions to verify generated code that you deploy to target hardware using a MATLAB Coder workflow with an Embedded Coder license. By using PIL with hardware, you can more effectively generate customized code for your hardware by profiling speed and algorithm performance. You have the option of using the command-line workflow or the MATLAB Coder app to configure your target hardware for PIL executions.

This PIL execution is available with the following hardware support packages:

- Embedded Coder Support Package for BeagleBone® Black Hardware
- Embedded Coder Support Package for ARM Cortex-A Processors

To use this PIL execution, you must install one of these support packages. For more information, see:

- PIL Execution with ARM Cortex-A at the Command Line
- PIL Execution with ARM Cortex-A by Using the MATLAB Coder App

Code generation assumptions verified during PIL execution

The settings on the **More Settings > Hardware** tab specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

At the start of a processor-in-the-loop (PIL) execution, the software verifies the **Hardware** tab settings with reference to the target hardware. The software checks:


- The correctness of settings. For example, the integer bit length in the **Sizes > int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

If required, the software generates warnings and errors.

Control of signed right shifts in generated code

You can now control the use of signed right shifts in your generated code. Some coding standards do not allow bitwise operations on signed integers. Disabling the use of signed shifts in generated code increases the likelihood of compliance with MISRA. When you specify that signed right shifts should not be used in your generated code, the software replaces signed shifts with a call to a function that performs the operation without the use of signed shifts.

To specify that MATLAB Coder not use signed right shifts:

- Using the MATLAB Coder app:
 - 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
 - 2 Set **Build type** to one of the following:

- Source Code
 - Static Library (.lib)
 - Dynamic Library (.dll)
 - Executable (.exe)
- 3** Click **More Settings**.
 - 4** On the **Code Appearance** tab, clear the **Allow right shifts on signed integers** check box.
- Using the command-line interface:
 - 1** Create a code configuration object for 'lib', 'dll', or 'exe'.

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```
 - 2** Set the EnableSignedRightShifts property to false.

```
cfg.EnableSignedRightShifts = false;
```

Detection of multiword operations

When an operation has an input or output larger than the largest word size of your processor, the generated code contains multiword operations. Multiword operations can be inefficient on hardware. The expensive fixed-point operations check now highlights expressions in your MATLAB code that could result in multiword operations in generated code. For more information on this check, see [Find and Address Multiword Operations](#).

Model Architecture and Design

MISRA-C 2012: Comply with mandatory and required rules

Model Advisor checks support compliance with MISRA C:2012. Previously, Model Advisor checks supported compliance with MISRA C:2004. To check that you developed your model or subsystem to increase the likelihood of generating MISRA C:2012 compliant code:

- 1 Open the Model Advisor.
- 2 Navigate to **By Task > Modeling Guidelines for MISRA C:2012**.
- 3 Run the checks in the folder.

The following table summarizes the check changes. For information about MISRA C versions and updates, see MISRA C Guidelines.

Check	Update	Addresses
Check configuration parameters for MISRA C:2012	Renamed from Check configuration parameters for MISRA-C:2004 compliance	MISRA C:2012
Check for blocks not recommended for MISRA C:2012	Renamed from Check for blocks not recommended for MISRA-C:2004 compliance	MISRA C:2012
Check for bitwise operations on signed integers	None	MISRA C:2012, Dir 10.1
Check for recursive function calls	New	MISRA C:2012, Dir 17.2
Check for equality and inequality operations on floating-point values	New	MISRA C:2012, Dir 1.1
Check for switch case expressions without a default case	New	MISRA C:2012, Rule 16.4

AUTOSAR 4.1.3 and 4.2: Import and export ARXML and generate code for latest AUTOSAR standard

R2015b extends AUTOSAR schema support to schema 4.2 (revision 4.2.1) and schema 4.1 (revision 4.1.3). For a detailed list of AUTOSAR schemas supported for import and export of arxml files and generation of AUTOSAR-compatible C code, see Select an AUTOSAR Schema.

R2015b provides many other enhancements to Simulink modeling of AUTOSAR elements and AUTOSAR code generation. For more information, see:

- Under Model Architecture and Design:
 - “AUTOSAR sender-receiver modeling” on page 16-6
 - “AUTOSAR client-server modeling” on page 16-8
 - “AUTOSAR nonvolatile data communication modeling” on page 16-9
 - “AUTOSAR component behavior modeling” on page 16-11

- “AUTOSAR COM_AXIS lookup table modeling” on page 16-12
- Under Code Generation:
 - “AUTOSAR arxml round-trip” on page 16-17
 - “Toolchain controls for AUTOSAR code generation” on page 16-19
 - “AUTOSAR RTE file generation enhanced for SIL and PIL” on page 16-19
 - “Lookup table blocks with new even spacing specification generate AUTOSAR compatible IFX library routines” on page 16-20

AUTOSAR sender-receiver modeling

R2015b enhances AUTOSAR sender-receiver modeling with support for:

- `IsUpdated` API for receiver ports
- Data element invalidation policies on sender ports
- End-to-end protection for sender and receiver ports
- `DataReceiveErrorEvent` for receiver ports
- `Rte_IWriteRef` for sender ports

IsUpdated API for receiver ports

AUTOSAR defines quality of service attributes, such as `ErrorStatus` and `IsUpdated`, for sender-receiver interfaces. R2015b adds support for the AUTOSAR `IsUpdated` attribute and API. The `IsUpdated` attribute allows an AUTOSAR receiver to detect when a receiver port data element has received data since the last read occurred. When data is idle, the receiver can save computational resources. You can:

- Import an AUTOSAR receiver port for which `IsUpdated` service is configured.
- Use Simulink to configure an AUTOSAR receiver port for `IsUpdated` service.
- Generate C and arxml code for an AUTOSAR receiver port for which `IsUpdated` service is configured.

For more information, see [Configure AUTOSAR Receiver Port for IsUpdated Service](#).

Data element invalidation policies on sender ports

AUTOSAR defines an invalidation mechanism for data elements on AUTOSAR sender ports. To protect downstream data consumers from receiving invalid data, you can define an invalidation policy for a sender port data element. R2015b adds support for data element invalidation policies on sender ports. You can:

- Import AUTOSAR sender port data elements for which an invalidation policy is configured.
- Use Simulink to configure an invalidation policy for AUTOSAR sender port data elements.
- Generate C and arxml code for AUTOSAR sender port data elements for which an invalidation policy is configured.

For more information, see [Configure AUTOSAR Sender Port for Data Element Invalidation](#).

End-to-end protection for sender and receiver ports

AUTOSAR end-to-end (E2E) protection for sender and receiver ports is based on the E2E library. E2E is a C library that you can use to transmit data securely between AUTOSAR components. End-to-end

protection adds additional information to an outbound data packet. The component receiving the packet can then verify independently that the received data packet matches the sent packet. Potentially, the receiving component can detect errors and take action.

For easier integration of AUTOSAR generated code with AUTOSAR E2E solutions, R2015b adds support for AUTOSAR E2E protection. You can:

- Import AUTOSAR sender port and receiver ports for which E2E protection is configured.
- Use Simulink to configure an AUTOSAR sender or receiver port for E2E protection.
- Generate C and arxml code for AUTOSAR sender and receiver ports for which E2E protection is configured.

For more information, see [Configure AUTOSAR S-R Interface Port for End-To-End Protection](#).

DataReceiveErrorEvent for receiver ports

In AUTOSAR sender-receiver communication between software components, the run-time environment (RTE) raises a `DataReceiveErrorEvent` when the communication layer reports an error in data reception by the receiver component. For example, the event can indicate that the sender component failed to reply within an `aliveTimeout` limit, or that the sender component sent invalid data.

R2015b adds support for creating `DataReceiveErrorEvents` in AUTOSAR receiver components. You can:

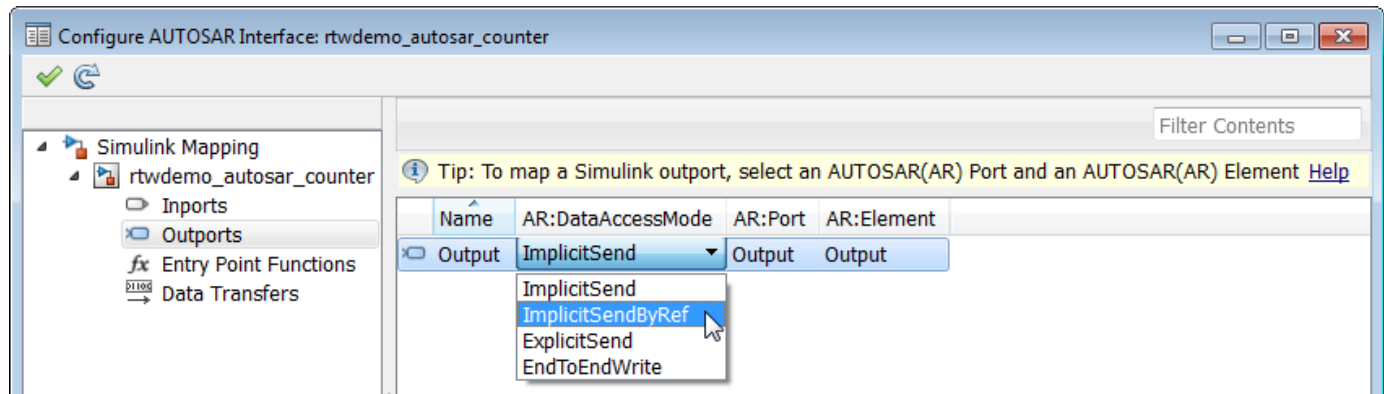
- Import an AUTOSAR `DataReceiveErrorEvent` definition.
- Use Simulink to define a `DataReceiveErrorEvent`.
- Generate arxml code for AUTOSAR receiver ports for which a `DataReceiveErrorEvent` is configured.

For more information, see [Configure AUTOSAR Receiver Port for DataReceiveErrorEvent](#).

Rte_IWriteRef for sender ports

In R2015b, you can leverage the `Rte_IWriteRef` API (AUTOSAR Release 4.x) when writing to AUTOSAR sender ports. `Rte_IWriteRef` returns a reference to the write data, which the runnable code can use to directly update the corresponding data elements. The API provides constant execution time for writes of any data element type, including structure and matrix data.

If you want AUTOSAR sender port data to be written using `Rte_IWriteRef` rather than `Rte_IWrite`, configure the corresponding Simulink root outputport for `ImplicitSendByRef` access. For example, suppose that you open the example model `rtwdemo_autosar_counter`, and change the data access mode of its root outputport, `Output`, from `ImplicitSend` to `ImplicitSendByRef`.



When you generate code, in `rtwdemo_autosar_counter.c`, the model step function uses `Rte_IWriteRef` to write the sender port data.

```
void Runnable_Step(void)
{
    ...
    int32_T *tmp;
    tmp = Rte_IWriteRef_Runnable_Step_Output_Output();
    ...
    /* Output: '<Root>/Output' incorporates:
     * Gain: '<S1>/Gain'
     * Inport: '<Root>/Input'
     * ...
     */
    *tmp = Rte_Prm_rCounter_K() * Rte_IRead_Runnable_Step_Input_Input();
    ...
}
```

AUTOSAR client-server modeling

R2015b enhances AUTOSAR client-server modeling with support for:

- AUTOSAR error status
- AUTOSAR NVRAM memory services

AUTOSAR error status

In R2015b, you can model AUTOSAR application error status for client-server error handling. In Simulink, you can:

- Import `arxml` code that implements client-server error handling.
- Configure error handling for a client-server interface.
- Generate C and `arxml` code for client-server error handling.

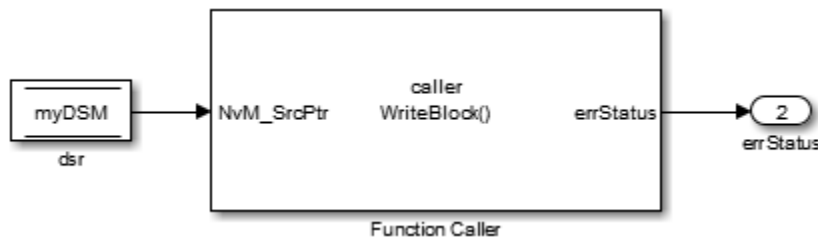
For more information, see [Configure AUTOSAR Client-Server Error Handling](#).

AUTOSAR NVRAM memory services

R2015b provides improved support for AUTOSAR nonvolatile RAM memory (NvM) services, including the NvM APIs `ReadBlock`, `WriteBlock`, and `RestoreBlockDefaults`. On ECU hardware startup or shutdown, or in response to an explicit read or write request, NvM services store data needed by the AUTOSAR software component. To better support NvM services, Embedded Coder:

- Imports and exports the `void` pointer data type that the NvM APIs use.
- Imports and exports asynchronous-server call points for calling the NvM APIs. The `arxml` importer creates Function Caller blocks to model the call points.
- Enforces constraints for modeling the RAM block required for NvM API calls. A data store memory block models the RAM block, and must directly connect to the Function Caller block.
- Generates C code that provides the RAM block to the NvM API calls without creating a local buffer.

Here is an example of Data Store Read and Function Caller blocks that model an asynchronous call to the NvM WriteBlock service.



The generated C code calls the NvM WriteBlock service with the global RAM block as an argument.

```
appErrType = Rte_Call_WriteBlock_client_WriteBlock(Rte_Pim_myDSM());
```

Compatibility Considerations


Enforcing the new modeling constraints can generate errors for models that previously did not get errors. For example, if a Function Caller block configured to call an AUTOSAR NvM API does not directly connect to a data store block, Embedded Coder generates an error.

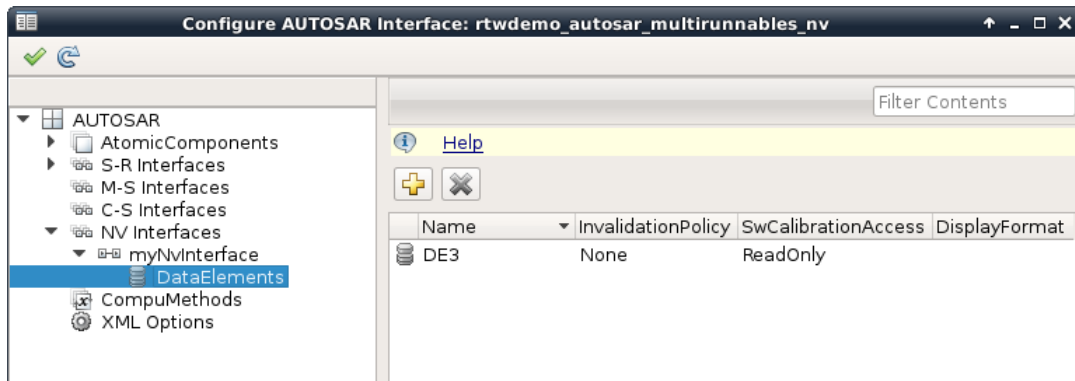
AUTOSAR nonvolatile data communication modeling

In R2015b, you can model AUTOSAR nonvolatile (NV) data communication, as defined in AUTOSAR Release 4.0 or later. To implement NV data communication, AUTOSAR software components define provide and require ports that send and receive NV data. In Simulink, you can:

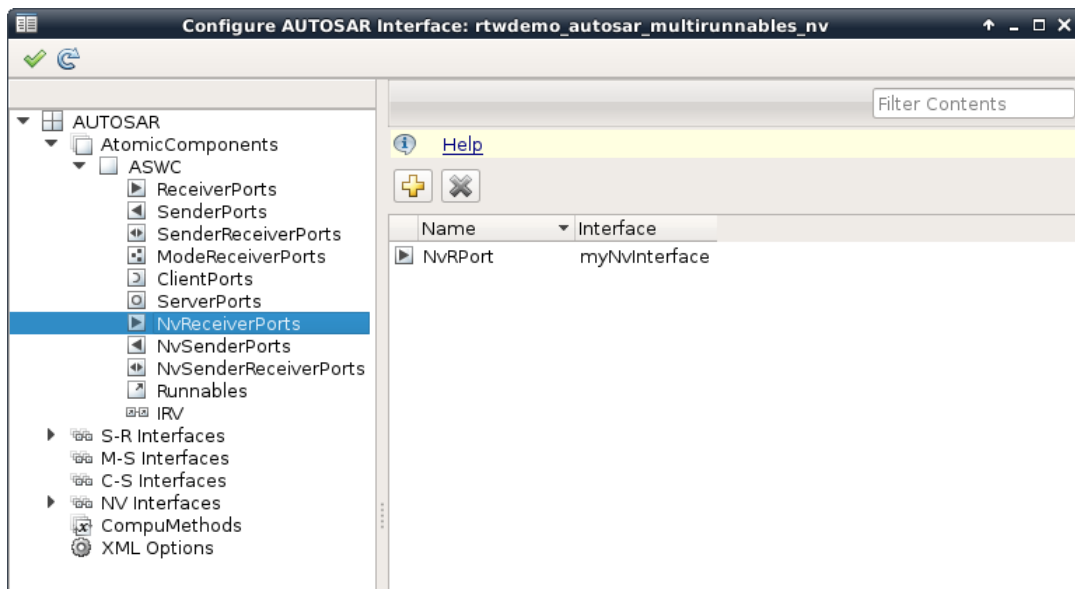
- Import AUTOSAR NV data communication definitions from `arxml` code.
- Create AUTOSAR NV data communication elements, including an NV interface and ports, and map Simulink inports and outports to AUTOSAR NV ports.
- Generate C and `arxml` code for AUTOSAR NV data communication elements.

To create NV data communication elements in Simulink:

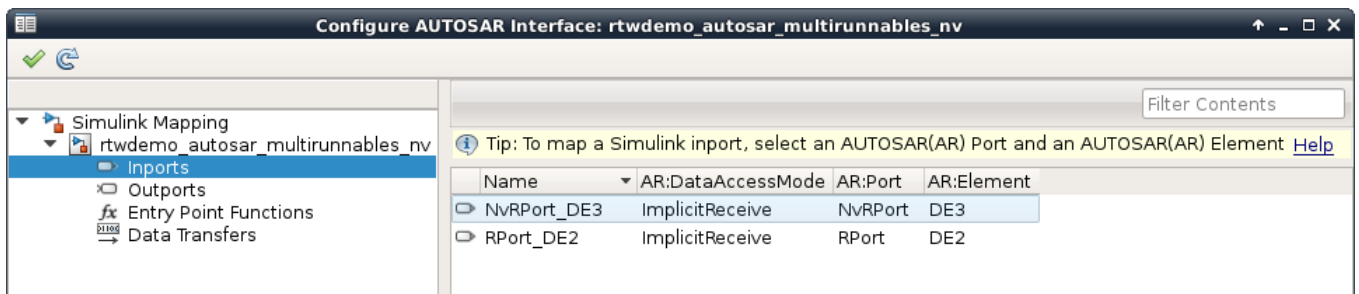
- 1 Open the Configure AUTOSAR Interface dialog box and select **AUTOSAR Properties**.
- 2 Select **NV Interfaces**. Click the **Add** icon  to create a new NV data interface. Specify its name and the number of associated NV data elements.
- 3 Select and expand the new NV interface. Select **Data Elements**, and modify the data element attributes.



- 4 In the left-hand pane of the Configure AUTOSAR Interface dialog box, under **AUTOSAR**, select **AtomicComponents**. Expand **AtomicComponents** and select an AUTOSAR component. Expand the component.
- 5 Select and use the **NvReceiverPorts**, **NvSenderPorts**, and **NvSenderReceiverPorts** views to add the NV ports you require. For each NV port, select the NV interface you created.



- 6 Switch to the Simulink mapping view. Select **Simulink-AUTOSAR Mapping**.
- 7 Select and use the **Imports** and **Exports** views to map Simulink inports and outports to AUTOSAR NV ports. For each inport or outport, select an AUTOSAR port, data element, and data access mode.



To programmatically configure AUTOSAR NV data communication elements, use the AUTOSAR property and mapping functions. For example, the following MATLAB code adds an AUTOSAR NV data interface and an NV receiver port to an open model. It then maps a Simulink inport to the AUTOSAR NV receiver port.

```
% Add AUTOSAR NV data interface myNvInterface with NV data element DE3
arProps = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables_nv');
addPackageableElement(arProps, 'NvDataInterface', '/pkg/if', 'myNvInterface');
add(arProps, 'myNvInterface', 'DataElements', 'DE3');

% Add AUTOSAR NV receiver port NvRPort, associated with myNvInterface
add(arProps, 'ASWC', 'NvReceiverPorts', 'NvRPort', 'Interface', 'myNvInterface');

% Map Simulink inport NvRPort_DE3 to AUTOSAR port/element pair NvRPort and DE3
slMap = autosar.api.getSimulinkMapping('rtwdemo_autosar_multirunnables_nv');
mapInport(slMap, 'NvRPort_DE3', 'NvRPort', 'DE3', 'ImplicitReceive');
```

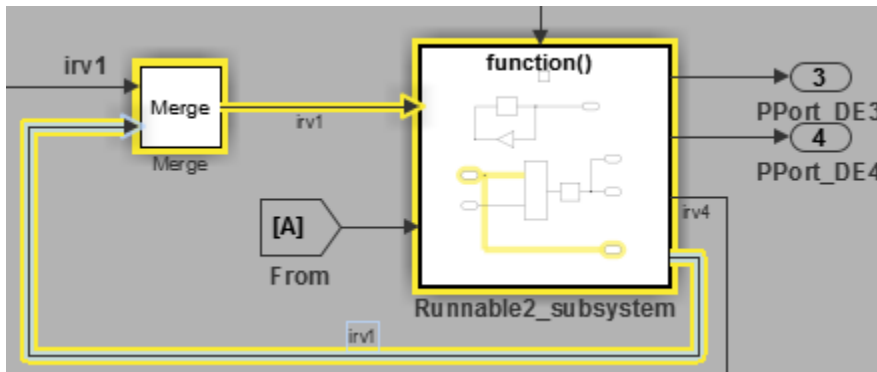
AUTOSAR component behavior modeling

R2015b enhances AUTOSAR component behavior modeling with support for:

- IRVs in feedback loops
- Constant memory with `const` or `volatile` type qualifiers

IRVs in feedback loops

Simulink modeling now supports an AUTOSAR inter-runnable feedback loop, that is, AUTOSAR runnables accessing an AUTOSAR inter-runnable variable (IRV) with both read and write access. For example, in the figure, Runnable2_subsystem can read and write `irv1`. (Signal `irv1` is shown in **Highlight Signal to Source** view.) In previous releases, the software flagged an error for this modeling pattern.

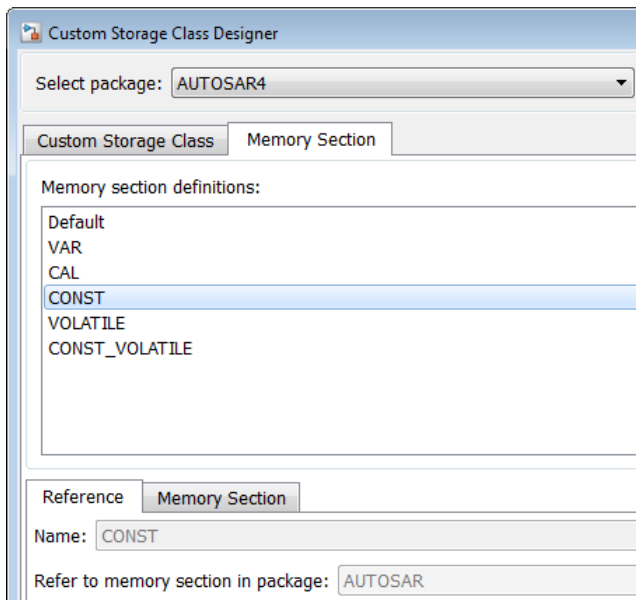


Constant memory with `const` or `volatile` type qualifiers

When modeling an AUTOSAR constant or static memory variable (AUTOSAR schema 4.x), you can now generate `const`, `volatile`, or `const volatile` qualifiers in C code to control data access.

You model AUTOSAR constant memory and static memory using `AUTOSAR4.Parameter` and `AUTOSAR4.Signal` data objects with a global storage class. Optionally, you can create custom storage classes and memory sections to customize the code generated for the global memory data, as described in Design Custom Storage Classes and Memory Sections. The AUTOSAR4 data class package now provides `CONST`, `VOLATILE`, and `CONST_VOLATILE` memory section definitions for configuring the `const`, `volatile`, and `const volatile` qualifiers. You can reference the new

memory-section values in `cscdesigner` to set up memory sections, and then reference the values from within `AUTOSAR4.Parameter` and `AUTOSAR4.Signal` data objects.



AUTOSAR COM_AXIS lookup table modeling

R2015b provides the ability to model common axis (COM_AXIS) lookup tables for AUTOSAR applications. You can:

- Import AUTOSAR calibration parameters of category CURVE, MAP, CUBOID, and COM_AXIS from `arxml` files into Simulink. The importer creates corresponding model content, including n-D Lookup Table blocks and parameter objects.
- Use Simulink to create a COM_AXIS table and configure it for AUTOSAR run-time calibration.
- Export COM_AXIS lookup table information in `arxml` code, including calibration parameters of category CURVE, MAP, CUBOID, and COM_AXIS.

For more information, see [Calibration Parameters for COM_AXIS Lookup Tables and Configure COM_AXIS Lookup Table for Measurement and Calibration](#).

Embedded Coder model templates

In R2015b, Embedded Coder templates provide you with a starting point for quickly developing models for code generation. Embedded Coder templates provide starting models for the following applications:

- Code Generation System. Create a model to get started with code generation.
- Exported functions. Create a model for generating code from function-call subsystems.
- Fixed-step, multirate. Create a fixed-step model with multiple rates for production code generation.
- Fixed-step, single-rate. Create a fixed-step model with a single rate for production code generation.

In the templates, traceability and reporting are turned on so that you can easily evaluate your generated code. The model configuration settings are based on code generation objectives for execution efficiency and traceability.

For more information on using the templates, see [Create a Model Configured for Code Generation Using Embedded Coder Templates](#).

Removal of uncalled Disable functions from generated code

In R2015a, the code generator created `Disable` functions that the generated code did not call. In R2015b, the code generator does not create uncalled `Disable` functions, except in the following cases:

- A model containing a Model Reference block or an S-function block.
- You are exporting code for a function-call subsystem.

In these cases, the code generator creates `Disable` functions that the generated code might not call. The code generator does not have enough information to determine whether the generated code requires the `Disable` functions.

This enhancement reduces code size and ROM consumption.

Enhancement to option for generating preprocessor conditionals

Previously, the **Code Generation > Interface** pane of the Model Configuration Parameters dialog box contained the option to **Generate preprocessor conditionals**. When you set this option to `Enable all` or `Disable all`, the global setting overrode the local setting **Generate preprocessor conditionals** that you specified on Variant Subsystem or Variant Model blocks.

In R2015b, the following enhancements have been made to the **Generate preprocessor conditionals** option.

- The option is now local to Variant Subsystem and Variant Model blocks. The global option has been removed from the Model Configuration Parameters dialog box. This enhancement eliminates the confusion regarding which option, global or local, is active.
- When you select this option, Simulink analyzes variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of variant choices.
- The Model Advisor now includes a check to identify models whose global **Generate preprocessor conditionals** option is set to `Enable all` or `Disable all`. The check provides instructions on how to migrate the global setting to individual variant blocks.

Compatibility Considerations

- Previously, when the **Generate preprocessor conditionals** option was switched on, Simulink analyzed variant choices only during the code generation phase. Now, Simulink performs this analysis during the update diagram phase. As a result, errors that you would normally see during code generation appear earlier, during an update diagram.
- If you load a pre-R2015b model whose global **Generate preprocessor conditionals** option was set to `Enable all` or `Disable all`, Embedded Coder generates a warning. The warning

contains instructions on how to migrate the global setting to individual variant blocks. After the migration is complete, the affected variant blocks behave as they did in previous releases.

Data, Function, and File Definition

Tokenized function names for custom storage class GetSet

When you apply the custom storage class `GetSet` to a signal, block parameter, or state, you specify the names of functions to read or write the data in the generated code. In R2015b, when you identify these function names by specifying the properties `GetFunction` and `SetFunction`, you can use the token `$N`. The generated code calls the functions that you specify by replacing the token with the name of the signal, parameter, or state.

For example, if you specify the property `GetFunction` as `get_$N_data` for a signal named `mySig`, the generated code calls the function `get_mySig_data` to access the signal.

When you apply the custom storage class `GetSet` to new signals, parameters, or states, the default `GetFunction` value is `get_$N`, and the default `SetFunction` value is `set_$N`.

For more information, see [Access Data Through Functions with Custom Storage Class GetSet](#).

Code Generation

Embedded Coder Quick Start: Quickly configure model to generate reusable and efficient code

The Embedded Coder Quick Start tool helps you quickly generate readable, efficient code from your Simulink model. To start the tool, from the model window, select **Code > C/C++ > Embedded Coder Quick Start**.

You must select preferences about your code generation objectives and target environment. The tool then validates your choices against the model and presents the parameter changes required to generate code. If you choose to generate code, the tool executes the changes to your configuration set and generates the code.

When code generation is complete, links to the documentation present possible next steps, such as customizing your generated code and refining code optimizations.

For more information, see [Generate Code with the Embedded Coder Quick Start Tool](#).

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 `ode-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 to preserve model-to-code traceability.

Two exceptions to the character escape sequence replacement scheme are:

- Comments in code generation template (`.cgt`) files
- Variables and function names in Target Language Compiler (`.tlc`) files

By default, code generation template files do not contain encoding information. The operating system reads the files in the user default encoding, regardless of the encoding that you use to write the file. You can enable escape sequence replacements by adding the following token to your template file:

```
<encodingIn = "encoding">
```

Replace *encoding* with a string that names a standard character encoding scheme, such as UTF-8, ISO-8859-1, or windows-1251.

Target Language Compiler 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.

MISRA C:2012 code generation objective

The Code Generation Advisor includes a new objective for MISRA C:2012 guidelines. Setting the objective increases the likelihood of generating MISRA C:2012 compliant code. The MISRA C:2012 guideline objective replaces the MISRA-C:2004 guideline.

For more information, see Configure Model for Code Generation Objectives Using Code Generation Advisor.

Compatibility Considerations

The MISRA C:2012 guideline objective replaces the MISRA-C:2004 guideline. If you use the command-line to set the `ObjectivePriorities` parameter to `MISRA-C:2004 guideline`, Embedded Coder will use the MISRA C:2012 guideline objective.

AUTOSAR arxml round-trip

R2015b enhances the AUTOSAR arxml round-trip workflow with support for:

- Editable AUTOSAR display format for calibration
- Configurable export of AUTOSAR internal data constraints
- AUTOSAR reference bases
- AUTOSAR-typed per-instance memory import

Editable AUTOSAR display format for calibration

AUTOSAR display format specifications control the width and precision display for calibration and measurement data. In R2015b, you can import and export AUTOSAR display format specifications, and edit the specifications in Simulink. You can specify display format for the following AUTOSAR data objects and elements:

- Signal and parameter data objects (AUTOSAR and AUTOSAR4 classes)
- Inter-runnable variables
- Sender-receiver interface data elements
- Client-server interface operation arguments
- `CompuMethods`

For more information, see Configure AUTOSAR Display Format for Measurement and Calibration.

Configurable export of AUTOSAR internal data constraints

In releases before R2015b, you could not control the export or packaging of AUTOSAR internal data constraints from Simulink. Code generation exported internal data constraints to AUTOSAR package `DataConstrs` at a fixed location under the AUTOSAR datatype package.

In R2015b, you can enable or disable export of AUTOSAR internal data constraints. Export now is disabled by default. Optionally, you can specify the name and path of an AUTOSAR package into which internal data constraints are exported. For more information, see [Configure AUTOSAR Internal Data Constraints Export](#).

AUTOSAR reference bases

Embedded Coder now can import AUTOSAR reference bases from arxml code into a model. Reference bases, which are defined in AUTOSAR Release 4.0, allow the use of relative paths in AUTOSAR specifications of packageable elements. In this arxml code example, reference base CMs resolves to /pkg/Components/MyComponent/CompuMethods.

```

<REFERENCE-BASES>
  <REFERENCE-BASE>
    <SHORT-LABEL>MyComponent</SHORT-LABEL>
    <IS-DEFAULT>>true</IS-DEFAULT>
    <PACKAGE-REF DEST="AR-PACKAGE">
      /pkg/Components/MyComponent
    </PACKAGE-REF>
  </REFERENCE-BASE>
  <REFERENCE-BASE>
    <SHORT-LABEL>IFs</SHORT-LABEL>
    <IS-DEFAULT>>false</IS-DEFAULT>
    <PACKAGE-REF BASE="MyComponent" DEST="AR-PACKAGE">
      PortInterfaces
    </PACKAGE-REF>
  </REFERENCE-BASE>
  <REFERENCE-BASE>
    <SHORT-LABEL>CMs</SHORT-LABEL>
    <IS-DEFAULT>>false</IS-DEFAULT>
    <PACKAGE-REF BASE="myComponent" DEST="AR-PACKAGE">
      CompuMethods
    </PACKAGE-REF>
  </REFERENCE-BASE>
</REFERENCE-BASES>
<APPLICATION-PRIMITIVE-DATA-TYPE>
  <SHORT-NAME>t_bool_OneToOne</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    ...
    <SW-CALIBRATION-ACCESS>NOT-ACCESSIBLE</SW-CALIBRATION-ACCESS>
    <COMPU-METHOD-REF BASE="CMs" DEST="COMPU-METHOD">
      OneToOne
    </COMPU-METHOD-REF>
    ...
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>

```

AUTOSAR-typed per-instance memory import

R2014a introduced modeling and code generation support for AUTOSAR-typed per-instance memory (arTypedPerInstanceMemory) in Simulink models. With R2015b, you can import arTypedPerInstanceMemory definitions from arxml code into a model. When you import an arTypedPerInstanceMemory definition, the arxml importer:

- Creates an AUTOSAR.Signal data object, sets its **Storage class** to PerInstanceMemory, and configures per-instance memory attributes.

- Creates a Data Store Memory block that references the `AUTOSAR.Signal` object.

For more information, see [Per-Instance Memory and Configure AUTOSAR Per-Instance Memory](#).

Toolchain controls for AUTOSAR code generation

The AUTOSAR target (`autosar.tlc`) now supports toolchain controls for C code generation. When you select the AUTOSAR target, the Configuration Parameter dialog box displays toolchain parameters rather than the template makefile (TMF) parameters previously displayed. You can more flexibly configure AUTOSAR code generation, for example, for processor-in-the-loop (PIL) verification, or to leverage a toolchain-based hardware support package.

The screenshot shows a configuration dialog with two main sections: 'Target selection' and 'Build process'. In 'Target selection', 'System target file' is 'autosar.tlc', 'Language' is 'C', and 'Description' is 'AUTOSAR'. In 'Build process', under 'Toolchain settings', 'Toolchain' is 'GNU Tools for ARM Embedded Processors' and 'Build configuration' is 'Faster Builds'. There are 'Validate' and 'Show settings' buttons. A note below 'Faster Builds' says 'Minimize compilation and linking time'.

Other targets that support toolchain controls include the ERT targets `ert.tlc` and `ert_shrllib.tlc`.

AUTOSAR RTE file generation enhanced for SIL and PIL

Building an AUTOSAR model generates RTE (run-time environment) files into the `stub` subfolder of the model build folder. The RTE files have `.c` and `.h` extensions, and contain stub implementations of the AUTOSAR `Rte` functions. The stub implementations can be used to test the generated C code in Simulink, for example, in software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations of the component under test. When the generated code ultimately is deployed in the AUTOSAR RTE, you replace the RTE stub files with externally-generated RTE files.

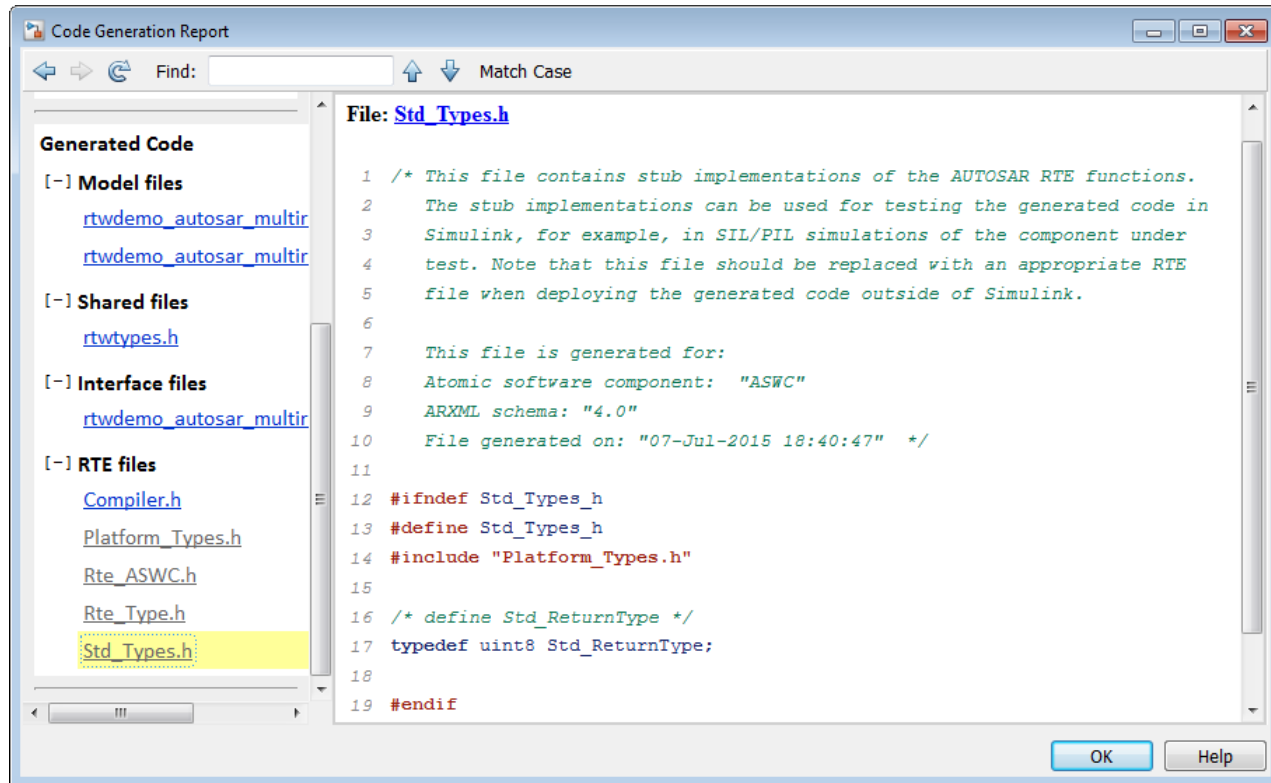
R2015b enhances the generated RTE stub files in many respects. The build generates most of the same RTE stub files as before, but with improved content:

- More closely reflects the AUTOSAR element content of the model.
- More closely resembles what an external RTE Generator creates.
- Better descriptions of content and possible uses.

R2015b also generates new stub files, `Std_Types.h`: and `Platform_Types.h`:

- `Std_Types.h` is a standard AUTOSAR file that defines basic data types.
- `Platform_Types.h` maps AUTOSAR base types to platform types.

- Std_Types.h includes Platform_Types.h, and is included by Rte_Type.h.



Lookup table blocks with new even spacing specification generate AUTOSAR compatible IFX library routines

As of R2015b, lookup table blocks generate AUTOSAR compatible IFX library routines. Lookup table blocks were enhanced to support a new specification for even-spacing breakpoints, which supports and generates AUTOSAR IFX routines.

For more information, see Code Replacement for AUTOSAR.

Control use of signed shifts in generated code

You can now control the use of signed right shifts in your generated code. Some coding standards do not allow bitwise operations on signed integers. Disabling the use of signed shifts in generated code increases the likelihood of compliance with MISRA. When you specify that signed right shifts should not be used in your generated code, the software replaces signed shifts with a call to a function that performs the operation without the use of signed shifts.

To specify that the code generator not use signed right shifts, in the Configuration Parameters dialog box, on the **Code Generation** > **Code Style** pane, clear Allow right shifts on signed integers or set the parameter EnableSignedRightShifts to off.

Code generation report with operator traceability

In R2015b, the HTML code generation report provides traceability between operators in the generated code and Simulink blocks. In the HTML report window, click an operator hyperlink to highlight the source block in the model. In the model, right-click an operator block. From the context menu, select **C/C++ Code > Navigate to C/C++ Code**. This selection highlights the generated code for the block in the HTML code generation report. Operator traceability information is included in the **Traceability Report** section of the code generation report. This information is also in the generated traceability matrix.

The code generation report does not provide traceability between operators and Stateflow or MATLAB Function blocks.

Deployment

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.

Code Replacement Tool uses simplified specification

As of R2015b, where possible, the Code Replacement Tool creates code replacement table entries by using an approach that significantly reduces the amount of relevant code. Instead of using separate function calls to create the entry, conceptual arguments, and implementation arguments, the tool uses the `createCRLEntry` function to create entries from conceptual and implementation argument string specifications. The tool continues to use the more verbose approach for entries that involve:

- C++ implementations
- Data alignment

- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, build information)
- Semaphore and mutex function replacements

For more information, see `createCRLEntry` and Define Code Replacement Mappings.

Code replacement support for new lookup table breakpoint specification

In R2015b, n-D Lookup Table and Prelookup blocks support a new specification for evenly spaced breakpoints. Rather than specifying breakpoints as a vector, for n-D Lookup Table blocks, you can enter values for **First point** and **Spacing** parameters for each dimension of the breakpoint data. For Prelookup blocks, you can enter values for **First point**, **Spacing**, and **Number of points**. The code replacement software supports this new breakpoint specification through alternative conceptual function signatures for n-D Lookup Table and Prelookup blocks.

For more information, see n-D Lookup Table, Prelookup, and Lookup Table Function Code Replacement.

Support for Analog Devices VisualDSP++ will be removed

Support for Analog Devices® VisualDSP++® will be removed in a future release.

Performance

RAM/ROM Optimization Improvements: Generate more efficient code using reusable storage class and converting data copies to pointer assignments

Reuse input and output of a block or subsystem

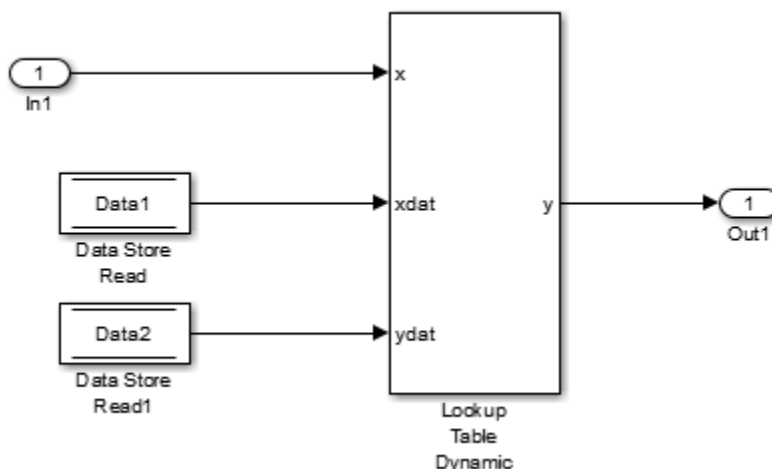
Previously, if a pair of model block I/O signals shared the same Reusable storage class specification, the code generator reused the root I/O signals in the generated code. In R2015b, this optimization extends to the input and output signals at a block or subsystem boundary if the input and output arguments have the same data types and sampling rates. This optimization can reduce the number of global variables, data copies, and RAM/ROM consumption in the generated code. For more information, see [Buffer Reuse Around a Block or Subsystem Boundary](#)

More efficient code for large data sets

Previously, for many data transfers involving vector signals, the code generator replaced a `for` loop controlled array element assignment with a `memcpy` function call. In R2015b, the code generator can replace a `for` loop controlled array element assignment that is inside of an `if-else` statement with a `memcpy` function call. The code generator can replace multiple array element assignments inside of a `for` loop with `memcpy` function calls. These optimizations improve execution speed.

In R2015b, the code generator attempts to replace `for` loop controlled array element assignments and `memcpy` function calls with pointer assignments. Because this optimization eliminates full array data copies, it improves execution speed and saves stack space.

Consider the following model named `dynamicLookup`. The Data Store Read blocks are copying data from their named data stores (`Data1` or `Data2`) to the input buffers of the Lookup Table.



In R2015a, the code generator produced this code:

```

/* Model step function */
void dynamicLookup_step(void)

```

```

{
  /* local block i/o variables */
  real32_T rtb_DataStoreRead[10];
  uint16_T rtb_DataStoreRead1[10];
  int32_T i;

  /* DataStoreRead: '<>/Data Store Read' */
  for (i = 0; i < 10; i++) {
    rtb_DataStoreRead[i] = Data1[i];

    /* DataStoreRead: '<>/Data Store Read1' */
    rtb_DataStoreRead1[i] = Data2[i];
  }
  ...
  LookUp_real_TU16_real32_T( &(dynamicLookup_Y.Out1), &rtb_DataStoreRead1[0],
    dynamicLookup_U.In1, &rtb_DataStoreRead[0], 9U);
}

```

In R2015b, the code generator produces this code:

```

/* Model step function */
void dynamicLookup_step(void)
{
  real32_T *rtb_DataStoreRead_0;
  uint16_T *rtb_DataStoreRead1_0;
  /* DataStoreRead: '<Root>/Data Store Read' */
  rtb_DataStoreRead_0 = (&(Data1[0]));
  /* DataStoreRead: '<Root>/Data Store Read1' incorporates:
  * DataStoreRead: '<Root>/Data Store Read'
  */
  rtb_DataStoreRead1_0=(&(Data2[0]));
  ...
  LookUp_real_TU16_real32_T( &(dynamicLookup_Y.Out1), rtb_DataStoreRead1_0,
    dynamicLookup_U.In1, rtb_DataStoreRead_0, 9U);
}

```

In R2015a, the generated code contains a for loop and data copies to the arrays `rtb_DataStoreRead` and `rtb_DataStoreRead1`. In R2015b, the code generator replaces the for loop with pointer assignments to the variables `rtb_DataStoreRead_0` and `rtb_DataStoreRead1_0`. For more information, see [Optimize Memory Usage for Vector Signal Assignments](#)

Live Execution Profiling: View PIL profile results during run-time

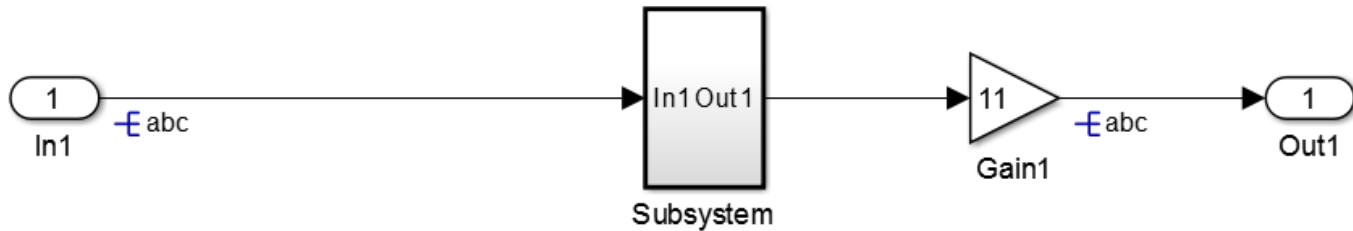
During a processor-in-the-loop (PIL) simulation, you can use the Simulation Data Inspector to view streamed task execution times. Previously, this data was available only at the end of the PIL simulation. For more information, see [View and Compare Code Execution Times](#).

Enhanced support for buffer reuse at the root-level input and output ports

Reusable custom storage class for model block input and output ports

Previously, if a pair of root-level model input and output signals used the same `Reusable` storage class specification, the code generator reused the root I/O signals in the generated code. In R2015b,

the code generator enables this optimization for models containing subsystems. This optimization can reduce data copies, global variables, and ROM/RAM consumption. For example, consider the following model named IObuffreuse.



In R2015a, the code generator produces the following code:

```

void IObuffreuse_Subsystem(const real_T rtu_In1[6], B_Subsystem_IObuffreuse_T
*localB)
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    localB->ca[i] = 4.0 * rtu_In1[i];
  }
}

void IObuffreuse_step(void)
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    abc_0[i] = abc[i];
  }

  IObuffreuse_Subsystem(abc_0, &IObuffreuse_B.Subsystem);
  for (i = 0; i < 6; i++) {
    abc[i] = 11.0 * IObuffreuse_B.Subsystem.ca[i];
  }
}

```

In R2015b, the code generator produces the following code:

```

void IObuffreuse_Subsystem(const real_T rtu_In1[6], B_Subsystem_IObuffreuse_T
*localB)
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    localB->ca[i] = 4.0 * rtu_In1[i];
  }
}

void IObuffreuse_step(void)
{
  int32_T i;
  IObuffreuse_Subsystem(&(abc[0]), &IObuffreuse_B.Subsystem);
  for (i = 0; i < 6; i++) {
    abc[i] = 11.0 * IObuffreuse_B.Subsystem.ca[i];
  }
}

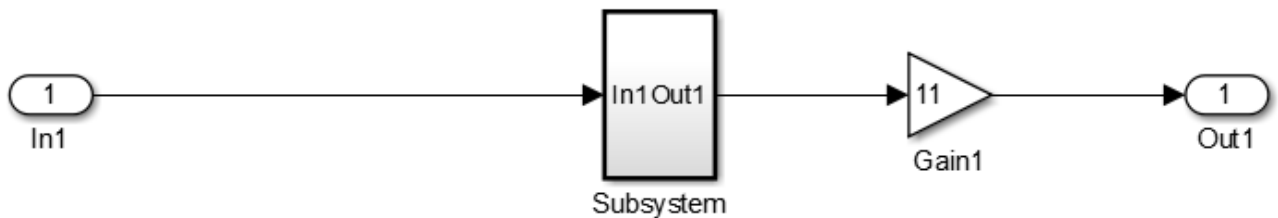
```

In R2015a, the generated code contains an additional buffer named `abc_0`. The code also contains a full array data copy from `abc_0` to `abc` in the model step function. In R2015b, the additional buffer and the full array data copy are not in the generated code.

For more information on how to configure your model to use this optimization, see [Buffer Reuse for Model Block Boundary and Unit Delay](#).

Combined input and output arguments with function prototype control

Previously, the code generator tried to reuse buffers for a pair of model step function input and output ports that were assigned the same argument name using function prototype control. This optimization can reduce data copies, global variables, and ROM/RAM consumption. In R2015b, the code generator enables this optimization for models containing subsystems. For example, consider the following model named `FPCioreuse`.



In R2015a, the code generator produces the following code:

```

void mg956114fpc2_custom(real_T arg_Inout1[6])
{
    int32_T i;
    for (i = 0; i < 6; i++) {
        arg_Inout1_0[i] = arg_Inout1[i];
    }

    FPCioReuse_Subsystem(arg_Inout1_0, &FPCioReuse_B.Subsystem);
    for (i = 0; i < 6; i++) {
        arg_Inout1[i] = 11.0 * FPCioReuse_B.Subsystem.ca[i];
    }
}
  
```

In R2015b, the code generator produces the following code:

```

void mg956114fpc2_custom(real_T arg_Inout1[6])
{
    int32_T i;
    FPCioReuse_Subsystem(arg_Inout1, &FPCioReuse_B.Subsystem);
    for (i = 0; i < 6; i++) {
        arg_Inout1[i] = 11.0 * FPCioReuse_B.Subsystem.ca[i];
    }
}
  
```

In R2015a, the code contains an additional buffer named `arg_Inout1_0`. The code also contains a full array data copy from `arg_Inout1` to `arg_Inout1_0`. In R2015b, the temporary buffer and full array data copy are not in the generated code.

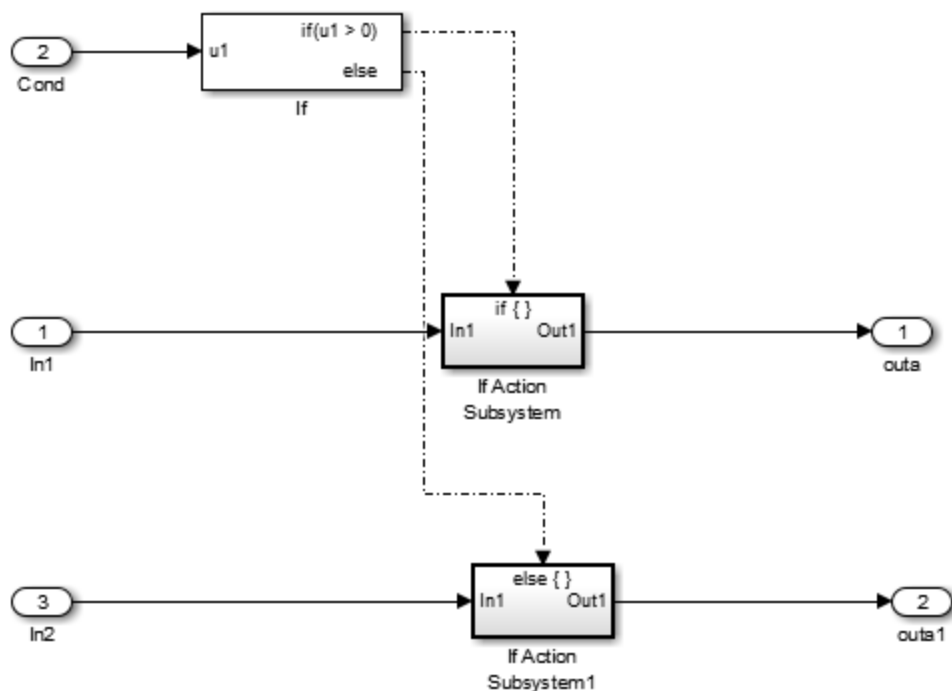
To configure model step function I/O arguments to allow buffer reuse, use either C function prototype control or C++ class interface control. For more information, see Combine Input and Output Arguments in Model Step Interface.

More efficient code for small subsystems

Previously, if a subsystem was in a model or model hierarchy more than once and the subsystem **Function packaging** was set to Auto, Embedded Coder generated a separate, reusable function with arguments.

In R2015b, if these subsystems are small and not too complex, the code generator inlines the code for each subsystem. This enhancement reduces data copies, RAM consumption, and code size. It also improves execution speed. For large-scale models containing thousands of subsystems, this enhancement saves time because you do not have to manually set **Function packaging** to Inline for each subsystem.

Consider the following model named `auto_funcpackaging`. This model contains two identical, simple subsystems named `if Action Subsystem` and `If Action Subsystem1`.



In R2015a, the code generator produced the following code:

```

void auto_funcpack_IfActionSubsystem(real_T rtu_In1,
rtDW_IfActionSubsystem_auto_fun *localDW)
{
    localDW->Gain = 4.0 * rtu_In1;
}

void auto_funcpackaging_step(void)
{
    if (auto_funcpackaging_U.Cond > 0.0) {
        auto_funcpack_IfActionSubsystem(auto_funcpackaging_U.In1,

```

```
    &auto_funcpackagin_DWork.IfActionSubsystem);
} else {
    auto_funcpack_IfActionSubsystem(auto_funcpackaging_U.In2,
    &auto_funcpackaging_DWork.IfActionSubsystem);
}

auto_funcpackaging_Y.outa =
    auto_funcpackaging_DWork.IfActionSubsystem.Gain;
auto_funcpackaging_Y.outa1 =
    auto_funcpackaging_DWork.IfActionSubsystem1.Gain;
}
```

In R2015b, the code generator produces this code:

```
void auto_funcpackaging_step(void)
{
    if (auto_funcpackaging_U.Cond > 0.0) {
        auto_funcpackaging_Y.outa = 4.0 * auto_funcpackaging_U.In1;
    } else {
        auto_funcpackaging_Y.outa1 = 4.0 * auto_funcpackaging_U.In2;
    }
}
```

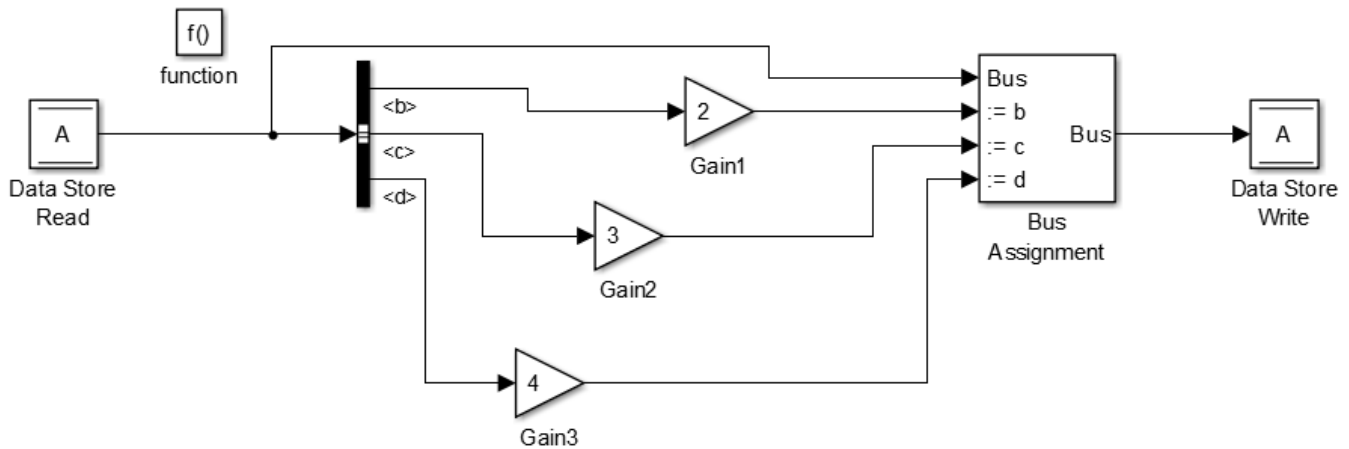
In R2015a, the code generator produced the reusable function named `auto_funcpack_IfActionSubsystem`, which is called twice in the generated code. In R2015b, because the subsystem consists of simple signal paths, the code generator inlines the code for each subsystem. For more information, see [Generate Inlined Subsystem Code](#)

More efficient code for Simulink.Bus objects

Previously, if a Data Store Memory block stored a `Simulink.Bus` object, and Data Store Read and Data Store Write blocks updated the `Simulink.Bus` object, there were extra data copies in the generated code.

In R2015b, the code generator has improved expression folding capabilities, so that these additional data copies are not in the generated code. This enhancement reduces code size and RAM consumption and increases execution speed.

For example, consider the following subsystem.



In R2015a, the code generator produced this code:

```
void f(void)
{
    real_T rtb_Gain2[170];
    real_T rtb_Gain3[190];
    int32_T i;
    for (i = 0; i < 170; i++) {
        rtb_Gain2[i] = 3.0 * rtDW.A.c[i];
    }

    for (i = 0; i < 190; i++) {
        rtb_Gain3[i] = 4.0 * rtDW.A.d[i];
    }

    for (i = 0; i < 150; i++) {
        rtDW.A.b[i] *= 2.0;
    }

    for (i = 0; i < 170; i++) {
        rtDW.A.c[i] = rtb_Gain2[i];
    }

    for (i = 0; i < 190; i++) {
        rtDW.A.d[i] = rtb_Gain3[i];
    }
}
```

In R2015b, the code generator produces this code:

```
void f(void)
{
    int32_T i;
    for (i = 0; i < 150; i++) {
        rtDW.A.b[i] *= 2.0;
    }

    for (i = 0; i < 170; i++) {
        rtDW.A.c[i] *= 3.0;
    }
}
```

```

}

for (i = 0; i < 190; i++) {
    rtDW.A.d[i] *= 4.0;
}
}

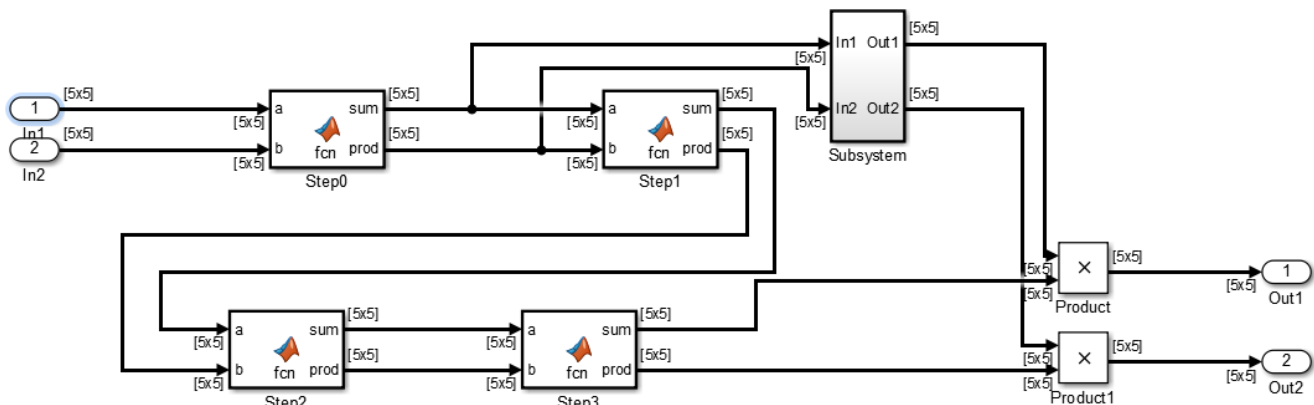
```

In R2015a, the generated code contained full array data copies from `rtb_Gain2` to `rtDW.A.c` and from `rtb_Gain3` to `rtDW.A.d`. In R2015b, if a Bus Assignment block source and destination are the same Data Store Memory block, the code generator implements the Bus Assignment block in place in the generated code. As a result, the extra data copies are not in the generated code.

Enhanced local variable reuse

In R2015b, the code generator reuses more local variables, which reduces RAM and ROM consumption.

Consider the following model named `local_reuse`. This model contains four identical MATLAB Functions and a subsystem. The signals are matrices of size `[5 5]`.



In R2015a, for the model step function, the code generator produced this code:

```

void local_reuse_step(void)
{
    real_T rtb_sum_g[25];
    real_T rtb_prod_o[25];
    real_T rtb_sum_a[25];
    real_T rtb_prod_h[25];
    real_T rtb_sum_j0[25];
    real_T rtb_prod_i[25];
    int32_T i;
    local_reuse_Step0(local_reuse_U.In1, local_reuse_U.In2, rtb_sum_g, rtb_prod_o);
    local_reuse_Subsystem(rtb_sum_g, rtb_prod_o, local_reuse_B.Gain,
        local_reuse_B.Gain1, &local_reuse_DW.Subsystem);
    local_reuse_Step0(rtb_sum_g, rtb_prod_o, rtb_sum_a, rtb_prod_h);
    local_reuse_Step0(rtb_sum_a, rtb_prod_h, rtb_sum_j0, rtb_prod_i);
    local_reuse_Step0(rtb_sum_j0, rtb_prod_i, rtb_sum_g, rtb_prod_o);
    for (i = 0; i < 25; i++) {
        local_reuse_Y.Out1[i] = local_reuse_B.Gain[i] * rtb_sum_g[i];
        local_reuse_Y.Out2[i] = local_reuse_B.Gain1[i] * rtb_prod_o[i];
    }
}

```

```

}
}

```

The generated code contained six local arrays, `rtb_sum_g`, `rtb_prod_o`, `rtb_sum_a`, `rtb_prod_h`, `rtb_sum_jo`, and `rtb_prod_i` to handle the input and output of the four MATLAB Functions.

In R2015b, for the model step function, the code generator produces this code:

```

void local_reuse_step(void)
{
    real_T rtb_sum_g[25];
    real_T rtb_prod_o[25];
    real_T rtb_sum_a[25];
    real_T rtb_prod_h[25];
    int32_T i;
    local_reuse_Step0(local_reuse_U.In1, local_reuse_U.In2, rtb_sum_g, rtb_prod_o);
    local_reuse_Subsystem(rtb_sum_g, rtb_prod_o, local_reuse_B.Gain,
        local_reuse_B.Gain1, &local_reuse_DW.Subsystem);
    local_reuse_Step0(rtb_sum_g, rtb_prod_o, rtb_sum_a, rtb_prod_h);
    local_reuse_Step0(rtb_sum_a, rtb_prod_h, rtb_sum_g, rtb_prod_o);
    local_reuse_Step0(rtb_sum_g, rtb_prod_o, rtb_sum_a, rtb_prod_h);
    for (i = 0; i < 25; i++) {
        local_reuse_Y.Out1[i] = local_reuse_B.Gain[i] * rtb_sum_a[i];
        local_reuse_Y.Out2[i] = local_reuse_B.Gain1[i] * rtb_prod_h[i];
    }
}
}

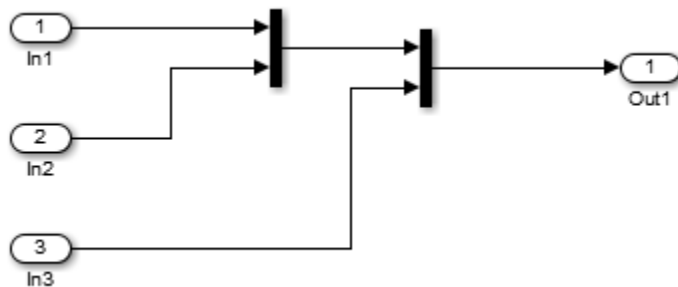
```

The generated code contains four local arrays, `rtb_sum_g`, `rtb_prod_o`, `rtb_sum_a`, and `rtb_prod_h` to handle the input and output of the four MATLAB Functions. Because the code generator reuses more local variables, there are two less local arrays than there were in R2015a.

Enhanced consolidation of for loops

Previously, the code generator tried to combine for loops that had the same number of iterations. In R2015b, the code generator combines more cases of for loops that have the same number of iterations. These for loops read and write to separate sections of the same array and write to scalar variables. This optimization conserves ROM consumption and improves execution speed.

Consider the following model named `loopfusion`. This model contains two Mux blocks that combine vector signals from three Inport blocks into an output vector signal. The three input vector signals have a dimension size of 5. The output vector signal has a dimension size of 15.



In R2015a, the code generator produced this code:

```
/* Model step function */
void loopfusion_step(void)
{
    int32_T i;

    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Inport: '<Root>/In3'
     */
    for (i = 0; i < 5; i++) {
        loopfusion_Y.Out1[i] = loopfusion_U.In1[i];
    }

    for (i = 0; i < 5; i++) {
        loopfusion_Y.Out1[i + 5] = loopfusion_U.In2[i];
    }

    for (i = 0; i < 5; i++) {
        loopfusion_Y.Out1[i + 10] = loopfusion_U.In3[i];
    }

    /* End of Outport: '<Root>/Out1' */
}
```

In R2015a, there were three for loops that wrote to three separate sections of the array, `loopfusion_Y.Out1`.

In R2015b, the code generator produces this code:

```
/* Model step function */
void loopfusion_step(void)
{
    int32_T i;

    /* Outport: '<Root>/Out1' incorporates:
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Inport: '<Root>/In3'
     */
    for (i = 0; i < 5; i++) {
        loopfusion_Y.Out1[i] = loopfusion_U.In1[i];
        loopfusion_Y.Out1[i + 5] = loopfusion_U.In2[i];
        loopfusion_Y.Out1[i + 10] = loopfusion_U.In3[i];
    }

    /* End of Outport: '<Root>/Out1' */
}
```

In R2015b, there is one for loop that writes to three separate sections of the array, `loopfusion.Out1`.

Verification

Faster SIL and PIL Verification Workflow

R2015b enables faster software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification by providing:

- Model block SIL/PIL and SIL/PIL block support for fast restart — You can tune parameters and run simulations without model recompilation.
- Model block SIL/PIL support for Accelerator mode — If you have a model with Model blocks in SIL/PIL mode, you can run the top-model simulation in Accelerator mode, which speeds up the simulation of components that are not in SIL or PIL mode.

For more information, see [Speed Up SIL/PIL Verification](#) .

Code generation assumptions verified during PIL simulation

The settings on the **Configuration Parameters > Hardware Implementation** pane specify target behavior, which result in the implementation of implicit assumptions in the generated code. Incorrect settings can lead to:

- Suboptimal code
- Code execution failure, incorrect code output, and nondeterministic code behavior

At the start of a PIL simulation, the software verifies the **Hardware Implementation** pane settings with reference to the target hardware. The software checks:

- The correctness of settings. For example, the integer bit length in the **Number of bits: int** field.
- Whether the settings are optimized. For example, the rounding of signed integer division in the **Signed integer division rounds to** field.

If required, the software generates warnings and errors.

SIL and PIL support for C++ class root-level I/O access methods

The **Configuration Parameters > Code Generation > Interface > External I/O access** parameter (`GenerateExternalIOAccessMethods`) specifies whether to generate root-level I/O signal access methods for a C++ class. R2015b provides SIL and PIL simulation support for these parameter values:

- **Structure-based method** — Code generator produces noninlined, structure-based access methods.
- **Inlined structure-based method** — Code generator produces inlined, structure-based access methods.

Previously, SIL and PIL simulations supported only access methods that were not structure-based.

For more information, see [External I/O access and Configure Step Method for Model Class](#).

Removal of Generate code only parameter restriction

You can run top-model and Model block SIL and PIL simulations even if you select the **Generate code only** (GenCodeOnly) parameter. Previously, running the SIL and PIL simulations with the parameter produced an error. For a SIL or PIL block, the restriction still applies. For additional **Generate code only** enhancements, see Smarter Code Regeneration: Regenerate code only when model settings that impact code are modified.

Removal of scheduling limitations that caused algebraic loops

In R2015b, the internal scheduling of messages between host and target in a SIL or PIL simulation is modified. This modification removes the S-function scheduling limitations that previously caused algebraic loops in SIL and PIL simulations.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.8

New Features

Compatibility Considerations

Code Generation from MATLAB Code


Indent style and size control for generated C/C++ code

You can control the indent style and size in C/C++ code generated from MATLAB code.

You can specify the K&R indent style or the Allman indent style. The K&R style places the opening brace of a control statement on the same line as the control statement. The Allman style places the opening brace on its own line at the same indentation level as the control statement.

Indent size is the number of characters per indentation level.

To specify the indent style and size using the MATLAB Coder app:

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
- 2 Set **Build type** to one of the following:
 - Source Code
 - Static Library (.lib)
 - Dynamic Library (.dll)
 - Executable (.exe)
- 3 Click **More Settings**.
- 4 On the **All Settings** tab, under **Advanced**, set **Indent style** to K&R or Allman.
- 5 On the **All Settings** tab, under **Advanced**, set **Indent size** to an integer from 2 to 8.

To specify the indent style and size using the command-line interface:

- 1 Create a code configuration object for 'lib', 'dll', or 'exe'. For example:

```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```
- 2 Set the `IndentStyle` property to 'K&R' or 'Allman'. For example:

```
cfg.IndentStyle = 'Allman';
```
- 3 Set the `IndentSize` property to an integer from 2 to 8. For example:

```
cfg.IndentSize = 4;
```


See Specify Indent Style for C/C++ Code.

Improved MISRA-C compliance for bitwise operations on signed integers

In previous releases, MATLAB Coder replaced multiplication by powers of two with signed left bitwise shifts. In R2015a, to increase the likelihood of compliance with MISRA C, you can disable this replacement. MISRA rule 12.7 does not allow bitwise operations on signed integers.

To specify that MATLAB Coder not replace multiplication by powers of two with signed left bitwise shifts:

- Using the MATLAB Coder app:

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
 - 2 Set **Build type** to one of the following:
 - Source Code
 - Static Library (.lib)
 - Dynamic Library (.dll)
 - Executable (.exe)
 - 3 Click **More Settings**.
 - 4 On the **Code Appearance** tab, clear the **Use signed shift left for fixed-point operations and multiplication by powers of 2** check box.
- Using the command-line interface:
 - 1 Create a code configuration object for 'lib', 'dll', or 'exe'. For example:


```
cfg = coder.config('lib','ecoder',true); % or dll or exe
```
 - 2 Set the EnableSignedLeftShifts property to false. For example:


```
cfg.EnableSignedLeftShifts = false;
```

See Control Signed Left Shifts in Generated Code.

Improved MISRA-C type cast compliance

You can specify the casting mode that MATLAB Coder uses for data type casts in the generated C/C++ code. You can specify these modes:

Casting Mode	Description
Nominal	Nominal casting mode is the default casting mode. The generated C/C++ code uses the default C compiler data type casting. When you do not have special data type information requirements, choose this option.
Standards Compliant	Generated C/C++ code has data type casts that conform to MISRA standards. The MISRA data type casting mode eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations.
Explicit	Generated C/C++ code has explicit data type casts. Explicit data type casts provide information about the amount of memory that the variable uses and the level of precision for calculations using the variable.

To specify the casting mode using the MATLAB Coder app:

- 1 On the **Generate Code** page, to open the **Generate** dialog box, click the **Generate** arrow .
- 2 Click **More Settings**.

- 3 On the **All Settings** tab, under **Advanced**, set **Casting mode** to Nominal, Standards Compliant, or Explicit.

To specify the casting mode using the command-line interface:

- 1 Create a code configuration object for 'lib', 'dll', or 'exe'. For example:

```
cfg = coder.config('lib', 'ecoder', true); % or dll or exe
```
- 2 Set the CastingMode property to 'Nominal', 'Standards', or 'Explicit'. For example:

```
cfg = CastingMode = 'Standard';
```

See Control Data Type Casts in Generated Code.

Model Architecture and Design

AUTOSAR improvements including multi-runnable modeling and code efficiency

R2015a provides many enhancements to Simulink modeling of AUTOSAR elements and AUTOSAR code generation. Highlights include:

- AUTOSAR multi-runnable modeling using Simulink rate-based multitasking
- Improved traceability for AUTOSAR RTE implicit read

For more information about AUTOSAR-related enhancements in R2015a, see:

- Under Model Architecture and Design:
 - “AUTOSAR multi-runnable modeling using Simulink rate-based multitasking” on page 17-6
 - “Enhanced modeling with AUTOSAR system constants” on page 17-6
 - “AUTOSAR CompuMethod enhancements” on page 17-7
- Under Code Generation:
 - “Improved traceability for AUTOSAR RTE implicit read” on page 17-12
 - “Configurable aliveTimeout value for AUTOSAR ports” on page 17-13
 - “AUTOSAR calibration parameter export for COM_AXIS lookup tables” on page 17-13

Combined input/output arguments with function prototype control

In R2015a, the code generator tries to reuse buffers for a pair of model step function input/output ports assigned the same argument name using function prototype control. The corresponding inport and outport blocks must have the same data type and sampling rate. This reuse can eliminate buffers in the generated code.

To configure model step function I/O arguments to allow buffer reuse, use either C function prototype control or C++ class interface control. For more information, see Combine Input and Output Arguments in Model Step Interface.

Improved MISRA-C compliance for bitwise operations on signed integers

You can specify that the code generator not replace multiplications by powers of two with signed bitwise shifts, increasing the likelihood of generating code that is compliant with MISRA-C. MISRA rule 12.7 does not allow bitwise operations on signed integers. Previously, the code generator replaced multiplications by powers of two with signed bitwise shifts.

To specify that the code generator not replace multiplications by power of two with signed bitwise shifts, in the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, clear Replace multiplications by powers of two with signed bitwise shifts or set the parameter `EnableSignedLeftShifts` to `off`.

To improve MISRA-C compliance for bitwise operations on signed integers, run the following checks:

- Check for bitwise operations on signed integers - New check to identify blocks that contain bitwise operations on signed integers.
- Check configuration parameters for MISRA-C:2004 compliance - Enhanced check that verifies that you cleared **Code Generation > Code Style > Replace multiplications by powers of two with signed bitwise shifts**.

AUTOSAR multi-runnable modeling using Simulink rate-based multitasking

In previous releases, you modeled a multi-runnable AUTOSAR software component using Simulink function-call subsystems or Simulink Function blocks at the top level of a model. In R2015a, you can model a multi-runnable AUTOSAR software component using Simulink rate-based multitasking. Using this approach, you can:

- Create an AUTOSAR software component with multiple periodic runnables in Simulink.
- Import an AUTOSAR software component with multiple periodic runnables from `arxml` into Simulink.
- Migrate an existing rate-based, multitasking Simulink model to the AUTOSAR target.

For more information, see [Multi-Runnable Software Components and Configure Multiple Runnables Using Rate-Based Multitasking](#).

Compatibility Considerations

Before R2015a, you could not configure a multitasking model for the AUTOSAR target. If you attempted to import an AUTOSAR software component with multiple periodic runnables and create a rate-based model (that is, if you invoked `arxml.importer` method `createComponentAsModel` with `CreateInternalBehavior` set to `false`), the importer would:

- Discard all but one runnable and create a rate-based, single-tasking model.
- For each AUTOSAR port, create an inport or outport and related Simulink elements even if the port was not accessed by the AUTOSAR runnable.

Performing the same import in R2015b produces different results in two respects. The importer:

- Creates a rate-based, multitasking model, rather than rate-based, single-tasking.
- For each AUTOSAR port, creates an inport or outport and related Simulink elements only if the port is accessed by an AUTOSAR runnable.

Enhanced modeling with AUTOSAR system constants

In previous releases, you could define AUTOSAR system constants (`SwSystemConstants`) in Simulink, but their use was limited to condition formulas inside variant subsystems and model references. In R2015a, you can directly reference AUTOSAR system constants in Simulink algorithms. For example, you could reference a system constant in a Gain block.

For more information, see [System Constants and Model AUTOSAR Component Behavior](#).

AUTOSAR CompuMethod enhancements

R2015a significantly enhances AUTOSAR CompuMethod related workflows in Simulink. You can:

- Configure the properties of imported AUTOSAR CompuMethods
- Create and configure AUTOSAR CompuMethods in Simulink
- Use externally-defined AUTOSAR CompuMethods
- Use externally-defined AUTOSAR Units

For more information, see [Configure AUTOSAR CompuMethods](#).

Preprocessor conditionals for single variant choice

Previously, you could not generate preprocessor conditionals if a variant subsystem in your model contained a single variant choice.

In R2015a, you can represent an empty subsystem as a variant choice. During code generation, if the empty variant choice is inactive, the generated code does not contain the `#elif` preprocessor conditional. Instead, the active variant choice is enclosed between a `#if` and an `#endif`.

Data, Function, and File Definition

Control of Boolean and data type limit identifiers in generated code

In R2015a, if you want to associate the data type limit identifiers with the data type names, you can use command-line parameters to replace these default data type limit identifiers:

- `MAX_int8_T`
- `MAX_int16_T`
- `MAX_int32_T`
- `MAX_uint8_T`
- `MAX_uint16_T`
- `MAX_uint32_T`
- `MIN_int8_T`
- `MIN_int16_T`
- `MIN_int32_T`

You can also use command-line parameters to:

- Replace the default `true` and `false` Boolean identifiers.
- Import a header file with the Boolean and data type limit identifier definitions.

For more information, see [Specify Boolean and Data Type Limit Identifiers](#).

Names of built-in storage classes reserved

You can no longer define custom storage classes with the same name as the built-in storage classes `Auto`, `SimulinkGlobal`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. The Custom Storage Class Designer now fails validation of custom storage classes that have these names.

Compatibility Considerations

If you previously defined custom storage classes with the same name as the built-in storage classes, MATLAB returns an error when you try to create data objects that use any of the custom storage classes defined in the affected package. If you try to load such data objects from a MAT-file, the objects do not load successfully.

To resolve these compatibility issues:

- 1 Rename the affected custom storage classes.
- 2 Update your MATLAB code to use the new names.
- 3 Recover affected data objects from existing MAT-files.

To recover affected data objects from existing MAT-files:

- 1 Start a prior release of MATLAB that uses the affected custom storage classes.
- 2 Load the MAT-files.

- 3** Use the function `matlab.io.saveVariablesToScript` to generate a MATLAB script that defines the affected data objects.
- 4** Manually update the generated script with the new names of your custom storage classes.
- 5** In release R2015a or later of MATLAB, rename the affected custom storage classes.
- 6** Run the generated script in release R2015a or later of MATLAB.

Code Generation

Simplified Code Replacement Library specification plus more replacements involving integer operations

Simplified Code Replacement Library specification

R2015a introduces a simpler approach to defining code replacement table entries programmatically. This approach significantly reduces the amount of code that you write. Consider using this approach if both of the following conditions apply:

- The workflow that you use for defining mappings involves copying, pasting, and editing existing mappings.
- You prefer not to use the Code Replacement Tool to create an initial mapping definition.

To use the approach, specify conceptual and implementation information for a table entry as detailed string specifications in a call to the function `createCRLEntry`.

This approach for defining mappings for code replacement table entries does not support:

- C++ implementations
- Data alignment
- Operator replacement with net slope arguments
- Entry parameter specifications (for example, priority, algorithm, building information)
- Semaphore and mutex function replacements

For more information, see `createCRLEntry` and Define Code Replacement Mappings.

More replacements involving integer operations

As of R2015a, code replacement opportunities have been improved for the following binary-point scaling operations. To increase match opportunities, the code generator applies equivalent scaling to inputs before performing the stored integer operation. However, input scaling occurs only if a match exists and the code generator is able to apply the replacement for the stored integer operation.

Operator	Key	Scalar, Vector, Matrix Support	Real, Complex Support
Addition (+)	RTW_OP_ADD	Scalar Vector Matrix	Real Complex
Subtraction (-)	RTW_OP_MINUS	Scalar Vector Matrix	Real Complex
Multiplication (*)	RTW_OP_MUL	Scalar	Real
Division (/)	RTW_OP_DIV	Scalar	Real
Element-wise matrix multiplication (.*)	RTW_OP_ELEM_MUL	Vector Matrix	Real

Improved readability for shared header file 'rtwtypes.h'

To improve code readability and reduce code review cost, in the `rtwtypes.h` file, the software does not generate the following definitions:

- The preprocessor directive `#define __TMWTYPES__`. The removal of this preprocessor directive prevents the inclusion of `tmwtypes.h`, making `rtwtypes.h` the single source of type definitions.
- Definitions for zero-crossing detection in triggered subsystems. For example:

```
#ifndef __ZERO_CROSSING_TYPES_H__
#define __ZERO_CROSSING_TYPES_H__

/* Trigger directions: falling, either, and rising */
typedef enum {
    FALLING_ZERO_CROSSING = -1,
    ANY_ZERO_CROSSING = 0,
    RISING_ZERO_CROSSING = 1
} ZCDirection;

/* Previous state of a trigger signal */
...
#endif
```

Models containing triggered subsystems require zero-crossing definitions when the trigger is rising, falling, or either. In R2015a, the software generates these definitions in a separate file called `zero_crossing_types.h`. The software creates the file only if the model requires the file.

Compatibility Considerations

Because of the removal of the `#define __TMWTYPES__` directive, the `rtwtypes.h` file generated using R2015a might not be compatible with code that you generate using a previous release. For example, in some circumstances, the generated code from an older release might include `tmwtypes.h` after `rtwtypes.h`. This code does not compile without the `#define __TMWTYPES__` directive.

If your build process uses custom code that includes the header file `tmwtypes.h` instead of `rtwtypes.h`, you might observe a compiler error that indicates a redefined type.

To avoid this error, in the custom code, replace:

```
#include "tmwtypes.h"
```

with:

```
#include "rtwtypes.h"
```

If you use the `mex` command to compile custom code for an S-function, include `tmwtypes.h` for the `mex` compilation and `rtwtypes.h` for the code generation compilation:

```
#ifdef MATLAB_MEX_FILE
#include "tmwtypes.h"
#else
#include "rtwtypes.h"
#endif
```

Alternatively, before generating code for your model, configure the model for backward compatibility by setting the parameter `InferredTypesCompatibility` to `on`.

```
set_param(model, 'InferredTypesCompatibility', 'on')
```

When you enable backward compatibility, the code generator creates the preprocessor directive `#define __TMWTYPES__` inside `model.h`.

New and enhanced Model Advisor checks for MISRA-C compliance

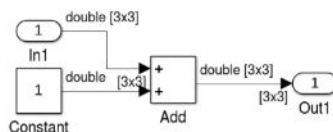
To improve MISRA-C compliance, you can run the following Model Advisor checks:

Check	New or Enhanced	Description	Addresses MISRA-C Rule Numbers
Check for bitwise operations on signed integers	New	Identifies blocks that contain bitwise operations on signed integers.	12.7
Check configuration parameters for MISRA-C:2004 compliance	Enhanced	Now verifies that you cleared Code Generation > Code Style > Replace multiplications by powers of two with signed bitwise shifts	12.7
Check for blocks not recommended for MISRA-C:2004 compliance	Enhanced	Now identifies Lookup Table blocks using cubic spline interpolation or extrapolation methods.	11.4 and 11.5

Improved traceability for AUTOSAR RTE implicit read

AUTOSAR code generation now generates more traceable and readable code for a root inport that models an AUTOSAR RTE implicit read, especially when the inport data type is a matrix.

For example, consider root inport `In1` the following model:



In R2014b, the generated code introduces a hidden Signal Conversion block:

```
void Runnable_Step(void)
{
  const real_T *rtb_TmpSignalConversionAtIn1Out;
  real_T tmp[9];
  int32_T
  /* SignalConversion: '<Root>/TmpSignal ConversionAtIn1Outputport1' incorporate
  Inport: '<Root>/In1 */
  rtb_TmpSignalConversionAtIn1Out = Rte_IRead_Runnable_Step_Input_Element0();
  /* Sum: '<Root>/Add' incorporates:
   * Constant: '<Root>/Constant'
   */
  for (i = 0; i < 9; i++) {
    tmp[i] = rtb_TmpSignalConversionAtIn1Out [i] + 1.0;
  }
}
```

```

    ...
    Rte_IWrite_Runnable_Step_Output_Output(tmp);
}

```

In R2015a, the generated code is traceable and more readable. A hyperlink is generated for <Root>/In1.

```

void Runnable_Step(void)
{
const real_T *tmp_In1;
real_T tmp[9];
int32_T i;
/* Inport: '<Root>/In1' */
tmp_In1 = Rte_IRead_Runnable_Step_Input_Element0();
/* Sum: '<Root>/Add' incorporates:
 * Constant: '<Root>/Constant'
 */
for (i = 0; i < 9; i++) {
    tmp[i] = rtb_tmp_In1[i] + 1.0;
}
    ...
    Rte_IWrite_Runnable_Step_Output_Output(tmp);
}

```

Configurable aliveTimeout value for AUTOSAR ports

In AUTOSAR applications, the `aliveTimeout` value for an AUTOSAR port specifies the amount of time in seconds after which the AUTOSAR software component must be notified if the port has not received data according to a specified timing description. In previous releases, an `arxml` export generated a fixed `aliveTimeout` value of 60 for each AUTOSAR port, without providing a way to modify the `aliveTimeout` value in Simulink.

The software now allows you to configure an `aliveTimeout` value that subsequent `arxml` exports generate for each AUTOSAR port. For more information, see [Configure AUTOSAR Port aliveTimeout Value](#).

AUTOSAR calibration parameter export for COM_AXIS lookup tables

For shared axis (COM_AXIS) lookup tables, AUTOSAR code generation now exports an `arxml` that supports run-time calibration of lookup table parameters. To configure a lookup table for run-time calibration, add an n-D Lookup Table block to your model and configure it for COM_AXIS data. For table data and axis data that you want to tune or manipulate at run-time, reference AUTOSAR calibration parameters. For more information, see [Calibration Parameters for COM_AXIS Lookup Tables](#).

Fixed-point scaling information in Code Interface Report

Fixed-point scaling information is added to the code generation report in the **Code Interface Report** section. Better accessibility to this information makes it easier for you to integrate your code with generated code containing fixed-point data types. Each fixed-point entry in a report table has a value in the new **Scaling** column giving its data type and fraction length using Simulink fixed-point data type notation. Here is an example of fixed-point data representations in the **Outputs** table.

Outports

Block Name	Code Identifier	Data Type	Scaling	Dimension
<Root>/Out1	<i>Defined externally</i>	uint32_T	ufix32_En14	1
<Root>/Out2	<i>Defined externally</i>	int32_T	sfix32_En12	1

You must have a Fixed-Point Designer™ license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see Fixed-Point Data Type and Scaling Notation.

Unsigned integer minimum data limit identifiers

The following unsigned integer minimum data limit identifiers are no longer defined in `rtwtypes.h`:

- `MIN_uint8_T`
- `MIN_uint16_T`
- `MIN_uint32_T`
- `MIN_uint64_T`

Previously, the unsigned integer minimum data limit identifiers defined in `rtwtypes.h` were potentially not used in the generated code:

- Standard C header files do not provide an unsigned integer minimum data limit constant.
- In most instances, the code generator did not replace `0` with the unsigned integer minimum limit identifier.

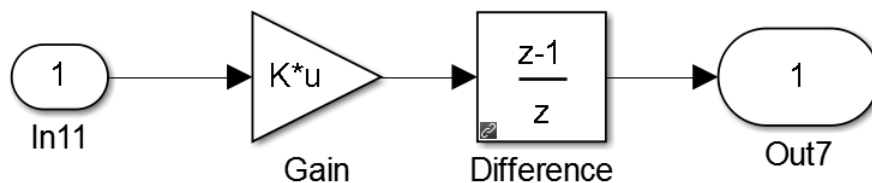
Compatibility Considerations

If you previously used unsigned integer minimum data limit identifiers in custom code, for example in an S-Function, replace the limit with `0`.

Default iteration variable data type

The default data type for iteration variables in the generated code is a 32-bit integer. Previously, the default data type was `int` with an unspecified bit size.

For example, consider the following model.



The code generator produced this code in R2014b:

```
{
  int_T i;
  for (i = 0; i < 16; i++) {
    test1_2a_Y.Out7[i].re = (0L);
    test1_2a_Y.Out7[i].im = (0L);
  }
}
```

The code generator produces this code in R2015a:

```
{
  int32_T i;
  for (i = 0; i < 16; i++) {
    test1_2a_Y.Out7[i].re = (0L);
    test1_2a_Y.Out7[i].im = (0L);
  }
}
```

Deployment

Code Replacement Viewer enhanced

- MATLAB command for invoking the Code Replacement Viewer is renamed from `RTW.viewTf1` to `crviewer`.
- The trace information for misses that occur during the match process is reformatted as a table.

For more information, see [Verify Code Replacements](#).

Model configuration parameter considered for division operator code replacements

When determining match criteria for division operator code replacement entries, the code generator uses model configuration parameter **Signed integer division rounds to** (`ProdIntDivRoundTo`) to determine equivalent rounding modes. For example, assume that **Signed integer division rounds to** is set to `Floor`. The code generator matches model division operations with integer rounding modes set to `simplest` or `floor` to division operator code replacement entries with the **Rounding mode** (`RoundingModes`) parameter set to `Simplest` or `Floor`.

Lookup table algorithm parameter specification enhancements

R2015a introduces enhancements for setting algorithm parameters for lookup table function code replacement table entries.

- From the Code Replacement Tool, you can specify multiple values for an algorithm parameter.
- Programming interface improvements include:
 - Algorithm parameter set objects for discovering and managing algorithm parameter settings.
 - For a given lookup table function, default settings for unchanged algorithm parameters.
 - Validation of syntax, parameter names, and values in parameter assignment statements.
 - `getAlgorithmParameters` function for examining the algorithm parameter settings for a lookup table function code replacement table entry.
 - `setAlgorithmParameters` function for setting the algorithm parameters for a lookup table function code replacement table entry.

For more information, see [getAlgorithmParameters](#), [setAlgorithmParameters](#), and [Lookup Table Function Code Replacement](#).

Header file for Basic Linear Algebra Subroutine (BLAS) multiplication function code replacement example changed

The header file for the Basic Linear Algebra Subroutine (BLAS) multiplication function code replacement example changed from `blascompat32.h` to `blascompat32_crl.h`. The associated include path for this header file changed to `matlab/toolbox/rtw/rtwdemos/crl_demo`. For more information, see “Improved readability for shared header file ‘rtwtypes.h’” on page 17-11.

Code replacement detection of overflow and rounding mode equivalence

As of R2015a, the code replacement software detects overflow and rounding mode equivalence for real scalar multiplication and division operations. When an operation does not overflow, based on input and output data types, a match occurs for code replacement table entries with the saturation mode set to **Wrap on Overflow** (RTW_WRAP_ON_OVERFLOW). Similarly, if the code replacement software detects equivalent rounding modes, a match occurs.

Feature being removed in a future release

The Filter Design and Analysis Tool option to target the Code Composer Studio™ IDE will be removed in a future release. The Filter Design and Analysis Tool is available with Signal Processing Toolbox™.

Performance

More efficient code involving model references, unit delays, and global data references

Reusable custom storage class for Model block input/output ports

Previously, if a pair of root-level model input and output signals used the same `Reusable` storage class specification, the code generator could reuse the root I/O signals in the generated code. In R2015a, this optimization extends to Model block I/O signals. The code generator tries to reuse buffers for a pair of Model block I/O signals with the same `Reusable` storage class specification. This reuse can eliminate buffers in the generated code.

The input/output signals must have the same data types and sampling rates. This optimization does not apply to conditional output ports.

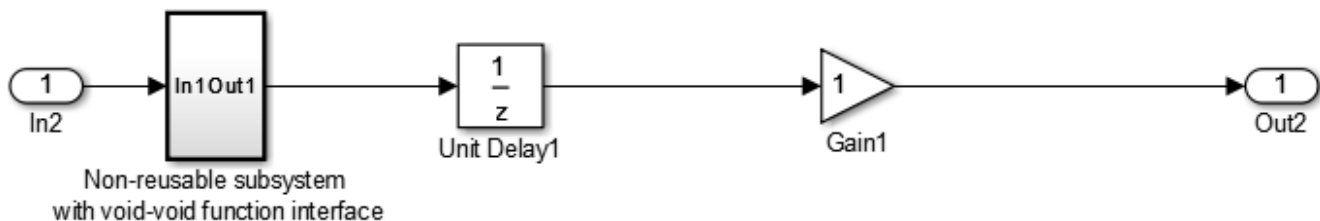
For more information on how to configure your model to take advantage of this optimization, see [Buffer Reuse for Model Block Boundary and Unit Delay](#).

Reuse input, output, and state of Unit Delay block

If any of the following conditions exist, the code generator tries to reuse the input, output, and state of a Unit Delay block:

- In the Configuration Parameters dialog box, on the **Optimizations > Signals and Parameters** pane, you select `Use global` to hold temporary results from the **Optimize global data access** list.
- You use the same `Reusable` custom storage class specification for a pair of input and state arguments or a pair of output and state arguments of a Unit Delay block.
- You use a `Reusable` custom storage class specification for a state argument of a Unit Delay block.

The reusable input, output, and state arguments must have the same data types and sampling rates. This optimization can reduce the number of global variables. For example, consider the following model.



In R2014b, the code generator produces the following code:

```
DW_reuse_ex_T reuse_ex_DW;
void reuse_ex_step(void)
{
    reuse_ex_Y.Out2 = reuse_ex_P.Gain1_Gain * reuse_ex_DW.UnitDelay1_DSTATE;
    reuse_ex_Subsystem();
    reuse_ex_DW.UnitDelay1_DSTATE = reuse_ex_B.Gain2;
}
```

In R2015a, the code generator produces the following code:

```
void reuse_ex_step(void)
{
    reuse_ex_Y.Out2 = reuse_ex_P.Gain1_Gain * reuse_ex_B.Gain2;
    reuse_ex_Subsystem();
}
```

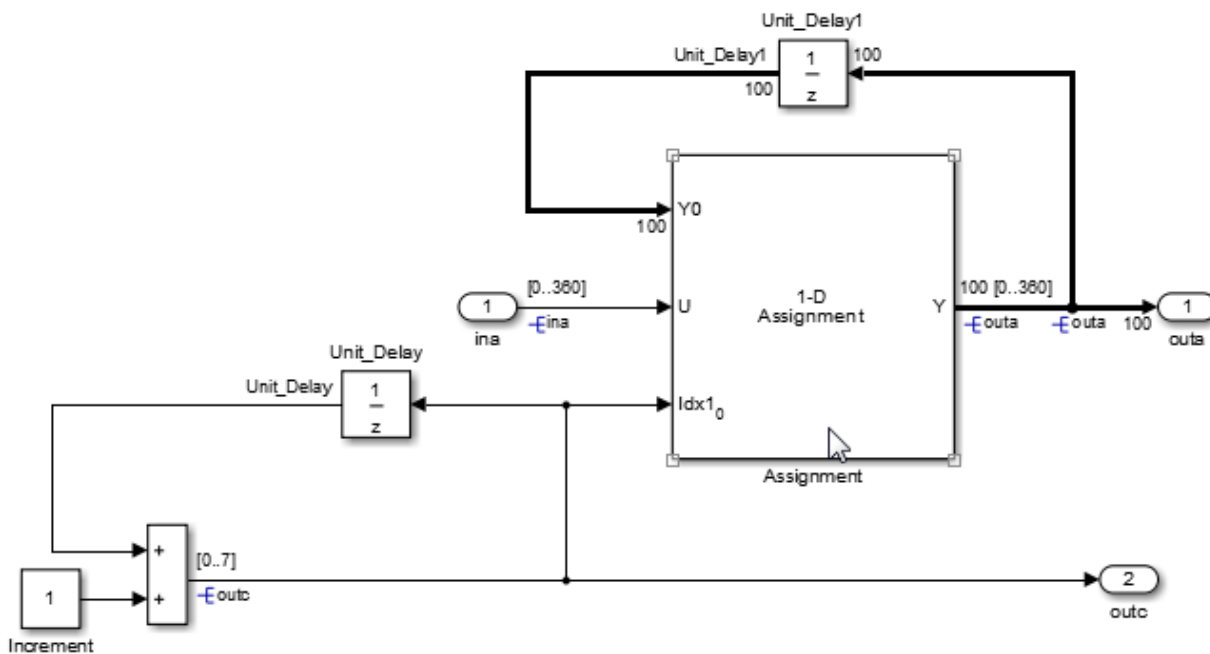
For more information on how to configure your model to use this optimization, see Buffer Reuse for Model Block Boundary and Unit Delay.

Enhanced variable reuse optimizations

The code generator has improved analysis of data copies to provide more variables for reuse and more consistent variable reuse behavior. These enhancements result in:

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

For example, consider the following model.



The code generator produced this code in R2014b:

```
int32_T i;

/* Sum: '<Root>/Sum' incorporates:
 * Constant: '<Root>/Increment'
 * UnitDelay: '<Root>/Unit_Delay'
```

```
*/
outc = (uint8_T)(outc + 1);

/* Assignment: '<Root>/Assignment' incorporates:
 * Inport: '<Root>/ina'
 * UnitDelay: '<Root>/Unit_Delay1'
 */
for (i = 0; i < 100; i++) {
    outa[i] = mg909420_DWork.Unit_Delay1_DSTATE[i];
}

outa[outc] = ina;

/* End of Assignment: '<Root>/Assignment' */

/* Update for UnitDelay: '<Root>/Unit_Delay1' */
for (i = 0; i < 100; i++) {
    mg909420_DWork.Unit_Delay1_DSTATE[i] = outa[i];
}
```

The code generator produces this code in R2015a:

```
outc = (uint8_T)(outc + 1);

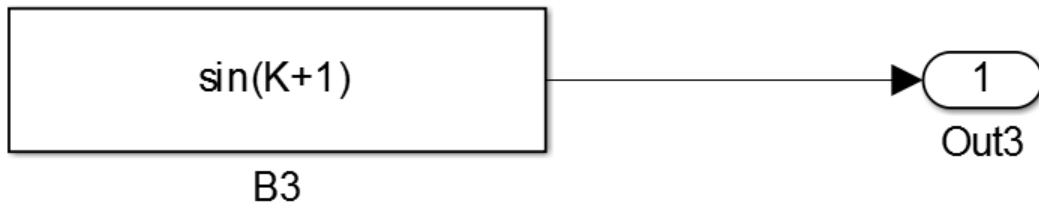
/* Assignment: '<Root>/Assignment' incorporates:
 * Inport: '<Root>/ina'
 */
outa[outc] = ina;
```

Strategic caching of global variable references

The code generator replaces global variables used for temporary storage with local variables. This replacement enables expression folding and other optimizations available for local variables, resulting in:

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

For example, in the following model, the signal from a Constant block feeds into an Outport block. On the **Optimization > Signals and Parameters** pane, the **Optimize global data access** parameter is set to Minimize global data access.



The code generator produced this code in R2014b:

```
/* Outport: '<Root>/Out3' incorporates:
 * Constant: '<Root>/B3'
 */
mg1003222_Y.Out3 = K + 1.0;
mg1003222_Y.Out3 = sin(mg1003222_Y.Out3);
```

The code generator produces this code in R2015a:

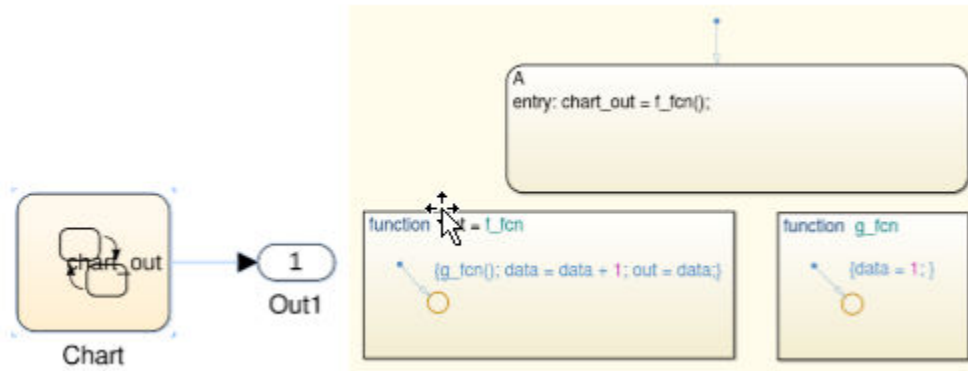
```
/* Outport: '<Root>/Out3' incorporates:
 * Constant: '<Root>/B3'
 */
mg1003222_Y.Out3 = sin(K + 1.0);
```

Enhanced global variable localization optimizations

The code generator has more information to determine which global variables it can replace with local variables. It can also update function interfaces to pass these local variables. With these enhancements, the code generator can:

- Enable more optimizations for local variables.
- Potentially reduce the number and use of global variables.

For example, consider the following Stateflow chart.



The code generator produced this code in R2014b:

```
/* Function for Chart: '<Root>/Chart' */
static real_T test_f_fcn(void)
{
    /* MATLAB Function 'f_fcn': '<S1>:5' */
    /* Graphical Function 'f_fcn': '<S1>:5' */
    /* '<S1>:10:1' */
    test_g_fcn();
    /* '<S1>:10:1' */
    test_DW.data++;
    /* '<S1>:10:1' */
    return test_DW.data;
}

```

...

```
/* Function for Chart: '<Root>/Chart' */
static void test_g_fcn(void)
{
    /* MATLAB Function 'g_fcn': '<S1>:13' */
    /* Graphical Function 'g_fcn': '<S1>:13' */
    /* '<S1>:12:1' */
    test_DW.data = 1.0;
}

```

The code generator produces this code in R2015a:

```
/* Function for Chart: '<Root>/Chart' */
static real_T test_f_fcn(void)
{
    real_T out;
    real_T data;
    /* MATLAB Function 'f_fcn': '<S1>:5' */
    /* Graphical Function 'f_fcn': '<S1>:5' */
    /* '<S1>:10:1' */
    test_g_fcn(&data);
    /* '<S1>:10:1' */
    out = data + 1.0;
    /* '<S1>:10:1' */
    return out;
}

```

```

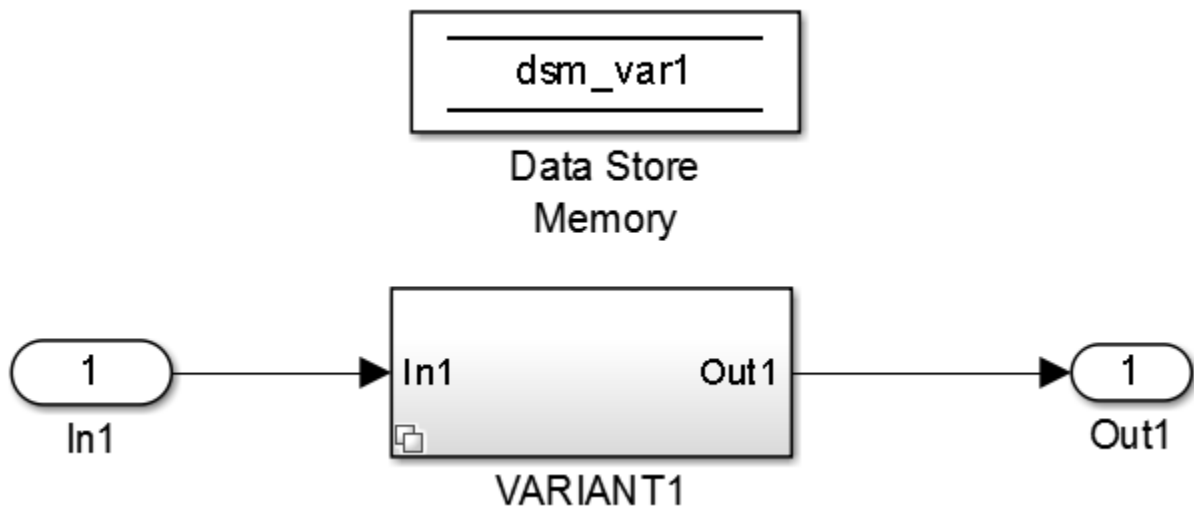
...
/* Function for Chart: '<Root>/Chart' */
static void test_g_fcn(real_T *data)
{
  /* MATLAB Function 'g_fcn': '<S1>:13' */
  /* Graphical Function 'g_fcn': '<S1>:13' */
  /* '<S1>:12:1' */
  *data = 1.0;
}

```

Conditional compilation of Data Store Memory block memory definition and declaration

When a Data Store Memory block has a non-auto storage class and variant subsystems reference the block, the code conditionally compiles the definition and declaration of the block memory. To compile, the code uses the preprocessor conditions associated with the variant subsystems. Previously, the code did not conditionally compile the definition and declaration of the block memory, resulting in the declaration and definition of global variables that the code potentially did not use.

For example, consider the following model.



In R2014b, the code generator produces this code:

```

volatile real_T dsm_var1;
void dsm_variants_ex_initialize(void)
{
  /* custom states */
  dsm_var1 = 0.0;
}

```

In R2015a, the code generator produces code using preprocessor conditionals:

```

#if VARI1 == USE
  volatile real_T dsm_var1;
#endif /* VARI1 == USE */

```

```
void dsm_variants_ex_initialize(void)
{
    /* custom states */
    #if VARI1 == USE
        dsm_var1 = 0.0;
    #endif /* VARI1 == USE */
}
```

Ternary Boolean expressions transformed into assignment statements

In R2015a, the code generator removes the conditional part of a ternary Boolean expression, leaving an assignment statement. An assignment statement in place of a ternary Boolean expression improves execution speed and reduces RAM/ROM.

Observe the following lines of code generated in R2014b:

```
uint32_T a;
uint32_T b;
a = (a<b)?1U:0U;
```

Compare the same lines of code generated in R2015a:

```
uint32_T a;
uint32_T b;
a = uint32_T(a<b);
```


Verification

SIL/PIL for protected models and SIL source code debugging using Microsoft Visual Studio Express

- “SIL/PIL for protected models” on page 17-25
- “SIL source code debugging using Microsoft Visual Studio Express” on page 17-25

SIL/PIL for protected models

To verify the behavior of code generated from protected models, use Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations.

This feature supports:

- Generated code with standalone (Top model) and model reference (Model reference) code interfaces.
- AUTOSAR models, including packaged ARXML files.
- Execution-time profiling of task entry-point functions.

The screenshot shows a dialog box titled "Create Protected Model". It contains the following elements:

- Description:** Create a protected model(.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.
- Allow user of protected model to:**
 - Open read-only view of model
 - Simulate
 - Use generated code
- Code interface:** A dropdown menu with "Model reference" selected.
- Content type:** A dropdown menu with "Top model" selected.

For more information, see:

- Create a Protected Model
- Simulink.ModelReference.protect
- Referenced Model Simulation Using SIL or PIL

SIL source code debugging using Microsoft Visual Studio Express

Embedded Coder supports Microsoft Visual Studio® Express 2013 for Windows Desktop for debugging code during software-in-the-loop (SIL) simulations. To specify Microsoft Visual Studio Express for SIL debugging:

- In MATLAB, select the Microsoft Windows SDK 7.1 compiler.
- On the **Configuration Parameters > Code Generation > Verification** pane, select the **Enable source-level debugging for SIL simulations** check box.

For more information, see Debug Code During SIL Simulations.

Model block SIL/PIL parameter renamed

The following SIL/PIL changes apply to the Model block:

- The command-line parameter `CodeUnderTest` is renamed `CodeInterface`.
- In the Function Block Parameters dialog box, the field **Code under test** is renamed **Code interface**.

ERT S-Function block no longer supported for AUTOSAR

As of R2015a, to verify code generated from AUTOSAR software component, use the SIL block.

For more information, see [Verify AUTOSAR C Code with SIL and PIL](#).

Compatibility Considerations

R2014a introduced the ability to switch between two SIL block behaviors—legacy (ERT S-function) and unified (SIL block). The software also indicated that ERT S-function support for code verification would be removed in a future release. Starting in R2015a, for AUTOSAR code generation, use the SIL block.

SIL/PIL support for replacing boolean data type with int8

You can replace the `boolean` built-in data type with an integer type in generated code. Before R2015a, SIL and PIL execution supported data type replacement of `boolean` with `uint8`. As of R2015a, SIL and PIL execution supports replacement of `boolean` with `uint8` or `int8`.

For more information, see [Replace boolean with Specific Integer Data Type and Data Type Replacement](#).

SIL/PIL support for generated access methods for C++ model class root-level I/O signals

In the Configuration Parameters dialog box, on the **Code Generation > Interface** pane, the **External I/O access** parameter (`GenerateExternalIOAccessMethods`) specifies whether to generate access methods for root-level I/O signals for a C++ model class. Before R2015a, SIL and PIL simulations required that you set this parameter to `None`. As of R2015a, you can run SIL and PIL simulations for code that you generate with the parameter set to `Method` or `Inlined` method. These settings cause the code generator to produce noninlined or inlined access methods for the root-level I/O signals for the class.

For more information, see [External I/O access and Configure Step Method for Model Class](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.7

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Processor-in-the-loop (PIL) verification and execution profiling for MATLAB code

Use processor-in-the-loop (PIL) execution to verify code that you intend to deploy in production. PIL execution involves cross-compiling and running library object code on your target processor through a MATLAB PIL interface. You can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of library code.

Before running PIL executions on your target hardware, specify a connectivity configuration for your target. See [PIL Customization for Target Environment and Create PIL Target Connectivity Configuration](#).

You can run a PIL execution:

- Using the MATLAB Coder Project Interface. See [Processor-in-the-Loop Execution Through Project Interface](#).
- At the command line. See [Processor-in-the-Loop Execution From Command Line](#).

Through software-in-the-loop (SIL) and processor-in-the-loop (PIL) execution, you can produce execution time profiles of code generated from entry-point functions. Use these profiles to determine:

- Whether the generated code meets real-time requirements of your target hardware.
- Which entry-point functions require performance improvement.

For more information, see [Execution Time Profiling](#).

Software-in-the-loop verification improvements for MATLAB Coder

The following table lists software-in-the-loop (SIL) execution improvements.

Feature		R2014b support	Previous support
Code debugging during SIL execution		Linux: GNU® Data Display Debugger (DDD) Windows: Microsoft Visual Studio debugger	Windows: Microsoft Visual Studio debugger
Interface types	Multiple entry points	Yes	No
Size	Static variable-size arrays	Yes	Limited to function arguments that were fixed-size structures with variable-size fields.

For more information, see:

- [Code Debugging During SIL Execution](#)
- [SIL/PIL Execution Support and Limitations](#)

Additional options for custom banners and comments in C and C++ code generated from MATLAB code

In a code generation template (CGT) file, you can now specify the following:

- Custom banners for shared utility functions
- Custom comments before individual code sections such as `Include Files` and `Function Declarations`
- doxygen style comments

The `style` attribute options for doxygen style comments are `doxygen` and `doxygen_qt`. The `TargetLang` and `CommentStyle` code configuration object properties determine the use of C or C++ style comments with the doxygen style comments.

doxygen with C style comments

```
/**
 * multiple line comments
 * second line
 */
```

doxygen with C++ style comments

```
///
/// multiple line comments
/// second line
///
```

doxygen_qt with C style comments

```
/*!
 * multiple line comments
 * second line
 */
```

doxygen_qt with C++ style comments

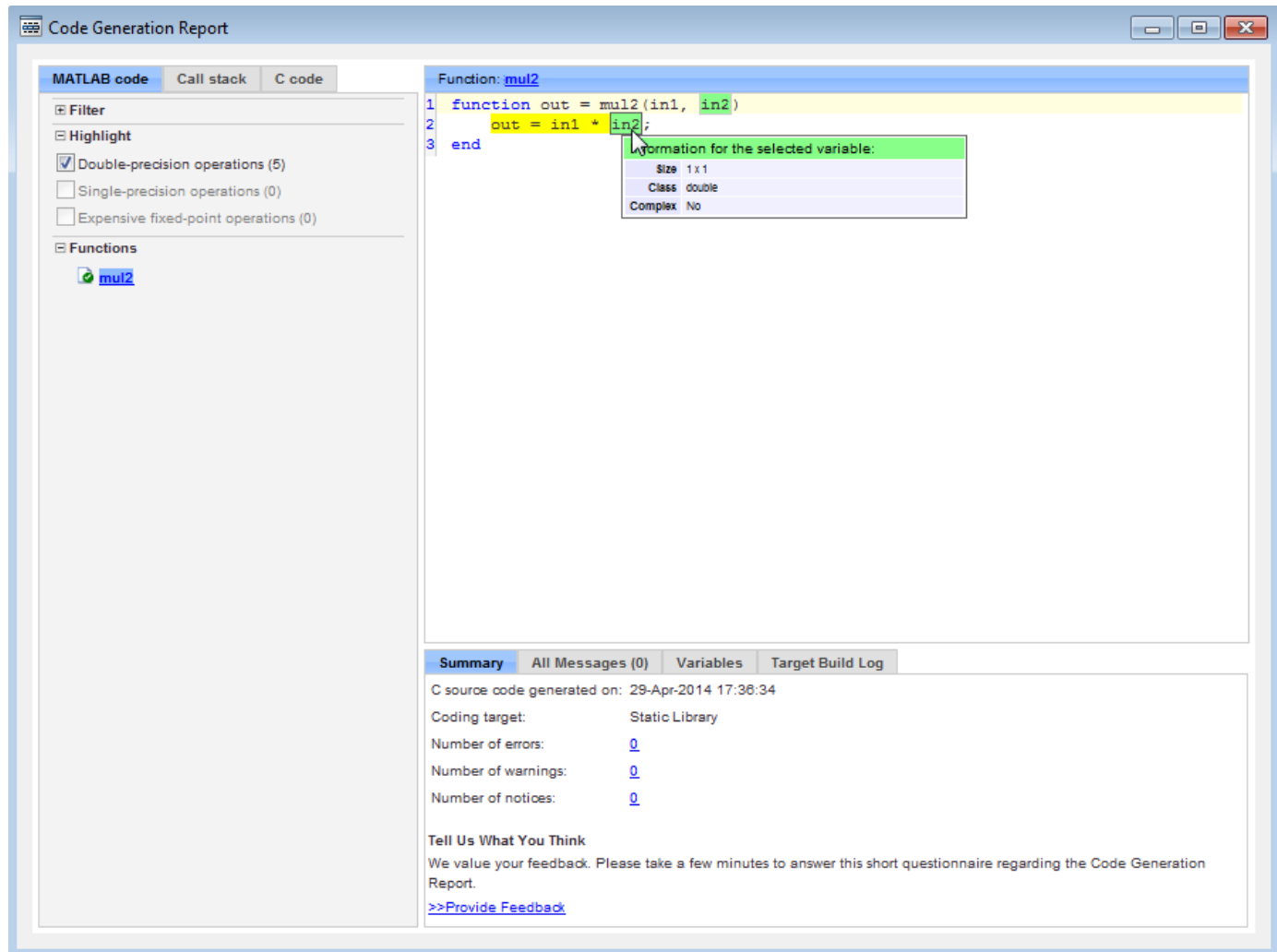
```
/*!
/*! multiple line comments
/*! second line
/*!
```

See Code Generation Template (CGT) Files for MATLAB.

Highlighting of potential data type issues in code generation reports

When you generate standalone code from MATLAB code, you now have the option to highlight potential data type issues in the code generation report. The report highlights MATLAB code that results in single-precision and double-precision operations in the generated C/C++ code. If you have a Fixed-Point Designer license, the report also highlights expressions in the MATLAB code that result in expensive fixed-point operations in the generated code. The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that result in cumbersome multiplication and division, and expensive rounding in generated C/C++ code.

The following example report highlights MATLAB code that results in double-precision operations in the generated C code.



The checks are disabled by default.

To enable the checks in a project, on the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data types issues** check boxes.

To enable the checks at the command line:

- 1 Create a configuration object to generate standalone C/C++ code for an embedded target. For example:

```
cfg = coder.config('lib', 'ecoder', true);
```

- 2 Set the HighlightPotentialDataTypeIssues property to true:

```
cfg.HighlightPotentialDataTypeIssues = true;
```

See [Highlight Potential Data Type Issues in a Report](#) and [Find Potential Data Type Issues in Generated Code](#).

If you have a Fixed-Point Designer license, you have the option to highlight potential data type issues in the generated HTML report that is available after the fixed-point type validation step of the fixed-point conversion process. An Embedded Coder license is not required to highlight potential data types issues in this report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations.

The following example report highlights MATLAB code that requires expensive fixed-point operations.

The screenshot shows the 'Code Generation Report' window. On the left, there is a 'MATLAB code' tab and a 'Call stack' tab. Under 'MATLAB code', there is a 'Filter' section with 'Highlight' options: 'Double-precision operations (6)', 'Single-precision operations (0)', and 'Expensive fixed-point operations (1)' (checked). Below that, there are 'Functions' listed: 'mul2_fixpt' and 'mul2_wrapper_fixpt'. The main area shows the MATLAB code for the function 'mul2_fixpt'. The code is as follows:

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 %       Generated by MATLAB 8.4 and Fixed-Point Designer 4.3
4 %
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 %#codegen
7 function out = mul2_fixpt(in1,in2)
8
9 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Saturate', 'ProductMo
10 out = fi(in1*in2, 0, 15, 0, fm);
11 end

```

The line 'out = fi(in1*in2, 0, 15, 0, fm);' is highlighted in yellow. A tooltip is displayed over the expression 'in1*in2', showing the following information:

Information for the selected expression:	
Size	1 x 1
Class	embedded.fi
Complex	No
Signedness	Unsigned
VVL	128
FL	6

At the bottom of the window, there is a 'Summary' section with the following information:

- C source code generated on: 30-Apr-2014 10:34:13
- Coding target: C MEX Function
- Number of errors: 0
- Number of warnings: 0
- Number of notices: 0

There is also a 'Tell Us What You Think' section with a link to 'Provide Feedback'.

The checks are disabled by default. To enable the checks in a project:

- 1 In the Fixed-Point Conversion Tool, click **Advanced** to view the advanced settings.
- 2 Set **Highlight potential data type issues** to Yes.

To enable the checks at the command line:

- 1 Create a floating-point to fixed-point conversion configuration object:

```
fxptcfg = coder.config('fixpt');
```
- 2 Set the HighlightPotentialDataTypeIssues property to true.

```
fxptcfg.HighlightPotentialDataTypeIssues = true;
```

See Data Type Issues in Generated Code.

Model Architecture and Design

AUTOSAR targeting updates including 4.1 ARXML, client/server with Simulink Functions, multi-instance components, and IFL/IFX libraries

R2014b provides many enhancements to AUTOSAR code generation and Simulink modeling of AUTOSAR elements. Highlights include:

- Support for AUTOSAR Release 4.1, including:
 - AUTOSAR 4.1 (schema version 4.1.1) arxml and C code generation
 - AUTOSAR 4.1 initialization events
 - AUTOSAR 4.1 provide-require ports
- Ability to model AUTOSAR clients and servers in Simulink, using Simulink Function and Function Caller blocks.
- Ability to model multi-instance AUTOSAR software components (SWCs) in Simulink, using the `Reusable function` setting of the model parameter **Code interface packaging**.
- AUTOSAR code replacement library support for:
 - Floating-point interpolation (IFL) and fixed-point interpolation (IFX) library routines.
 - Functions that perform a multiplication, and then a division operation in sequence.
 - Addition and subtraction operator replacements for cast-after-operation algorithms. (For more information, see “Algorithm specification for addition and subtraction operator replacement” on page 18-21.)

For more information about AUTOSAR-related enhancements in R2014b, see:

- “Support for AUTOSAR Release 4.1” on page 18-11
- “AUTOSAR client and server modeling” on page 18-7
- “Multi-instance AUTOSAR atomic software components” on page 18-12
- Code Replacement for AUTOSAR
- “Support Package for AUTOSAR Standard” on page 18-10
- “AUTOSAR help navigation enhancements” on page 18-11

AUTOSAR client and server modeling

Beginning in R2014b, you can model AUTOSAR clients and servers in Simulink for simulation and code generation.

- Use Simulink Function blocks at the root level of a model to model AUTOSAR servers.
- Use Function Caller blocks to model AUTOSAR client invocations.
- Use the top-model export-functions modeling style to create interconnected Simulink functions, function-calls, and root model inports and outports.

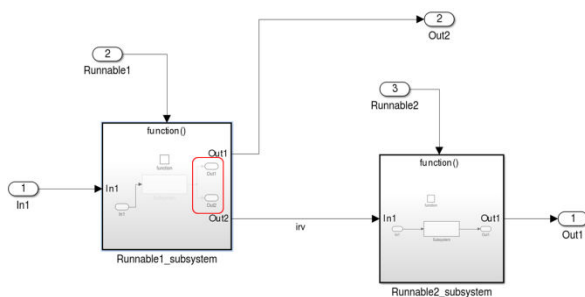
For more information, see Client-Server Interface and Configure AUTOSAR Client-Server Communication.

Global From and Goto blocks for AUTOSAR modeling

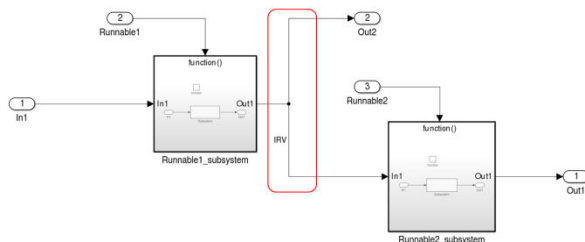
Beginning in R2014b, you can use global From and Goto blocks in a model configured for AUTOSAR. With From and Goto blocks, you can pass a signal from one block to another without actually connecting them. You can model AUTOSAR runnables with more flexibility and cleaner separation of components and interfaces.

AUTOSAR IRV branch from output signal allowed outside runnable

In previous releases, if you wanted to branch an AUTOSAR runnable output signal to an AUTOSAR inter-runnable variable (IRV) and a Simulink model root output, AUTOSAR code generation supported only branching inside the runnable.



Beginning in R2014b, AUTOSAR code generation supports branching outside the runnable. This modeling pattern can potentially generate more efficient C code, for example, with fewer global variables and fewer block I/O buffers.



The following guidelines and constraints apply to the new modeling pattern:

- You can branch a runnable output signal to only one root output outside a runnable boundary.
- When a runnable output signal branches to an IRV and a root output outside the runnable subsystem:
 - Only Goto and From blocks are allowed between the source and the destination of the signal.
 - You cannot conditionally write to the IRV or root output.
- When a runnable output signal does not branch, only Goto/From and Merge blocks are allowed between the source and the destination of the signal.

Data, Function, and File Definition

Constant sample time limitation for AUTOSAR models

Previously, for models using the AUTOSAR target, the compiler reported a warning if you configured a root-level Outputport block to inherit a constant sample time from its sources. The compiler then set the sample time of the root-level Outputport block to the fundamental rate of the model. In R2014b, this warning becomes an error.

Iteration variable in For Iterator block uses signal name

The code generator allows use of the signal name as part of the iteration variable name in the For Iterator block. Using the signal name increases the traceability of the generated code.

You can control the name of the iteration variable. Specify the setting for **Local temporary variables** on the **Code Generation > Symbols** pane. The signal name is the \$N part of the variable name.

Previously, the code generator used a default name, incorporating the name of the system hierarchy for the iteration variable.

See also For Iterator and Local temporary variables.

Data type replacement specification can be used across models

When you specify data type replacement names for a model, the code generator can use the replacement types to generate shared functions and constants. You save RAM/ROM space and the code generator can use the user-defined types consistently.

For more information, see Data Type Replacement.

Definition file for grouped custom storage classes

When defining custom storage classes of the Struct or BitField type, you can now specify the definition file for exported grouped custom storage classes.

Type definition location for custom storage classes

Previously, the type definitions for data that used the Struct or BitField custom storage class were generated into the `model_types.h` header file. Now, those type definitions are generated into the same header file as that containing the data declarations (`model.h`, by default). If you specify a header file for such grouped custom storage classes, then both the type definitions and the data declarations are generated into that specified file.

GetFunction and SetFunction included in checks for identifier clash

Simulink now includes the `GetFunction` and `SetFunction` properties of custom storage class attributes during checks for identifier name clashes in data objects. Previously, these properties were ignored during identifier clash detection.

Code Generation

Enhanced reporting of eliminated blocks

In R2014b, the **Eliminated/Virtual Blocks** section of the traceability report includes a more accurate list of blocks eliminated by optimization. For these blocks, the code can now identify if the block was eliminated by a code generation optimization or by a block reduction. The comments for these blocks are more informative and include the following changes:

- Previously, a block eliminated from a model during code generation was reported as `Not traceable`. In R2014b, the block comment is `Eliminated by code generation optimization`.
- Previously, a block eliminated by Simulink block reduction was reported as `Not traceable`. In R2014b, the block comment is the same optimization information available in the `model.h` file when you select **Code Generation > Comments > Show eliminated blocks**.
- Previously, a block eliminated by code generation or block reduction was reported as `Not traceable` in the Model Optimization Rationale column of a generated traceability matrix. In R2014b, a block eliminated by code generation has `CodeGenerationReducedBlock` in the Model Optimization Rationale column. A block eliminated by block reduction has `SimulationReducedBlock` in this column.

For more information on traceability reports, see [Customize Traceability Reports](#).

Improved MISRA-C type cast compliance

You can choose how the code generator specifies data type casts in the generated code, including an option to choose MISRA data type cast compliance. The MISRA data type casting eliminates common MISRA standard violations, including address arithmetic and assignment. It reduces 10.1, 10.2, 10.3, and 10.4 violations.

You can also choose data type casting that is minimal or explicit.

For more information, see [Control Cast Expressions in Generated Code](#).

Support Package for AUTOSAR Standard

Beginning in R2014b, Embedded Coder software provides add-on support for the AUTOSAR standard via the Embedded Coder Support Package for AUTOSAR Standard. With the support package installed, you can create and modify an AUTOSAR configuration for a model, model AUTOSAR elements, and generate ARXML and AUTOSAR-compatible C code from a model.

To download and install the support package,

- 1 On the MATLAB Toolstrip, click **Add-Ons > Get Hardware Support Packages**.
- 2 Select **Install from Internet** and click **Next**.
- 3 From the list of available support packages, select **AUTOSAR Standard**.
- 4 To complete the installation, follow the instructions provided by Support Package Installer.

For more information, see [Support Package Installation](#).

Compatibility Considerations

AUTOSAR models and scripts that worked without a support package before R2014b now require Embedded Coder Support Package for AUTOSAR Standard. Install the support package before working with AUTOSAR models and scripts.

AUTOSAR help navigation enhancements

To make it easier to find AUTOSAR topics within MATLAB documentation, R2014b introduces the following AUTOSAR documentation enhancements:

- New AUTOSAR landing page in MATLAB Help — Encapsulates the entire Embedded Coder AUTOSAR workflow.

AUTOSAR

Develop AUTOSAR software components for automotive systems

Modeling Patterns for AUTOSAR

Modeling AUTOSAR Software Components for simulation and code generation

AUTOSAR Component Creation

Create Simulink[®] model representation of AUTOSAR software component

AUTOSAR Component Development

Develop and validate AUTOSAR software component

AUTOSAR Code Generation

Export component XML description and C code for AUTOSAR run-time environment

- New *Embedded Coder AUTOSAR* book in PDF format — Collects AUTOSAR concepts, examples, how-to topics, and reference material in a PDF file to help Simulink users learn how to model AUTOSAR components.

PDF Documentation for Embedded Coder

[Embedded Coder Getting Started Guide](#)

[Embedded Coder User's Guide](#)

[Embedded Coder AUTOSAR](#)

[Embedded Coder Reference](#)

[Embedded Coder Release Notes](#)

Support for AUTOSAR Release 4.1

AUTOSAR 4.1 ARXML and C code generation

The software now supports AUTOSAR Release 4.1 (schema version 4.1.1) for import and export of arxml files and generation of AUTOSAR-compatible C code.

If you import schema version 4.1.1 arxml code into Simulink, the arxml importer detects and uses the schema version, and sets the schema version parameter in the model to 4.1.

For information on specifying an AUTOSAR schema version for code generation, see [Select an AUTOSAR Schema](#).

AUTOSAR 4.1 InitEvent support

Beginning in R2014b, you can model AUTOSAR initialization events (`InitEvents`), as defined in AUTOSAR schema version 4.1. You can use an `InitEvent` to designate an AUTOSAR runnable as an initialization runnable, and then map an initialization function to the runnable.

In previous releases, you could use AUTOSAR mode management to set up software component initialization. For example, you could define a `ModeDeclarationGroup` with a mode for setting up and initializing a software component. `InitEvent` provides a potentially lighter-weight alternative to the mode-based approach.

If you import `arxml` code that describes a runnable with an `InitEvent`, the `arxml` importer configures the runnable in Simulink as an initialization runnable.

Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For more information, see [Configure AUTOSAR Initialization Runnable](#).

AUTOSAR 4.1 provide-require port support

Beginning in R2014b, you can model AUTOSAR provide-require ports (`PRPorts`), as defined in AUTOSAR schema version 4.1. `PRPorts` are a third type of port, in addition to provide ports (`PPorts`) and require ports (`RPorts`), that can be associated with an AUTOSAR sender-receiver interface. For example, you can:

- Map a Simulink inport/outport pair to a data element of an AUTOSAR provide require port. Generated code complies with Simulink and AUTOSAR semantics.
- Import AUTOSAR provide-require ports for sender-receiver interfaces from ARXML files.
- Export AUTOSAR provide-require ports to ARXML files.

For more information, see [Configure AUTOSAR Provide-Require Port](#).

Multi-instance AUTOSAR atomic software components

In previous releases, AUTOSAR software components (SWCs) modeled in Simulink were single-instance. Beginning in R2014b, you can model multi-instance AUTOSAR SWCs in Simulink. For example, you can:

- Map and configure a Simulink model as a multi-instance AUTOSAR SWC, and validate the configuration.
- Generate C code with reentrant runnable functions and multi-instance RTE API calls.
- Verify AUTOSAR multi-instance C code with SIL and PIL simulations.
- Import and export multi-instance AUTOSAR SWC description XML files.

For more information and limitations, see [Multi-Instance Atomic Software Components](#).

AUTOSAR arxml import and export

AUTOSAR R4.x compliant data type support

AUTOSAR data types workflow improvements

R2014b provides enhanced AUTOSAR Release 4.x compliant data type support.

- For round-trip workflows involving AUTOSAR components originated outside MATLAB, the arxml importer and exporter preserve data type information and mapping for each imported AUTOSAR data type.
- For AUTOSAR components originated in Simulink, the software generates AUTOSAR application, implementation, and base types to preserve the information contained within Simulink data types.

For more information, see Release 4.x Data Types.

Application data type export control

For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated data types with storage types.

Beginning in R2014b, if you want to override the default behavior for generating application types, you can configure the arxml exporter to generate an application type, along with the implementation type and base type, for each exported AUTOSAR data type. For more information, see Control Application Data Type Generation.

DataMappingSet package and name control

In previous releases, for AUTOSAR software components created in Simulink, users did not have control over the AUTOSAR package and short name exported for AUTOSAR data type mapping sets. The arxml exporter generated the short name `DataMappingSet` for every data type mapping set. The exporter used a rule-based package path that was not configurable in Simulink.

Beginning in R2014b, you can control the package and short-name for data type mapping sets. To configure the data type mapping set package for export, set the `XMLOptions` property `DataMappingPackage` using the Configure AUTOSAR Interface dialog box or the AUTOSAR property set function. For example:

Additional Packages	
ApplicationDataType Package:	<input type="text"/>
SwBaseType Package:	<input type="text"/>
DataMappingSet Package:	<input type="text" value="/pkg/dt/DataTypeMappings"/>

The exported arxml uses the specified package. The default mapping set short-name is the component name ASWC prefixed to `DataTypeMappingsSet`. You can specify a short name for a data type mapping set using the AUTOSAR property function `addPackageableElement`.

For more information, see Configure DataMappingSet Package and Name.

Data initialization with ApplicationValueSpecification

AUTOSAR Release 4.0 introduced application data types and implementation data types, which represent the application-level physical attributes and implementation-level attributes of AUTOSAR data types. To initialize AUTOSAR data objects typed by application data type, R4.1 requires AUTOSAR application value specifications (`ApplicationValueSpecifications`).

Beginning in R2014b, for AUTOSAR data initialization with `ApplicationValueSpecification`, Embedded Coder provides the following support:

- The arxml importer uses `ApplicationValueSpecifications` found in imported arxml files to initialize the corresponding data objects in the Simulink model.

- If you select AUTOSAR schema 4.0 or later for a model that contains AUTOSAR data typed by application data type, code generation exports a `arxml` code that uses `ApplicationValueSpecifications` to specify initial values for AUTOSAR data.

AUTOSAR CompuMethod control

CompuMethod direction for linear functions

In previous releases, Embedded Coder software imported AUTOSAR computational methods (CompuMethods) described in `arxml` code and preserved them across round-trips between an AUTOSAR authoring tool (AAT) and Simulink. For designs originated in Simulink, the `arxml` exporter created schema-compliant CompuMethods, but did not allow users control over CompuMethod attributes, including the direction of CompuMethod conversion between internal and physical representations of a value. For CompuMethods originated in Simulink, the exporter generated only the forward, internal-to-physical direction.

Beginning in R2014b, you can control how conversion direction is described in exported CompuMethods. Using either the Configure AUTOSAR Interface dialog box or the AUTOSAR property set function, you can specify one of the following CompuMethod direction values:

- `InternalToPhys` (default) — Generate CompuMethod sections for conversion of internal values into their physical representations.
- `PhysToInternal` — Generate CompuMethod sections for conversion of physical values into their internal representations.
- `Bidirectional` — Generate CompuMethod sections for both internal-to-physical and physical-to-internal conversion directions.

For more information, see [CompuMethod Direction for Linear Functions](#).

CompuMethod generated for each ApplicationDataType

In previous releases, the `arxml` exporter preserved AUTOSAR computational methods (CompuMethods) that you imported into Simulink, but for designs originated in Simulink, generated CompuMethods only for fixed point application types.

Beginning in R2014b, the exporter generates CompuMethods for every primitive application type. Measurement and calibration tools can monitor and interact with more application data. For more information, see [CompuMethod Categories for Data Types](#).

Unit reference generated for each CompuMethod

In previous releases, exported CompuMethods did not contain unit references. Beginning in R2014b:

- The `arxml` importer preserves unit and physical dimension information found in imported CompuMethods. The software preserves CompuMethod unit and physical dimension information across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.
- For designs originated in Simulink, the exporter generates a unit reference for each CompuMethod.

Providing a unit for each exported CompuMethod helps support measurement and calibration tool use of exported AUTOSAR data. For more information, see [CompuMethod Unit References](#).

Rational function CompuMethod for dual-scaled parameter

R2014b provides greater control over the AUTOSAR CompuMethods generated for AUTOSAR dual-scaled parameters. For an AUTOSAR dual-scaled parameter, which stores two scaled values of the

same physical value, the software generates the `CompuMethod` category `RAT_FUNC`. The computation method can be a first-order rational function. For more information, see [Rational Function CompuMethod for Dual-Scaled Parameter](#).

Improved AUTOSAR package configuration

In previous releases, the `arxml` exporter generated a fixed file and package structure for packaging AUTOSAR elements. Beginning in R2014b, Embedded Coder software provides more flexible configuration and management of AUTOSAR packages. For example:

- AUTOSAR packages and their elements now are fully preserved across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.
- AUTOSAR XML options in Simulink include ten new packaging parameters (`XmlOptions` properties). You can now easily group AUTOSAR elements of the following categories into packages:
 - Application data types (schema 4.x)
 - Software base types (schema 4.x)
 - Data type mapping sets (schema 4.x)
 - Constants and values
 - Physical data constraints (referenced by application data types or data prototypes)
 - System constants (schema 4.x)
 - Software address methods
 - Mode declaration groups
 - Computational methods
 - Units and unit groups (schema 4.x)

For more information, see [Configure AUTOSAR Package Structure](#).

AUTOSAR calibration component export

In previous releases, the software exported an AUTOSAR calibration component (`ParameterSwComponent`) only if it had been created in an AUTOSAR authoring tool (AAT) and imported into Simulink from an `arxml` file.

Beginning in R2014b, the software can export an AUTOSAR calibration component originated in Simulink. To configure AUTOSAR parameters for export in a calibration component, use the custom storage class (CSC) `CalPrm` with `AUTOSAR.Parameter` data objects. For more information, see [Model AUTOSAR Calibration Parameters and Configure AUTOSAR Calibration Component](#).

Simulink Min and Max mapping to AUTOSAR physical data constraints

Beginning in R2014b, in models configured for AUTOSAR, the software maps minimum and maximum values for Simulink data to the corresponding physical constraint values for AUTOSAR application data types. Specifically:

- If you import ARXML files, `PhysConstr` values on `ApplicationDataTypes` in the ARXML files are imported to `Min` and `Max` values on the corresponding Simulink data objects and root-level I/O signals.

- When you export ARXML from a model, the Min and Max values specified on Simulink data objects and root-level I/O signals are exported to the corresponding `ApplicationDataType` `PhysConstrs` in the ARXML files.

AUTOSAR `addPackageableElement` replaces `add*Interface` functions

R2014b introduces a new AUTOSAR property function, `addPackageableElement`, for adding packaged elements to the AUTOSAR configuration of a model. The function syntax is:

```
addPackageableElement(arProps, category, package, name)
addPackageableElement(arProps, category, package, name, property, value)
```

See the `addPackageableElement` reference page. For an example of using `addPackageableElement` as part of configuring a `DataTypeMappingSet` element for an AUTOSAR model, see “`DataTypeMappingSet` package and name control” on page 18-13.

Compatibility Considerations

Using the function `addPackageableElement` with element categories `ModeSwitchInterface` or `SenderReceiverInterface` replaces the following equivalent AUTOSAR property functions:

- `addMSInterface(arProps, qName, property, value)`
- `addSRInterface(arProps, qName, property, value)`

If an existing script calls `addMSInterface` or `addSRInterface`, replace the call with an equivalent call to `addPackageableElement`. For example, consider the `addSRInterface` call in the following code:

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addSRInterface(arProps, '/pkg/if/Interface3', 'IsService', true);
ifPaths=find(arProps, [], 'SenderReceiverInterface', ...
    'IsService', true, 'PathType', 'FullyQualified')
```

Replace the `addSRInterface` call with an equivalent `addPackageableElement` call. For example:

```
open_system('rtwdemo_autosar_multirunnables');
arProps=autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
addPackageableElement(arProps, 'SenderReceiverInterface', '/pkg/if', 'Interface3', ...
    'IsService', true);
ifPaths=find(arProps, [], 'SenderReceiverInterface', ...
    'IsService', true, 'PathType', 'FullyQualified')
```

Code generation report with enhanced navigation and integrated access to code metrics data

In R2014b, the following enhancements improve navigation and access to code metrics in the code generation report:

- Model-to-code navigation toolbar at the top of the code window with buttons to navigate forward and backward through the highlighted code for a model block.
- Lines in a navigation sidebar show the locations of the highlighted code in the current file. Hovering your cursor over a line shows you the code line number. Clicking the line takes you directly to the code.

- Code inspect window provides code metrics and links to definitions when you click linked variables or functions in the code.
- Hovering your cursor over global variables and functions in the code window opens a window with code metrics data.

For more information, see [Trace Model Objects to Generated Code](#) and [View Code Metrics and Definitions in the Generated Code](#).

Updated license requirements for viewing code generation report

In 2014b, if you open a code generation report from a MATLAB menu, the software checks out the same licenses that were required when you created the report at the time of code generation. You can view the HTML report in a Web browser, but the following code generation report features are not available:

- Traceability between the code and the model.
- Code metrics data when you hover over global variables and functions in the code window.

Compatibility Considerations

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

Option for doxygen style comments in generated code

You can now specify doxygen style comments in a code generation template (CGT) file. The style attribute options for these comments are `doxygen`, `doxygen_cpp`, `doxygen_qt`, and `doxygen_qt_cpp`.

`doxygen` with C style comments

```
/**
 * multiple line comments
 * second line
 */
```

`doxygen_cpp` with C++ style comments

```
///
/// multiple line comments
/// second line
///
```

`doxygen_qt` with C style comments

```
/*!
 * multiple line comments
 * second line
 */
```

`doxygen_qt_cpp` with C++ style comments

```
/*!
/*! multiple line comments
```

```

//! second line
//!

```

For more information on using code generation template files to customize file and function banners, see [Generate Custom File and Function Banners](#).

Dynamic memory allocation parameters renamed

On the **Code Generation > Interface** pane, two dynamic memory allocation parameters are renamed.

Code Generation > Interface pane		Command line (unchanged)
R2014b	R2014a	
Use dynamic memory allocation for model initialization	Generate function to allocate model data	GenerateAllocFcn
Use dynamic memory allocation for model block instantiation	Use operator new for referenced model object registration	UseOperatorNewForModelRefRegistration

The command line names are unchanged.

Template makefile compatibility with execution time profiling

Consider a custom target that requires a template makefile (TMF) where the SHARED_OBJJS definition is based on SHARED_SRC. If you specify code execution profiling for your model, you might observe a failure when you try to build the model. The failure occurs because the folder that contains the shared utility object files is different from the folder that contains the corresponding source code. How you fix this issue depends on how SHARED_OBJJS is defined in your TMF. For example, you must replace:

```
SHARED_OBJJS = $(SHARED_SRC:.c=.obj)
```

with:

```
SHARED_OBJJS = $(SHARED_BIN_DIR)\*.obj
```

For more information, see [Customize Build to Use Shared Utility Code](#).

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 cannot 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

Embedded Coder support packages for AUTOSAR, TI Concerto, and Freescale FRDM-KL25Z

R2014b adds the following Embedded Coder support packages:

- Embedded Coder Support Package for AUTOSAR Standard — You can create and modify an AUTOSAR configuration for a model, model AUTOSAR elements, and generate ARXML and AUTOSAR-compatible C code from a model. For more information, see Support Package for AUTOSAR Standard.
- Embedded Coder Support Package for Texas Instruments C2000 F28M3x Concerto Processors — You can generate, build, and deploy code on Texas Instruments C2000 F28M35x/ F28M36x Concerto processors. For more information, see Support for Texas Instruments C2000 F28M3x Concerto Processors.
- Embedded Coder Support Package for Freescale FRDM-KL25Z Board — You can generate, build, and deploy a control algorithm on Freescale FRDM-KL25Z boards. For more information, see Support package for Freescale FRDM-KL25Z Board.

Relational operator replacement

You can now include code replacement mappings for basic relational operators (<, <=, >, >=, ==, and !=) in custom code replacement libraries. You can apply relational operator mappings to scalar, vector, or matrix data.

For more information, see Scalar Operator Code Replacement and Small Matrix Operation to Processor Code Replacement.

Code replacement involving vector and matrix data

- “Trigonometry function replacement” on page 18-20
- “Replacement of shift and cast operations involving vector and matrix operands” on page 18-21

Trigonometry function replacement

In R2014b, the C/C++ code generator supports code replacement of the following trigonometry functions for scalar, vector, and matrix input and for output arguments in code generated from:

- MATLAB functions
- MATLAB Function block
- MATLAB action language in Stateflow charts

Supported base types include floating point, complex, and noncomplex.

acos	asec	atand	cscd	sech
acosd	asecd	cos	csch	sin
acot	asech	cosd	hypot	sind
acotd	asin	cosh	log	sinh

acoth	asind	cot	log10	tan
acsc	atan	cotd	log2	tand
acscd	atan2	coth	sec	tanh
acsch	atan2d	csc	secd	

For more information, see [Map Math Functions to Application-Specific Implementations](#).

Replacement of shift and cast operations involving vector and matrix operands

In R2014b, you can specify code replacements for these vector and matrix operations:

- Cast (data type conversion), RTW_OP_CAST
- Shift Left, RTW_OP_SL
- Shift Right Arithmetic, RTW_OP_SRA
- Shift Right Logical, RTW_OP_SRL

For more information, see [Small Matrix Operation to Processor Code Replacement](#).

Algorithm specification for addition and subtraction operator replacement

Starting with R2014b, you can specify the algorithm—cast-before-operation (default) or cast-after-operation—for addition and subtraction operations that must be matched for operator replacement to occur.

For more information, see [Addition and Subtraction Operator Code Replacement](#).

Compatibility Considerations

By default, the code generator attempts to replace addition and subtraction operations as cast-before-operation algorithms. This replacement matches the behavior in R2013a through R2014a. If the code generator cannot classify an operation strictly as a cast-before-operation, some replacements for non-binary-point operations do not occur. For more information, see [Addition and Subtraction Operator Code Replacement](#).

If you are using a code replacement library developed with an earlier release, verify code replacements for addition and subtraction operators. For information, see [Review and Test Code Replacements](#).

Improved code replacement with output type cast absorption

Starting in R2014b, the code generator includes downcasts on the output of addition, subtraction, multiplication, and division operations involving real, scalar, and fixed-point data for code replacements.

For example, consider a case where `u1` and `u2` are of type `integer`. `y1` is of type `short` and the operation being replaced is `y = (short) (u1*u2)`. In previous releases, the multiplication operation was replaced without including the output cast.

```
y = (short) (my_mul_output_integer(u1, u2));
```

In R2014b, you can register an additional table replacement entry to get the following replacement:

```
y = my_mul_output_short(u1, u2);
```

The code generator does not handle intermediate casts for code replacement.

Lookup table function code replacement extended to 30 dimensions

R2014b introduces functions `interpND` and `lookupND`. You can specify these functions to increase the dimension support of code replaced for the Interpolation Using Prelookup and n-D Lookup Table blocks to 30. The conceptual signature that you specify for the code replacement table entry depends on the number of dimensions that you want the function to support.

For more information, see [Lookup Table Function Code Replacement](#)

Rounding mode support for lookup table function replacement

As of R2014b, the code generator supports use of algorithm parameters **Integer rounding mode** (`RndMeth`) and **Saturate on integer overflow** (`SaturateOnIntegerOverflow`) in code replacement specifications for lookup table functions.

For more information, see [Lookup Table Function Code Replacement](#).

Algorithm parameter value sets in code replacement table entries

Prior to R2014b, code replacement table entries could specify multiple values for an algorithm parameter. However, you had to specify them in separate code replacement table entries. For example, to specify that a lookup table function with a linear or binary index search trigger a match for code replacement, you specified the following calls to `addAlgorithmProperty` in two separate table entries:

Entry 1:

```
addAlgorithmProperty('IndexSearchMethod','Linear search');
```

Entry 2:

```
addAlgorithmProperty('IndexSearchMethod','Binary search');
```

As of this release, you can specify multiple values in a single call to `addAlgorithmProperty` in one entry. Specify the value part of the parameter name-value pair as a set of string values. This specification reduces the lines of code required for more complex, conceptual specifications. For example:

```
addAlgorithmProperty('IndexSearchMethod', {'Linear search', ...  
                                             'Binary search'});
```

For more information, see [addAlgorithmProperty](#) and [Lookup Table Function Code Replacement](#).

coder.replace support for functions specified with varargin input variable

As of R2014b, the `coder.replace` function supports MATLAB functions that specify a variable-length input argument list by using a `varargin` input variable.

For more information, see `coder.replace`.

Documentation installation with hardware support package

Starting in R2014b, each hardware support package has its own documentation. For a list of Embedded Coder support packages, see [Embedded Coder Supported Hardware](#).

Support package for Altera SoC platform

You can use the Embedded Coder Support Package for Intel SoC Devices to generate, build, and deploy code to the Altera® Cyclone V SoC development kit or to the Arrow SoCKit development board. The executable runs in the Linux environment on the ARM Cortex-A9 processor on the Altera SoC platform.

See [Install Support for Altera SoC Platform](#).

For more information, see [Embedded Coder Support Package for Altera SoC Platform](#).

Support package for BeagleBone Black hardware

You can use the Embedded Coder Support Package for BeagleBone Black Hardware to generate, build, and deploy code to the BeagleBone Black board.

See [Install Support for BeagleBone Black Hardware](#).

For more information, see [Embedded Coder Support Package for BeagleBone Black Hardware](#).

Support for Eclipse IDE has been removed

Embedded Coder support for Eclipse™ IDE has been removed.

You can no longer use Embedded Coder with Eclipse IDE to build and run an executable on BeagleBoard hardware or ARM processors.

Compatibility Considerations

To replace some of the capabilities provided by Eclipse IDE, consider using:

- Embedded Coder Support Package for ARM Cortex-A Processors
- Simulink Support Package for BeagleBoard® Hardware

To install support packages, see `supportPackageInstaller`.

Support for Green Hills MULTI IDE has been removed

Embedded Coder support for Green Hills® MULTI® IDE has been discontinued for R2014b.

Compatibility Considerations

If you are using the Embedded Coder Support Package for Green Hills MULTI IDE, the support package is available for use with previous releases for an unspecified length of time.

Support for Texas Instruments C5000 DSPs will be removed

Support for Texas Instruments C5000™ DSPs will be removed in a future release.

Performance

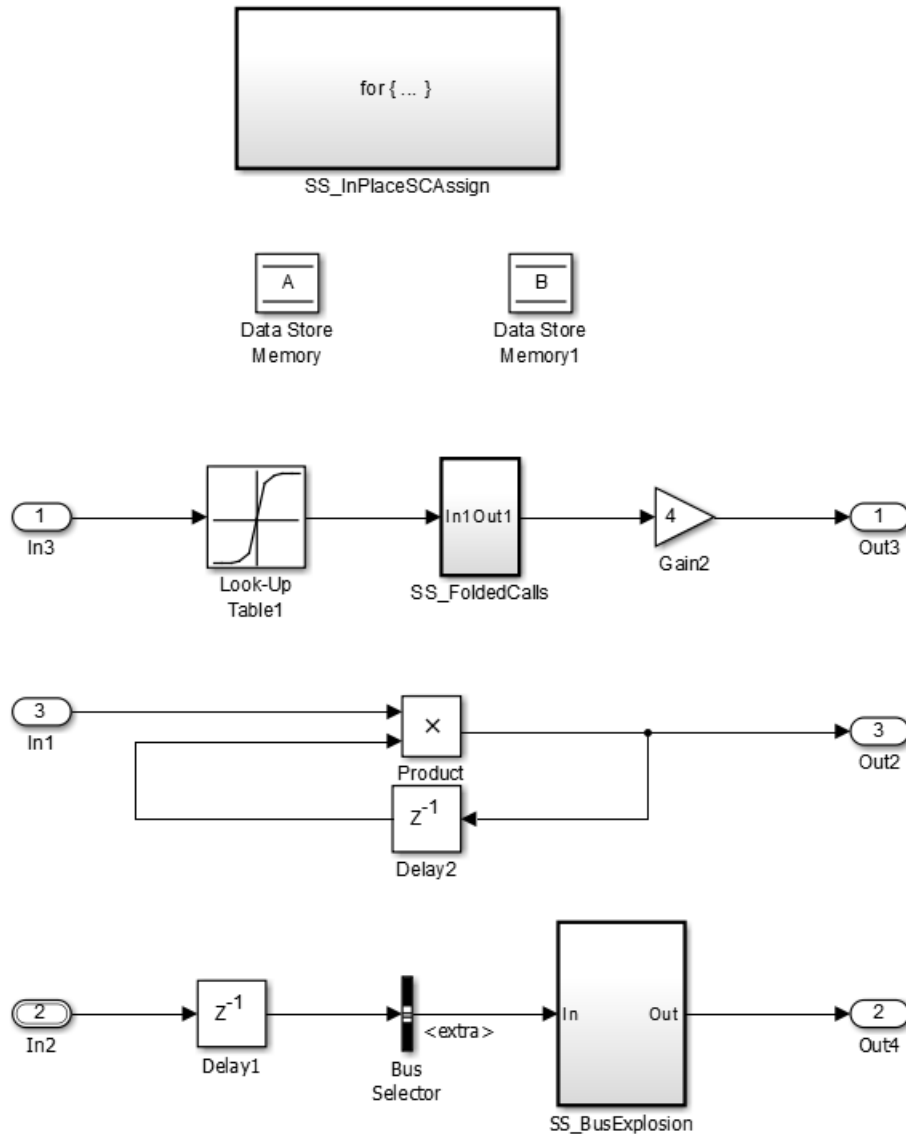
Reduced RAM and faster execution for modeling patterns including select-assign-iterate blocks, subsystem interfaces, and model references

- “Example Model” on page 18-25
- “In-place assignments for select-assign-iterate pattern” on page 18-26
- “Subsystem signal information” on page 18-28
- “Variable reuse around call site” on page 18-28

Code generation produces code with more optimizations, reducing RAM/ROM consumption and improving execution speed. The ability of the code generator to perform more optimizations is due to the following efficiency enhancements.

Example Model

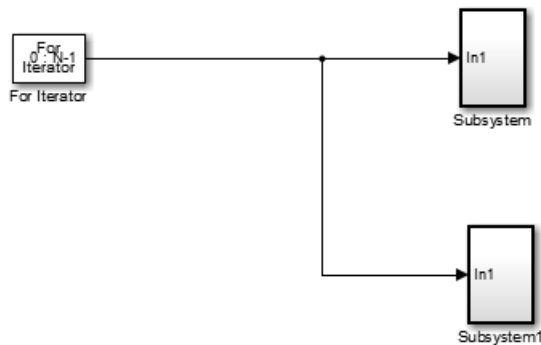
Consider the model `example_subsys1`, that contains the subsystem and models used for the examples for each optimization:



In-place assignments for select-assign-iterate pattern

The code generator generates in-place assignments for the select-assign-iterate modeling pattern for the three subsystem function packaging options.

Example subsystem `SS_InPlaceSCAssign`:



The code generator produces this code for version R2014a:

```

/* Output and update for atomic system '<S4>/Subsystem1'*/
void example_subsys1_Subsystem(int32_T rtu_In1)
{
    int32_T i;

    /* Assignment: '<S6>/Assignment' incorporates:
     * DataStoreRead: '<S6>/Data Store Read'
     */
    if (example_subsys1_Dwork.ForIterator_IterationMarker<2){
        example_subsys1_Dwork.ForIterator_IterationMarker=2U;
        for(i=0;i<30;i++){
            example_subsys1_B.Assignment[i]=example_subsys1_DWork.B[i];
        }
    }

    example_subsys1_B.Assignment[rtu_In1]=rtu_In1;

    /* End of Assignment: '<S6>/Assignment'*/

    /* DataStoreWrite: '<S6>/DataStoreWrite'*/
    for(i=0;i<30;i++){
        example_subsys1_DWork.B[i]=example_subsys1_B.Assignment[i];
    }

    /* End of DataStoreWrite: '<S6>/DataStoreWrite'*/
}
  
```

The code generator produces this code for version R2014b:

```

/* Output and update for atomic system: '<S3>/Subsystem1' */
void example_subsys1_Subsystem1(int32_T rtu_In1)
{
    /* Assignment: '<S5>/Assignment' */
    if (example_subsys1_DWork.ForIterator_IterationMarker < 2) {
        example_subsys1_DWork.ForIterator_IterationMarker = 2U;
    }

    example_subsys1_DWork.B[rtu_In1] = rtu_In1;

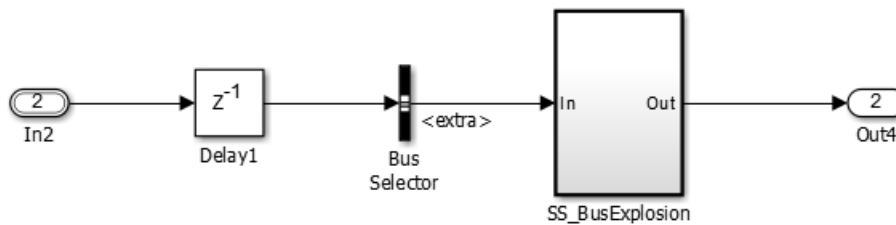
    /* End of Assignment: '<S5>/Assignment' */
}
  
```

The code generator produces less code, does not use iteration loops, and uses fewer variable references.

Subsystem signal information

The code generator has more information about signals passing through the subsystem boundary. It uses that information to generate more fully optimized code.

The code generator generates less code for this model:



The code generator produces this code for version R2014a:

```
BigBus rtb_Delay1;
/* Delay: '<Root>/Delay1' */
rtb_Delay1 = example_subsys1_DWork.Delay1_DSTATE;

/* Outputs for Atomic SubSystem: '<Root>/SS_BusExplosion' */
example_subsys1_SS_BusExplosion(rtb_Delay1.extra);

/* End of Outputs for SubSystem: '<Root>/SS_BusExplosion'*/
```

The code generator produces this code for version R2014b:

```
/* Delay: '<Root>/Delay1' */
example_subsys1_SS_BusExplosion(example_subsys1_DWork.Delay1_DSTATE.extra);

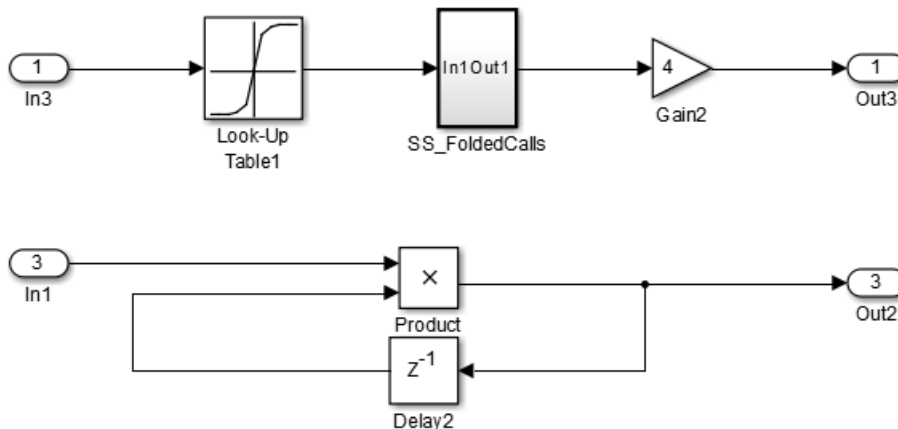
/* End of Outputs for SubSystem: '<Root>/SS_BusExplosion' */
```

The generated code requires fewer variables and fewer statements.

Variable reuse around call site

The code generator reuses variables around subsystem function call sites.

Example model:



The code generator produces this code for version R2014a:

```

/* Delay: '<Root>/Delay2'*/
for (i=0;i<12;i++){
  rtb_Delay2[i] = example_subsys1_DWork.Delay2_DSTATE[i];
}

/* End of Delay: '<Root>/Delay2'*/

for (i=0;i<12;i++){
  /*Product '<Root>/Product' incorporates:
  *Inport: '<Root>/In1'
  */
  rtb_Delay2_i=example_subsys1_U.In1[i]*rtb_Delay2[i];

  /*Outport: '<Root>/Out2'*/
  example_subsys1_Y.Out2[i]=rtb_Delay2_i;

  /* Product '<Root>/Product'*/
  rtb_Delay2[i]=rtb_Delay2_i;
}

/*Update for Delay: '<Root>/Delay2'*/
for (i=0;i<12;i++){
  example_subsys1_DWork.Delay2_DSTATE[i] = rtb_Delay2[i];
}

/*End of Update for Delay: '<Root>/Delay2'*/

```

The code generator produces this code for version R2014b:

```

/* Product: '<Root>/Product' incorporates:
* Delay: '<Root>/Delay2'
* Inport: '<Root>/In1'
*/
for (rtb_DataTypeConversion = 0; rtb_DataTypeConversion < 12;
    rtb_DataTypeConversion++) {
  example_subsys1_Y.Out2[rtb_DataTypeConversion] *=
    example_subsys1_U.In1[rtb_DataTypeConversion];
}

```

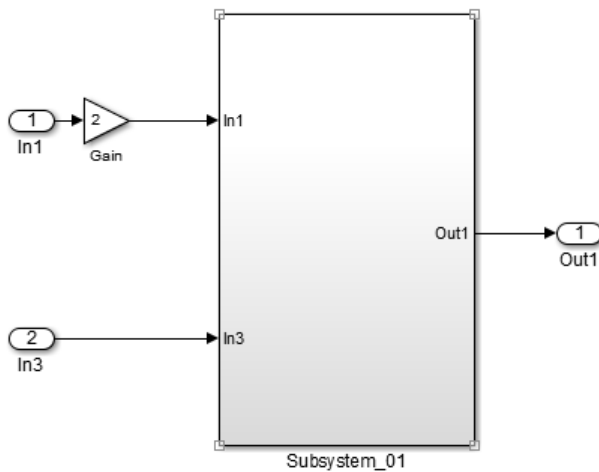
```
/* End of Product: '<Root>/Product' */
```

The code generator produces much less code, including one iteration loop instead of three iteration loops. It produces fewer variable references with the same functionality.

Global variable localization optimizations

When you generate code for a model, the code generator optimizes variable references by replacing global variables with local variables. This replacement improves execution speed and reduces RAM/ROM.

Consider this model, named `exlocal`:



Observe the following lines of code generated for R2014a in the `exlocal_ert_rtw` folder, in the `exlocal.c` file, in the `Model` step function.

```
exlocal_B.Gain[0] = 2.0 * exlocal_U.In1[0];
exlocal_B.Gain[1] = 2.0 * exlocal_U.In1[1];
exlocal_B.Gain[2] = 2.0 * exlocal_U.In1[2];
exlocal_B.Gain[3] = 2.0 * exlocal_U.In1[3];
```

and

```
exlocal_Subsystem_03(exlocal_U.In3, exlocal_B.Gain, &exlocal_Y.Out1);
```

Compare the same lines of code generated for R2014b.

```
Gain[0] = 2.0 * exlocal_U.In1[0];
Gain[1] = 2.0 * exlocal_U.In1[1];
Gain[2] = 2.0 * exlocal_U.In1[2];
Gain[3] = 2.0 * exlocal_U.In1[3];

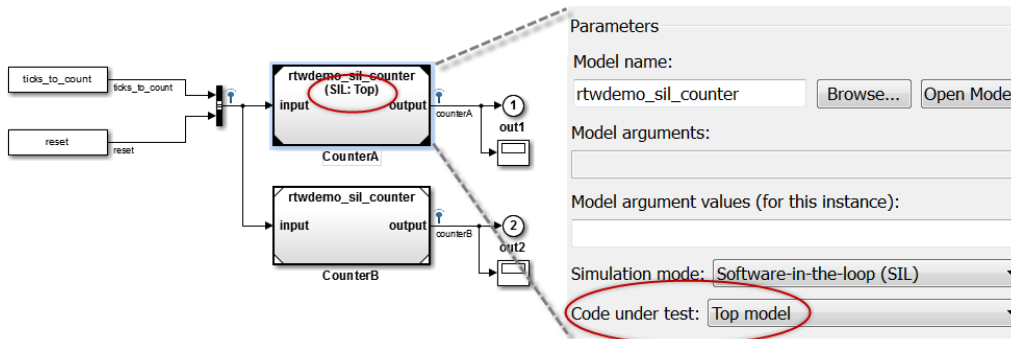
exlocal_Subsystem_03(Gain, exlocal_U.In3, &exlocal_Y.Out1);
```

For more information, see [Specify Global Variable Localization](#).

Verification

Top-model code testing with Model block SIL and PIL

You can run Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations to test code that is generated from a top model. This feature enables you to use a model-based test harness to verify code for a deployable top-level component. You can create test cases, switch easily between simulation modes, and analyze numerical results.



If you set the Model block parameter, **Simulation mode** (SimulationMode), to **Software-in-the-loop (SIL)** or **Processor-in-the-loop (PIL)**, the software provides a new parameter **Code under test** (CodeUnderTest) with the following options:

- **Top model** — Code generated from top model, with the standalone code interface. Previously, this code was tested by running a top-model SIL or PIL simulation or by creating a SIL or PIL block.
- **Model reference** (default) — Code generated from referenced model as part of a model reference hierarchy, which was previously the only behavior available for Model block SIL and PIL.

For more information, see [Referenced Model Simulation Using SIL or PIL](#).

SIL/PIL support for Simulink Function and Function Caller blocks

Use top-model and Model block SIL or PIL simulations to verify code generated from models that have Simulink Function and Function Caller blocks.

The software does not support SIL or PIL block verification for these blocks. Use the Model block SIL/PIL approach, with the **Code under test** block parameter set to **Top model**.

For more information, see:

- [Simulink Functions: Create and call functions across Simulink and Stateflow](#)
- [Choose a SIL or PIL Approach](#)

SIL debugging support for Linux

On a Linux system, you can use the GNU Data Display Debugger (DDD) to observe code behavior during a SIL simulation.

Previously, SIL debugging was available only for a Windows system.

For more information, see [Debug Code During SIL Simulations](#).

PIL support for test hardware approach

You can run processor-in-the-loop (PIL) simulations when the **Configuration Parameters > Hardware Implementation > Test hardware is the same as production hardware** check box is not selected.

SIL/PIL support for model initialization dynamic memory allocation

You can run SIL/PIL simulations with models that dynamically allocate memory for model data structures.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.6

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Template to customize code generation output for MATLAB Coder

You can use the `coder.MATLABCodeTemplate` class to customize code generation output for MATLAB Coder. Using a default or custom template, you can set token values to customize file banners, function banners, and file trailers.

For more information, see [Generate Custom File and Function Banners for C and C++ Code](#).

Compatibility Considerations

Beginning in R2014a, the code generator adds file and function banners to generated code by default. If you do not specify a code generation template (CGT) file to customize the banners, the code generator uses the default template file, `matlabcoder_default_template.cgt`, in the `matlabroot/toolbox/coder/matlabcoder/templates/` folder.

In-place function replacement with `coder.replace` in MATLAB

In R2014a, you can create code replacement table entries that specify in-place function replacement if you are generating C or C++ code from MATLAB code directly or from a MATLAB Function block. In-place code replacement is an optimization technique that uses a single buffer, that is, the same memory, to store function input and output data, as in `x=foo(x)`.

For more information, see [Specify In-Place Code Replacement and `coder.replace`](#).

Single-line (//) comment style available for generated code

In earlier releases, C and C++ code generation always used a multi-line (`/*...*/`) comment style. Beginning in R2014a, you can select a single-line (`//...`) comment style for generated code.

Set the comment style in one of the following ways:

- In a project, in the Project Settings dialog box **Code Appearance** tab, set **Comment Style** to one of the following values.

Value	Description
Auto(Use standard comment style of the target language)	For C, generate multi-line comments. For C++, generate single-line comments. (default)
Single-line (Use C++-style comments)	Generate single-line comments preceded by <code>//</code> .
Multi-line (Use C-style comments)	Generate single or multi-line comments delimited by <code>/*</code> and <code>*/</code> .

- At the command prompt, create a code generation configuration object. Set the `CommentStyle` parameter to one of the following values.

Value	Description
'Auto'	For C, generate multi-line comments. For C++, generate single-line comments. (default)
'Single-line'	Generate single-line comments preceded by //.
'Multi-line'	Generate single or multi-line comments delimited by /* and */.

For example, the following code sets the comment style to single-line comments:

```
cfg = coder.config('lib');
cfg.CommentStyle='Single-line';
```

Here is an example of generated code that uses single-line comments:

```
//
// mcadd.c
//
// Code generation for function 'mcadd'
//
```

Software-in-the-loop verification for MATLAB Coder

The following table summarizes software-in-the-loop (SIL) execution improvements.

Feature		R2014a support	Previous support
Output type	Dynamic library	Yes	No
Interface types	Constant inputs	Yes	Yes. If values passed through the SIL interface differ from the values used by the build process, the SIL execution uses the build values. The execution does not generate an error or warning.
	Constant global data	Yes. If values passed through the SIL interface differ from the values used by the build process, the SIL execution uses the build values. The execution does not generate an error or warning.	Not applicable.
Data types	Fixed-point data	Yes	Yes, with limitations.
	Multiword fixed-point data	Yes	No
	Empty values	Yes	No

Feature		R2014a support	Previous support
Size	Static variable-size arrays	Variable-size function arguments are not supported. For function arguments that are fixed-size structures, variable-size fields are supported.	No

For more information, see [SIL Execution Support and Limitations](#).

Change of default value for `MATLABFcnDesc`

Previously, the `MATLABFcnDesc` parameter of a `coder.EmbeddedCodeConfig` code generation configuration object had a default value of `false`. In R2014a, the default value of the `MATLABFcnDesc` parameter is `true`. When the value of the `MATLABFcnDesc` parameter is `true`, the MATLAB function help text is included in a function banner in generated code.

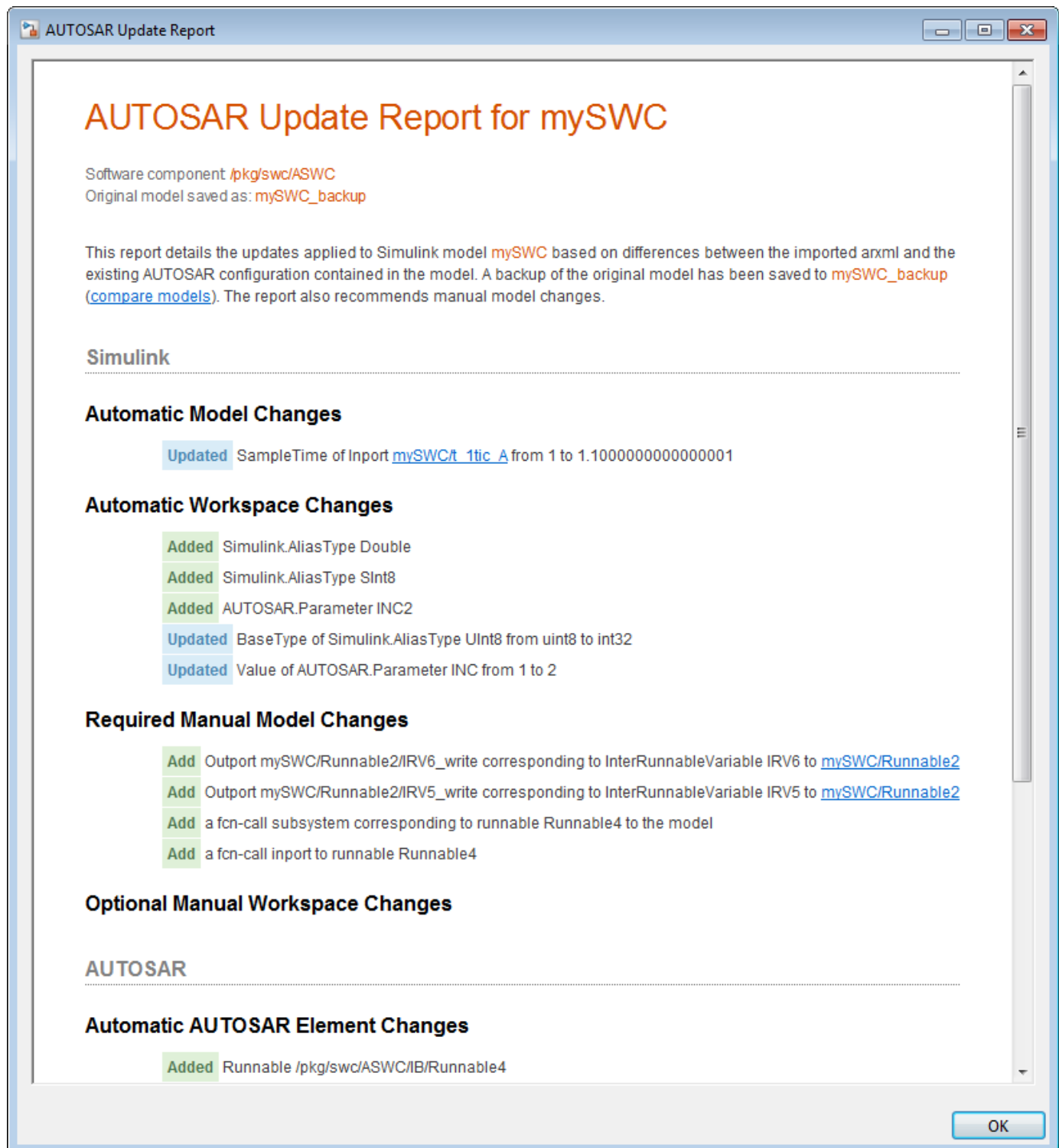
Model Architecture and Design

Capability to merge AUTOSAR authoring tool changes into Simulink models as part of round-trip iterations

To help support the round trip of AUTOSAR components between an AUTOSAR authoring tool (AAT) and the Simulink design environment, R2014a adds update and merge capabilities to the `arxml` importer.

Given a Simulink model into which you have imported `arxml` code or from which you have exported `arxml` code, suppose that changes have been made to the `arxml` information in an AAT. Using the `arxml.importer.updateModel` method, you can import the changed `arxml` information and request that the changes be merged into the model. The update/merge generates a report that details the updates applied to the model, and required changes that were not made automatically.

Here is an example of a generated AUTOSAR update report. For more information, see [Merge AUTOSAR Authoring Tool Changes Into a Model](#).



AUTOSAR 4.0 static and constant memory, AUTOSAR-typed per-instance memory, and VariationPointProxy

Static and constant memory

Beginning in R2014a, from a Simulink model, you can import and export AUTOSAR Static Memory and Constant Memory data, as defined by AUTOSAR schema version 4.0. Static Memory corresponds to Simulink internal global signals. Constant Memory corresponds to Simulink internal global parameters. When exported in arxml, Static Memory and Constant Memory allow the use of measurement and calibration tools to monitor the internal memory data.

For more information, see Model AUTOSAR Static and Constant Memory and Configure AUTOSAR Static or Constant Memory

AUTOSAR-typed per-instance memory

Beginning in R2014a, you can model AUTOSAR-typed per-instance memory (`arTypedPerInstanceMemory`) in Simulink models. This class of per-instance memory was introduced in AUTOSAR schema version 4.0. You describe `arTypedPerInstanceMemory` using standard AUTOSAR data types (rather than C types). When exported in arxml, `arTypedPerInstanceMemory` allows the use of measurement and calibration tools to monitor the global variable corresponding to per-instance memory.

For more information, see Model AUTOSAR Per-Instance Memory, Configure AUTOSAR Per-Instance Memory, and the example model `rtwdemo_autosar_PIM`, which has been updated to use `arTypedPerInstanceMemory`.

Variation point proxy

Beginning in R2014a, you can model an AUTOSAR `VariationPointProxy`, as defined in AUTOSAR schema 4.0. The Simulink elements include:

- Variant Subsystem or Model Variant block to model a `VariationPointProxy` inside an AUTOSAR runnable.
- `AUTOSAR.Parameter` data objects to model AUTOSAR System Constants, representing the conditional values associated with the variant condition logic.
- `Simulink.Variant` data objects in the base workspace to define the variant condition logic.

For more information, see Configure AUTOSAR Variation Point Proxies.

Specify AUTOSAR runnable symbol name distinct from short-name

In previous releases, Embedded Coder derived the symbol name of an AUTOSAR runnable from the user-specified short-name. Beginning in R2014a, you can specify an AUTOSAR runnable symbol name that is distinct from the runnable short-name. The runnable symbol-name can be specified using the Configure AUTOSAR Interface dialog box or by using the AUTOSAR property functions. The specified AUTOSAR runnable symbol-name is exported in arxml and C code. Also, you can import a runnable symbol name using the arxml importer.

For example, suppose that you open the example model `rtwdemo_autosar_multirunnables`, open the Configure AUTOSAR Interface dialog box, and use the Runnables view of the AUTOSAR Properties Explorer to change the symbol-name of `Runnable1` from `Runnable1` to `test_symbol`.

When you export code from the model, the symbol-name `test_symbol` appears in the exported `arxml` and C code as shown below.

Example 19.1. `rtwdemo_autosar_multirunnables.arxml`

```
<RUNNABLE-ENTITY UUID="65432c3e-34c7-5e82-4229-f6d04927eb78">
  <SHORT-NAME>Runnable1</SHORT-NAME>
  ...
  <SYMBOL>test_symbol</SYMBOL>
  ...
</RUNNABLE-ENTITY>
```

Example 19.2. `rtwdemo_autosar_multirunnables.c`

```
/* Output function for RootInportFunctionCallGenerator:
   '<Root>/RootFcnCall_InsertedFor_Runnable1_at_outport_1' */
void test_symbol(void)
{
  ...
}
```

For more information, see

- Configure AUTOSAR Component Using AUTOSAR Properties Explorer, step 8
- API example Set Runnable Symbol Name

Improved AUTOSAR `arxml` support for measurement and calibration

Embedded Coder now supports `arxml` import and export of the following AUTOSAR software data definition properties (`SwDataDefProps`):

- Software calibration access (`SwCalibrationAccess`) — Specifies measurement and calibration tool access to a data object.
- Software address method (`swAddrMethod`) — Specifies a method to access a data object (for example, a measurement or calibration parameter) according to a given address.
- Software alignment (`swAlignment`) — Specifies the intended alignment of a data object within a memory section.
- Software implementation policy (`swImplPolicy`) — Specifies the implementation policy for a data object, with respect to consistency mechanisms of variables.

In the Simulink environment, you can directly modify the `SwCalibrationAccess`, `swAddrMethod`, and `swAlignment` properties for some forms of AUTOSAR data. (You cannot modify the `swImplPolicy` property.) For more information, see [Configure AUTOSAR Data for Measurement and Calibration](#).

AUTOSAR data dictionary support

Beginning in R2014a, you can use a Simulink data dictionary in AUTOSAR workflows. For example, you can:

- Import AUTOSAR data and parameter objects into a data dictionary, instead of into the MATLAB base workspace.
- Leverage Simulink data dictionary object properties as you edit AUTOSAR data objects.
- Export `arxml` and C code reflecting the data dictionary object properties configured for the model.

For more information about importing data and parameter objects into a data dictionary, see the `DataDictionary` property for methods `arxml.importer.createComponentAsModel` and `arxml.importer.createCalibrationComponentObjects`.

Configure AUTOSAR Interface button removed from AUTOSAR Code Generation Options

The **Configure AUTOSAR Interface** button has been removed from the **AUTOSAR Code Generation Options** pane of the Simulink Configuration Parameters dialog box. The remaining content of the pane pertains directly to configuring AUTOSAR arxml and C code generation.

To configure an AUTOSAR interface for a model, open the model, check that the AUTOSAR target (`autosar.tlc`) is selected for the model, and do either of the following:

- In the Simulink Editor window, select **Code > C/C++ Code > Configure Model as AUTOSAR Component**.
- In the MATLAB command window, enter the command `autosar_ui_launch(model)`.

If your model is already configured for AUTOSAR, this action opens the Configure AUTOSAR Interface dialog box. If your model is not configured for AUTOSAR, dialog boxes first help you create an AUTOSAR interface, then open the Configure AUTOSAR Interface dialog box with the initial configuration displayed.

Subsystem methods of AUTOSAR arxml.importer class removed

Two subsystem-related methods of the `arxml.importer` class have been removed from the software:

- `arxml.importer.createComponentAsSubsystem` — Create AUTOSAR atomic software component as Simulink atomic subsystem.
- `arxml.importer.createOperationAsConfigurableSubsystems` — Create configurable Simulink subsystem library for client-server operation.

You now can model AUTOSAR multi-runnables as function-call subsystems at the top level of a model, rather than as function-call subsystems within a wrapper subsystem that represents the AUTOSAR software component.

Compatibility Considerations

If you are using `createComponentAsSubsystem` or `createOperationAsConfigurableSubsystems`, migrate to using the top-model-oriented approach described in Configure AUTOSAR Multiple Runnables.

Data, Function, and File Definition

Custom storage class and optimized class declarations for C++ class code generation

Custom storage class support for C++ class code generation

In previous releases, custom storage classes (CSCs) were not supported for C++ class code generation. Selecting C++ (Encapsulated) for a model forced on the model option **Ignore custom storage classes**.

Beginning in R2014a, you can use CSCs with C++ class code generation. The configuration requirements for using CSCs with C++ class code generation include the following Configuration Parameters dialog box settings:

- **Code Generation > Interface** pane:
 - Set **Code interface packaging** to C++ class.
 - Set **Multi-instance code error diagnostic** to a value other than Error.
- **Code Generation** pane: Clear the option **Ignore custom storage classes**.

For more information and limitations, see Specify Custom Storage Class for C++ Class Code Generation.

Improved code for C++ model class declarations

R2014a enhances generated C++ model class declarations in the following ways:

- Automatically adds a copy constructor and an assignment operator to C++ class declarations when required to securely handle pointer members.
- Removes an unnecessary `rtModel` pointer declaration from C++ class declarations.

For more information, see Model Class Copy Constructor and Assignment Operator.

Constant sample time limitations for root-level Output blocks

In R2014a, the sample time of root-level Output blocks is checked in the following ways:

- For models using Function Prototype Control or a C++ class interface, the validation check reports an error if a root-level Output block has a constant sample time.
- For models using the AUTOSAR target, the compiler reports a warning if a root-level Output block is configured to inherit a constant sample time from its sources. The compiler then sets the sample time of the root-level Output block to the fundamental rate of the model. This warning will become an error in a future release.
- When importing an AUTOSAR model from an XML description of a single runnable, the import tool sets the sample time of root-level Output blocks to the fundamental rate of the model.
- The Upgrade Advisor adds a check identifying root-level Output blocks with a constant sample time. If a model uses the AUTOSAR target, Function Prototype Control, or a C++ class interface, the check lists the Output blocks with a constant sample time. The check also includes possible actions to fix the blocks.

Example model `rtwdemo_cppencap` renamed to `rtwdemo_cppclass`

As part of the C++ class code interface packaging changes described in Simulink Coder release note Improved control of C and C++ code interface packaging, C++ class example model `rtwdemo_cppencap` has been renamed to `rtwdemo_cppclass`.

Unit Delay block optimization

In R2014a, when you specify a nonzero initial value or a global storage class, global block output reuse might eliminate the Unit Delay state in the generated code. Eliminating the Unit Delay state reduces data copies.

Code Generation

In-place function replacement with `coder.replace` in MATLAB and lookup table code replacement for Simulink

In-place function replacement with `coder.replace` in MATLAB

In R2014a, you can create code replacement table entries that specify in-place function replacement if you are generating C or C++ code from MATLAB code directly or from a MATLAB Function block. In-place code replacement is an optimization technique that uses a single buffer, that is, the same memory, to store function input and output data, as in `x=foo(x)`.

For more information, see [Specify In-Place Code Replacement](#) and `coder.replace`.

Lookup table code replacement for Simulink

In R2014a, you can replace these lookup table functions during code generation for Simulink models.

<code>interp1D</code>	<code>interp4D</code>	<code>lookup2D</code>	<code>lookup5D</code>
<code>interp2D</code>	<code>interp5D</code>	<code>lookup3D</code>	<code>lookupND_Direct</code>
<code>interp3D</code>	<code>lookup1D</code>	<code>lookup4D</code>	<code>prelookup</code>

When you create a replacement table entry for one these functions, you must specify a set of algorithm properties in addition to the usual code replacement function key, conceptual arguments, and other applicable math mode information. Specify the algorithm properties by using new algorithm property fields in the code replacement tool or the new `addAlgorithmProperty` function. Conceptual arguments and algorithm parameters must match for replacement to occur.

For more information, see [Map Lookup Table Functions to Application Implementations](#).

Global variable usage available in the static code metrics report

The static code metrics report displays maximum reads and writes within a function and total reads and writes for each global variable and each member in a global variable data structure.

This information helps you to analyze the benefits of different global variable optimization choices. You can also compare the generated code across different versions.

For more information, see [Generate Static Code Metrics Report for Simulink Model](#).

Single-line (//) comment style available for generated code

In earlier releases, C and C++ code generation used a multi-line (`/* . . . */`) comment style. Beginning in R2014a, you can select a single-line (`// . . .`) comment style for generated code using the command-line parameter `CommentStyle`. For example, the following command sets the comment style to single-line comments:

```
>> set_param('rtwdemo_counter','CommentStyle','Single-line')
```

Here is an example of code generated using the single-line comment style:

```
// Sum: '<Root>/Sum' incorporates:
// Constant: '<Root>/INC'
```

```
// UnitDelay: '<Root>/X'
rtb_sum_out = (uint8_T)(1U + rtwdemo_counter_DW.X);
```

Note

- Single-line style comments and the `CommentStyle` parameter are supported only for ERT-based targets. Comment style for GRT targets is unchanged in R2014a.
 - For C, select single-line comments only if your compiler supports them.
-

For more information, see [Specify Comment Style](#).

Code indentation support for namespace declarations in generated code

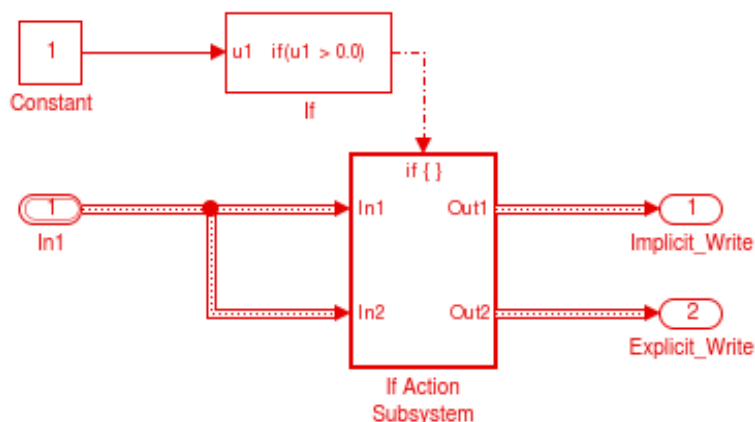
Previously, when specifying a namespace for a model class, the generated namespace code might be incorrectly indented if you selected K&R for the **Indent style** on the **Code Generation > Code Style** pane. In R2014a, the generated namespace code follows coding standards when you select the K&R style.

AUTOSAR C code generation enhancements

R2014a provides enhancements to AUTOSAR C code generation for AUTOSAR RTE-level data access APIs that improve efficiency and traceability of the generated C code. The changes include:

- Optimized generation of conditionally executed AUTOSAR explicit writes. A runnable can control whether an explicit RTE API call sends data element values.
- Additional traceability information in comments.
- More efficient expression folding and buffer reuse.

For example, in the following model, a constant value controls whether the software executes an explicit write.



In the C code generated for the step function, an explicit send (shown in **bold**) now appears inside conditional statements.

```

void Runnable_Step(void)
{
    if (mRelease_Conditional_P.Constant_Value > 0.0)
    {
        mRelease_Conditional_B.In1 =
            *Rte_IRead_Runnable_Step_RPorts_iIn1();

        Rte_Write_PPports_eOut2(
            Rte_IRead_Runnable_Step_RPorts_iIn1());
    }
    /* Outport: '<Root>/Implicit_Write' */
    Rte_IWrite_Runnable_Step_PPports_iOut1(
        &mRelease_Conditional_B.In1);
}

```

Static main program module for C++ class code generation

Beginning in R2014a, code generation supports use of a static main program module with C++ class code generated from a model. Previously, with ERT-based C++ encapsulation, code generation created an example main program and did not support use of a static main program.

In most cases, the easiest strategy for deploying generated C++ class code as a standalone program is to use the **Generate an example main program** option to generate the `ert_main.cpp` module. However, if you turn the **Generate an example main program** option off, you can use the module `matlabroot/rtw/c/src/common/rt_cppclass_main.cpp` as an example or template for developing your embedded applications. The module is not part of the generated code; it is provided as a basis for your custom modifications, and for use in simulation. For more information about using a static main program, see [Static Main Program Module](#).

Error message for data type replacement and classic call interface conflict

The model configuration options **Replace data type names in the generated code** (`EnableUserReplacementTypes`) and **Classic call interface** (`GRTInterface`) are mutually incompatible. Beginning in R2014a, if both model options are set to on, the model build generates an error message identifying the conflict. You must turn off one of the options.

In previous releases, if both options were set in a model reference hierarchy, build error messages did not precisely identify the conflict. The model build flagged a conflict between top and referenced models, without identifying the mutually incompatible options as the cause.

Compatibility Considerations

Beginning in R2014a, a conflict between **Replace data type names in the generated code** and **Classic call interface** is flagged with an error. You must turn off one of the options. If you have a model reference hierarchy and your intention is to use data type replacement, turn off **Classic call interface**. Make sure data type replacement settings match throughout the hierarchy.

Deployment

ARM Cortex-A optimized code generation using Ne10 library

You can replace generic code with Ne10-optimized code based on the ARM Neon general-purpose SIMD engine.

To use this code replacement library with the QEMU emulator for ARM Cortex-A processors, or with the Xilinx Zynq-7000 platform:

- 1 Install the Embedded Coder Support Package for ARM Cortex-A Processors, as described in [Install Support for ARM Cortex-A Processors](#).
- 2 Enable the code replacement library, as described in [Optimize Code for ARM Cortex-A Processors](#).

For more information, see:

- [Support Package for ARM Cortex-A Processors](#)
- [Support Package for Xilinx Zynq-7000 Platform](#)

Lookup table code replacement for Simulink

In R2014a, you can replace these lookup table functions during code generation for Simulink models.

interp1D	interp4D	lookup2D	lookup5D
interp2D	interp5D	lookup3D	lookupND_Direct
interp3D	lookup1D	lookup4D	prelookup

When you create a replacement table entry for one these functions, you must specify a set of algorithm properties in addition to the usual code replacement function key, conceptual arguments, and other applicable math mode information. Specify the algorithm properties by using new algorithm property fields in the code replacement tool or the new `addAlgorithmProperty` function. Conceptual arguments and algorithm parameters must match for replacement to occur.

For more information, see [Map Lookup Table Functions to Application Implementations](#).

Replacement of functions that take vector and matrix arguments

In R2014a, for Simulink Coder, you can specify code replacement conceptual arguments as vectors or matrices for these functions if the functions are generated from corresponding Simulink blocks.

abs	atanh	log	rSqrt	sincos
acosh	cos	log10	saturate	sinh
asinh	cosh	mod	sign	sqrt
atan	exp	pow	signPow	tan
atan2	hypot	rem	sin	tanh

When creating table entries for these functions, consider specifying mapping information, such as algorithm parameters and implementation attributes (for example, saturation and rounding). The

additional detail helps drive expected replacement behavior. For example, data types that you observe in a model might not match what the code generator uses as intermediate data types in an operation. To verify expected function replacement, inspect the generated code.

For more information, see [Map Math Functions to Application-Specific Implementations](#).

Logical data type support for arguments of replaced functions

In R2014a, when creating function arguments for inclusion in code replacement table entries, you can specify `logical` for the argument data type, which is equivalent to specifying `boolean`.

For more information, see [Manage Code Replacement Tables with the Code Replacement Tool and the `getTflArgFromString` function](#).

Code replacement data alignment for complex types

The code generator now supports code replacement data alignment of complex types.

For more information, see [Configure Data Alignment for Function Implementations and `addComplexTypeAlignment`](#).

Intel IPP (ANSI) and Intel IPP (ISO) code replacement libraries are combined

Code replacement library selections `Intel IPP (ANSI)` and `Intel IPP (ISO)` are replaced with a single library option, `Intel IPP`.

For information about setting the code replacement library, see [Code replacement library](#).

Compatibility Considerations

To specify either ANSI or ISO, use the new `Standard math library (TargetLangStandard)` parameter.

See [Standard math library](#).

Support for Eclipse IDE will be removed

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

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

- Build an executable from generated code on the host computer, and then run it on Linux using BeagleBoard hardware or an ARM processor.
- Build an executable from generated code on Linux using the BeagleBoard hardware (`remoteBuild`).
- 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 `gcc_target` configuration in XMakefile.
- Use Linux Task block.

Compatibility Considerations

For BeagleBoard, you can run `supportPackageInstaller` and install Simulink Support Package for BeagleBoard Hardware. For more information, see [BeagleBoard Hardware](#).

Support for Green Hills MULTI IDE will be removed

Embedded Coder Support Package for Green Hills MULTI IDE will be removed in a future release.

Support package for ARM Cortex-A processors

You can use the Embedded Coder Support Package for ARM Cortex-A Processors to:

- Run executables on Linux using a QEMU emulator for ARM Cortex-A9 processors.
- Generate Ne10-optimized code based on the ARM Neon general-purpose SIMD engine.
- Tune parameters on, and monitor data from, an executable running on the QEMU (External mode).
- Verify numeric accuracy and profile execution times using processor-in-the-loop (PIL) on the QEMU.

To download and install this feature, perform the steps described in [Install Support for ARM Cortex-A Processors](#).

For more information, see:

- [Support Package for ARM Cortex-A Processors](#)
- [Support Package for Xilinx Zynq-7000 Platform](#)

Support package for Texas Instruments C6000 processors

You can automatically generate code from Simulink models and execute it on TI's C6000 processors.

This feature includes the Embedded Coder Support Package for Texas Instruments C6000 Processors block library, which contains the following block libraries:

- Avnet S3ADSP DM6437 (`avnet_s3adsp_dm6437`)
- C6416 DSK (`c6416dsklib`)
- C6455 EVM (`c6455evmlib`)
- C6713 DSK (`c6713dsklib`)
- C6747 EVM (`c6747evmlib`)
- DM642 EVM (`dm642evmlib`)
- DM6437 EVM (`dm6437evmlib`)
- DM648 EVM (`dm648evmlib`)
- DSP/BIOS (`dspbioslib`)

- Optimization — C28x DMC (c28xdmclib)
- Optimization — C64x DSP Library (tic64dsplib)
- Scheduling (c6000dspcorelib)
- Target Communication (targetcommplib)

To install this support package, perform the steps described in [Install Support for C6000 DSPs](#).

For more information, see [Support Package for Texas Instruments C6000 DSPs](#).

Compatibility Considerations

Previous versions of Embedded Coder software had built-in support for C6000 processors. The current version of Embedded Coder does not have built-in support for C6000 processors.

To get support for C6000 processors, install Embedded Coder Support Package for Texas Instruments C6000 Processors, as described in the preceding section.

Updates to support package for Texas Instruments C2000 processors

The updated Embedded Coder Support Package for Texas Instruments C2000 Processors:

- Adds support for Texas Instruments Piccolo F2805x processors.
- Adds an example that shows how to use Control Law Accelerator (CLA).

To install or update this support package, perform the steps described in [Install Support for C2000 Processors](#).

For more information, see [Support Package for Texas Instruments C2000 Processors](#)

Updates to support package for Xilinx Zynq-7000 platform

The updated Embedded Coder Support Package for Xilinx Zynq-7000 Platform:

- Adds support for Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit.
- Installs the Embedded Coder Support Package for ARM Cortex-A Processors.
- Enables use of the `ert.tlc` system target file.

To install or update this support package, perform the steps described in [Install Support for Xilinx Zynq-7000 Platform](#).

For more information, see:

- [Support Package for Xilinx Zynq-7000 Platform](#)
- [Support Package for ARM Cortex-A Processors](#)

Updates to support package for STMicroelectronics STM32F4 Discovery board

The updated Embedded Coder Support Package for STMicroelectronics STM32F4-Discovery Board:

- Adds Memory Copy block, which enables you to read from and write to memory locations on the Discovery board.
- Adds a Mic in block, which enables you to read audio data from the MEMS microphone on the Discovery board.
- Adds a Audio out block, which sends the processed audio samples to the audio output connector on the Discovery board.
- Adds support for multitasking. This means that sub-rates can finish executing after the next base rate period begins. For example, by giving sub-rates more execution time, multitasking enables audio algorithms to process larger audio buffers.

To install or update this support package, perform the steps described in [Install Support for STMicroelectronics STM32F4 Discovery Board](#).

For more information, see [Support Package for STMicroelectronics STM32F4 Discovery Board](#)

Wind River Tornado (VxWorks 5.x) example main program option to be removed in future release

Using the **Templates** pane of the Configuration Parameters dialog box, you can configure an ERT-based model to generate an example main program for the Wind River Tornado® (VxWorks 5.x) RTOS. This capability will be removed from Embedded Coder software in a future release. If you generate code with the **Templates** pane parameter **Target operating system** set to `VxWorksExample`, the software displays a warning about future removal of the VxWorks 5.x example option.

Compatibility Considerations

In place of VxWorks 5.x support, consider using 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 [Support Package for Wind River VxWorks RTOS](#).

Performance

Additional options for reuse of global variables

In R2014a, on the **Optimization** pane, under **Signals and Parameters**, when you select Reuse global block outputs, the code generator reuses global variables for block outputs.

For more information, see Reuse Block Outputs in the Generated Code.

Enhanced global variable optimization options

In R2014a, you can choose a global variable reference optimization for the generated code.

In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, the Optimize global data access drop-down list provides the following options:

- None
Use default optimizations.
- Use global to hold temporary results
Maximize use of global variables.
- Minimize global data access
Minimize use of global variables by using local variables to hold intermediate values.

With an Embedded Coder license, if you select an embedded target such as `ert.tlc`, the software replaces the **Minimize data copies between local and global variables** check box with the **Optimize global data access** list. When **Minimize data copies between local and global variables** is selected, **Optimize global data access** is set to Use global to hold temporary results.

For more information, see Optimize Global Variable Usage.

for loops used to initialize arrays to zero

For signals with custom storage, code generation creates a `for` loop to initialize an array with matching values, such as all zeroes or ones. This initialization method reduces code size, especially for larger arrays. Previously, the generated code initialized each element individually on a separate line.

Verification

Software-in-the-loop simulation for physical models

You can run software-in-the-loop (SIL) simulations of models that use Simscape™ blocks.

SIL verification for subsystem code generation

You can use the SIL block approach to verify code generated from top-models and subsystems. In R2014a, SIL block verification supports the following features:

- Profiling of task and function execution times.
- Source-level debugging with the Microsoft Visual C++® debugger.

Compatibility Considerations

The table describes SIL block verification features that differ from the previous release. If you want to revert to previous SIL block behavior, in the Command Window, run:

```
silblocktype('legacy');
```

To restore R2014a SIL block behavior, run:

```
silblocktype('unified');
```

Feature	R2014a Details
Validation checks	<p>The software performs, with reference to your host computer architecture, stricter checks on active Hardware Implementation settings. If the software detects mismatches, the software generates errors.</p> <p>For example, if your host computer is a 64-bit Linux machine, you cannot specify the following combination of settings:</p> <ul style="list-style-type: none"> • Device vendor: Generic • Device type: 32-bit x86 compatible <p>To resolve the mismatch errors, do one of the following:</p> <ul style="list-style-type: none"> • In the Configuration Parameters > Code Generation > Verification pane, select the Enable portable word sizes check box. • In the Configuration Parameters > Hardware Implementation pane, through the Production hardware or Test hardware section, specify settings that correspond to your host computer architecture. <hr/> <p>The software generates an error if:</p> <ul style="list-style-type: none"> • The generated code for the component under test has been updated since the creation of the SIL block. • The MATLAB version has changed since the creation of the SIL block.

Feature	R2014a Details
GenerateErtSFunction parameter	<p>The GenerateErtSFunction parameter has the following command-line behavior:</p> <ul style="list-style-type: none"> • <code>set_param(model, 'GenerateErtSFunction', 'on')</code> generates a warning that the parameter will be removed in a future release. As a replacement, use the command <code>set_param(model, 'CreateSILPILBlock', 'SIL')</code>. • <code>set_param(model, 'GenerateErtSFunction', 'off')</code> does not change the parameter. As a replacement, use the command <code>set_param(model, 'CreateSILPILBlock', 'None')</code>. • <code>get_param(model, 'GenerateErtSFunction')</code> returns the value off. As a replacement, use the command <code>get_param(model, 'CreateSILPILBlock')</code>.
Parameter tuning	<p>During a SIL block simulation, the software does not support the tuning of block dialog box parameters. Through the SIL block dialog box, you can view the list of tunable global parameters</p> <p>The software does not support SIL block creation if all of the following apply:</p> <ul style="list-style-type: none"> • Code Generation > Interface > Code interface packaging is set to Reusable function. • Optimization > Signals and Parameters > Inline parameters is not selected. • The model contains parameters with storage class Auto or SimulinkGlobal.
Data definition and initialization	<p>In the SIL test harness, for signals that are internal with respect to the SIL block or models referenced by the SIL block, the software does not automatically define and initialize signals with imported storage classes.</p> <p>The software does not support automatic data definition and data transfer for local data stores in the SIL block.</p>
C++ class code (previously called C++ encapsulated code)	<p>For C++ class code:</p> <ul style="list-style-type: none"> • You must set External I/O access parameter to None. • Parameters are not tunable if Block parameter visibility is private and Block parameter access is either Method or Inlined method. <p>You can specify these settings through the Code Generation > Interface pane.</p>
Code generation report	<p>The code generation report does not display test harness files for your SIL block.</p>
Multiword fixed-point data	<p>At the SIL block interface, the software does not support multiword fixed-point data types. The software supports:</p> <ul style="list-style-type: none"> • At the block interface, single word data types that are wider than 32 bits. • Within the SIL block, multiword fixed-point data types.

Feature	R2014a Details
Boolean data type replacement	At the SIL block boundary, the software does not support the replacement of the <code>boolean</code> data type by integers.
GetSet custom storage class	At the SIL block boundary, the software does not support vectors with the GetSet custom storage class.
Asynchronous sample times	SIL block verification does not support asynchronous sample times.
Variable-size signals	At the SIL block boundary, the software does not support variable-size signals.
AUTOSAR server operation	SIL block verification does not support AUTOSAR server operation components.

SIL and PIL support for fixed-point data type override

At the SIL or PIL component boundary, the software supports signals with data types that are overridden by the Fixed-Point Tool **Data type override** parameter.

SIL and PIL support for Invoke AUTOSAR Server Operation block

You can perform SIL and PIL verification of code generated from models that have Invoke AUTOSAR Server Operation blocks.

SIL and PIL support for structure parameters with storage class SimulinkGlobal

The software supports the tuning of structure parameters with the `SimulinkGlobal` storage class for the following types of simulation:

- Top-model SIL and PIL
- SIL and PIL block

Previously, this feature was supported for only Model block SIL and PIL.

Model block SIL and PIL with export-function and asynchronous function-call models

In R2014a, you can use Model block SIL and PIL simulations to verify code generated from:

- Export-function models.
- Models with asynchronous function-call inputs, that is, models that use Asynchronous Task Specification blocks.

In addition to verification, you can:

- Perform source-level debugging.
- Generate execution time profiles.
- Observe code coverage.

Model block SIL and PIL does not support models with Asynchronous Task Specification blocks if the models also have blocks that use absolute time.

Model block SIL and PIL with disabled inline parameters

Model block SIL and PIL verification supports R2014a behavior of the `InlineParams` parameter with value `off`. For more information, see Simplified tuning of all parameters in referenced models.

Compatibility Considerations

Consider the following simulation settings for a top model with a Model block (referenced model):

- Top-model **Simulation > Mode**: Normal
- Model block **Simulation mode**: Software-in-the-loop (SIL) or Processor-in-the-loop (PIL)
- Referenced model **Optimization > Signals and Parameters > Inline parameters (InlineParams)**: Not selected (`off`)

Previously, when executing the Model block in SIL or PIL mode, the software overrode the `off` value of `InlineParams` and used the `on` value. The override action does not occur in R2014a. As a result, the tuning behavior for parameters with the `Auto` storage class is the same as the tuning behavior for parameters with the `SimulinkGlobal` storage class. For more information, see Tunable Parameters and SIL/PIL.

To revert to previous behavior, you must manually set `InlineParams` to `on`.

SIL and PIL block improvements

In Accelerator mode, you can run a simulation with a top model that has SIL or PIL blocks. Therefore, you can speed up simulation of your model components that are not SIL or PIL blocks.

The following features are supported for PIL block verification:

- Use of Goto and From blocks across the PIL block boundary.
- Use of virtual buses without bus objects across the PIL block boundary.
- Export of functions from triggered subsystems.

Previously, these features were supported for only SIL block verification.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.5

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Software-in-the-loop verification for MATLAB Coder

Use software-in-the-loop (SIL) execution to verify production-ready source code. SIL execution involves compiling and running static library code on your host computer. Through SIL execution, you can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of static library code.

Previously, verification was restricted to code generated for execution only within MATLAB. Now, in MATLAB, you can compile and run standalone code on the host computer through a MATLAB SIL interface.

You can run a SIL execution:

- Using the MATLAB Coder project interface. See [Software-in-the-Loop \(SIL\) Execution Through the Project Interface](#).
- From the command line. See [Software-in-the-Loop \(SIL\) Execution From the Command Line](#).

Custom generated identifiers for emxArray utility functions

You can customize generated identifiers for `emxArray` (embeddable `mxAarray`) utility functions. When you generate code that uses variable-size data, the code generation software exports utility functions to interact with `emxArray` data structures. Customize utility function identifiers to avoid name collisions when a function that uses variable-size data calls a library function that uses variable-size data.

To customize generated identifiers for `emxArray` utility functions:

- In a project

On the Project Settings dialog box **Code Appearance** tab, under **Identifier Format**, in the **EMX Array Utility Functions** field, enter the identifier format. For example, `'myemxMN'`.

- At the command line

Create a code generation configuration object and set the `CustomSymbolStrEMXArrayFcn` parameter to the identifier format. For example:

```
cfg = coder.config('lib');  
cfg.CustomSymbolStrEMXArrayFcn='myemx$M$N';
```

For details about the identifier format, see `coder.EmbeddedCodeConfig`.

Model Architecture and Design

Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior

R2013b enhances AUTOSAR modeling, component import, and programmatic control. See also “Support for AUTOSAR release 4.0.3 XML and generated code” on page 20-10.

Enhanced modeling and simulation of AUTOSAR multiple runnables

In previous releases, AUTOSAR multi-runnables were modeled as function-call subsystems within a wrapper subsystem in a Simulink model. To generate code, you right-clicked the wrapper subsystem and exported functions.

Beginning in R2013b, you can model AUTOSAR multi-runnables as function-call subsystems at the top level of a model, without having to use a wrapper subsystem. When you generate code for the model, each function-call subsystem representing a runnable appears in the model C code as a callable model entry-point function.

You can simulate multiple runnables in an AUTOSAR software component in multiple simulation modes. For example:

- For a periodic runnable, you can edit the properties of the function-call subsystem inport to set the sample time for a periodic event simulation.
- For a non-periodic runnable, you can edit the **Data Import/Export** pane of the Configuration Parameters dialog box to set up data loading for an asynchronous event simulation.

For more information, see Configure Multiple Runnables.

Enhanced ARXML import of AUTOSAR software component internal behavior

The AUTOSAR software component importer tool can automatically import the internal behavior of a multi-runnable AUTOSAR software component into a Simulink model. You can use the `createComponentAsModel` method of the class `arxml.importer` to specify that internal behavior be imported. For example:

```
>> obj = arxml.importer('mySWC.arxml');
>> obj.createComponentAsModel('/pkg/swc', 'CreateInternalBehavior', true)
```

The importer:

- Adds subsystem blocks in the model and maps them to corresponding runnables imported from the AUTOSAR software component.
- Adds signal lines in the model and maps them to corresponding inter-runnable variables (IRVs) imported from the AUTOSAR software component.

Ability to model AUTOSAR mode receiver ports and events

R2013b provides the ability to model AUTOSAR mode receiver ports and mode-switch events in Simulink. Specifically, you can:

- Model the mode receiver port for an AUTOSAR software component using a Simulink inport.
- Specify a mode-switch event to trigger an initialize function runnable or an exported function-call subsystem runnable.

For more information, see [Configure AUTOSAR Mode Receiver Ports and Events](#).

AUTOSAR dual-scaled parameter

The new `AUTOSAR.DualScaledParameter` class extends the capabilities of the `AUTOSAR.Parameter` class. You can define a parameter object that stores two scaled values of the same physical value. Suppose you want to store temperature measurements as Fahrenheit or Celsius values. You can define a parameter that stores the temperature in either measurement scale with a computational method to convert between the dual-scaled values.

You can use `AUTOSAR.DualScaledParameter` objects in your model for both simulation and code generation. The parameter computes the internal value before simulation or code generation via a computational method, which can be a first-order rational function. This offline computation results in leaner generated code.

Embedded Coder also generates an XML file for use by a calibration tool. This file contains the dual-scaled values and the corresponding computational method.

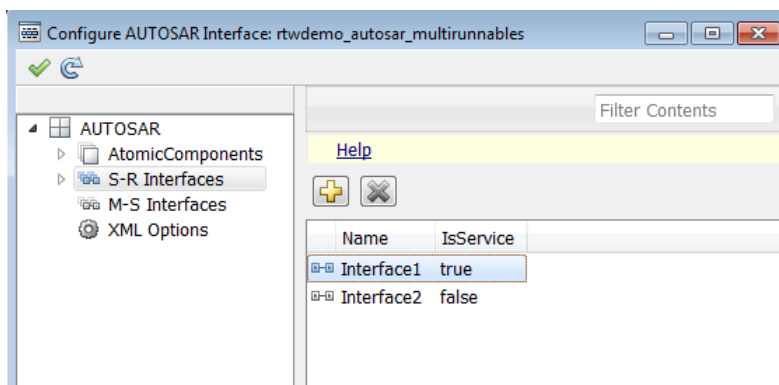
For more information, see [AUTOSAR.DualScaledParameter](#).

Programmatic interface for configuring AUTOSAR properties and Simulink-AUTOSAR mapping

R2013b provides a programmatic interface for configuring AUTOSAR properties and Simulink mapping information using MATLAB functions. You can programmatically get, set, add, and remove the same component properties and mapping information displayed in the **AUTOSAR Properties Explorer** and **Simulink-AUTOSAR Mapping Explorer** views of the [Configure AUTOSAR Interface](#) dialog box.

In the function syntax, you can use fully or partially qualified names to locate properties. For example, the following code sets the `IsService` property for the sender-receiver interface located at path `Interface1` in the example model `rtwdemo_autosar_multirunnables` to `true`. In this case, specifying the name `Interface1` is enough to locate the property.

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> set(propObj, 'Interface1', 'IsService', true);
```



If you added a sender-receiver interface to the component, you would specify a fully qualified path, for example:

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> addSRInterface(propObj, '/pkg/if/Interface3', 'IsService', true);
```

The new AUTOSAR configuration functions also validate syntax and semantics for requested AUTOSAR property and mapping changes.

Reorganization of Model Advisor Embedded Coder checks

Checks previously provided with Simulink in the Model Advisor Embedded Coder folder are now available with either Simulink Coder or Embedded Coder. For a list of checks available with each product, see:

- Simulink Coder Checks
- Embedded Coder Checks

Model Advisor fixed-point checks with additional coverage and optimization awareness

The Model Advisor fixed-point checks now cover blocks in base Simulink and System Toolboxes, the MATLAB Function block, System objects, Stateflow, and `fi` objects. These improved checks take into consideration model settings such as hardware configuration and code generation settings. These updated checks also avoid false negative results.

For more information, see:

- Identify blocks that generate expensive rounding code
- Identify questionable fixed-point operations
- Identify blocks that generate expensive fixed-point and saturation code

Protected model Web view

In R2013b, a read-only Web view of protected models is now available.

To include the Web view in the protected model, right-click the model reference block, and then select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**. Select the **Open read-only view of model** check box and click **Create**.

To enter a password, right-click the protected model shield icon and select **Authorize**. Enter the password and click **OK**. To show the Web view for a protected model, right-click the shield icon of the protected model and select **Show Web view**.

RTW.AutosarInterface class to be removed in a future release

In R2013b, a new programmatic interface for configuring AUTOSAR properties and mapping information for a Simulink model has replaced the `RTW.AutosarInterface` class used in earlier releases. The `RTW.AutosarInterface` class will be removed in a future release.

Compatibility Considerations

If you are using the `RTW.AutosarInterface` class and methods to programmatically control and validate the AUTOSAR configuration of a model, you should migrate to using the new AUTOSAR property and mapping functions listed in AUTOSAR Component Development. The new functions are designed to work with the component properties and mapping information displayed in the

AUTOSAR Properties Explorer and **Simulink-AUTOSAR Mapping** Explorer views of the Configure AUTOSAR Interface dialog box.

Subsystem methods of `arxml.importer` class to be removed in a future release

Beginning in R2013b, you can model AUTOSAR multi-runnables as function-call subsystems at the top level of a model, rather than as function-call subsystems within a wrapper subsystem that represents the AUTOSAR software component. The following methods of the `arxml.importer` class will be removed in a future release:

- `arxml.importer.createComponentAsSubsystem` — Create AUTOSAR atomic software component as Simulink atomic subsystem
- `arxml.importer.createOperationAsConfigurableSubsystems` — Create configurable Simulink subsystem library for client-server operation

Compatibility Considerations

If you are using `createComponentAsSubsystem` or `createOperationAsConfigurableSubsystems`, you should migrate to using the top model oriented approach described in [Configure Multiple Runnables](#).

Data, Function, and File Definition

Simplified global types file `rtwtypes.h` with invariant content

Previously, during rebuilds of a model hierarchy, the code generation process might have updated the content of the shared header file `rtwtypes.h`. If a model in the hierarchy changed, or the code generator detected a new model in the hierarchy, `rtwtypes.h` could be overwritten. When `rtwtypes.h` changes, you must recompile the code.

In R2013b, the code generator separates some of the `rtwtypes.h` content into separate header files that are generated only when certain model settings or components are present. Separate header files are generated, however, `rtwtypes.h` is unchanged. When certain model settings or components are present, the code generator creates the following new header files.

Model setting or component	Content generated to header file
Multiword data types	<code>multiword_types.h</code>
Model reference target	<code>model_reference_types.h</code>
Model reference blocks	<code>model_reference_types.h</code>
MAT-file logging is selected	<code>builtin_typeid_types.h</code> <code>multiword_types.h</code>
C API	<code>builtin_typeid_types.h</code>
Interface is set to External mode	<code>multiword_types.h</code>

For more information on files created during code generation, see [Files Created During the Build Process](#).

C++ encapsulation support for name space control and template-based file customization

Name space control for scoping C++ encapsulated model classes

R2013b adds name space control for scoping model classes generated using C++ encapsulation. You can use the **Namespace** parameter in the **Configure C++ Encapsulation Interface** dialog box to specify a name space for a model class. If specified, the name space is emitted in the generated code for the model class. To scope the C++ encapsulated model classes in a model reference hierarchy, you can specify a different name space for each referenced model. For more information, see [Use Name Spaces to Scope C++ Encapsulated Model Classes](#).

For more information on configuring C++ encapsulated model classes, see [Configure C++ Encapsulation Interfaces Using Graphical Interfaces](#).

Template-based customization of encapsulated C++ header and source files

Embedded Coder now supports the **Code Generation > Templates** pane of the **Configuration Parameters** dialog box for models that use C++ encapsulation. You can use the code and data templates to control the appearance of C++ code in generated model header and source files. For example, you can customize file and function banners to meet organization standards.

However, the following template file features that are supported for other language selections are not supported for C++ encapsulation:

- Free-form text outside template sections
- Custom tokens
- TLC commands (<! > tokens)

Shared utility naming control

You can customize a shared utility name. On the **Code Generation > Symbols** pane enter text and formatting characters in the **Shared utilities** box.

The default token string is \$N\$C.

Token	Description
\$N	The code generator inserts the shared utility function name.
\$C	When the combined text and utility name exceed the maximum identifier length, the code generator inserts an eight-character conditional checksum. This checksum ensures that the name is unique.

If the shared utility identifier exceeds the maximum length, characters are deleted from \$N and the eight-character conditional checksum is inserted.

For more information see

- Shared utilities
- Identifier Format Control
- Exceptions to Identifier Formatting Conventions

Expanded support for identifier names

When specifying temporary local variables, you can now use \$A to insert the data type acronym into your variable name. This capability provides you with a more consistent naming scheme.

- You can include \$A in naming for local temporary variables where previously it was supported only for local block output variables and field names of global types. For more information, see Identifier Format Control, Local temporary variables and Field name of global types.
- You can customize identifier names by specifying \$A which maps to the data type replacement setting. Previously the generated code changed the types, but not the identifier names. For more information, see Data Type Replacement.

Terminate function setting honored for subsystems and referenced models

In previous releases, model builds did not uniformly honor the setting of the model option **Terminate function required** when generating code for subsystems or referenced models. A model build could

generate termination code for a subsystem or referenced model when **Terminate function required** was cleared.

Beginning in R2013b, model builds honor the setting of **Terminate function required** for subsystems and referenced models. When **Terminate function required** is cleared, the build suppresses subsystem and referenced model termination code.

Compatibility Considerations

If an existing model relies on subsystem or referenced model termination code being generated despite the model option **Terminate function required** being cleared, consider turning on the **Terminate function required** option.

Code Generation

Support for AUTOSAR release 4.0.3 XML and generated code

R2013b adds AUTOSAR release 4.0.3 support, as follows:

- ARXML import and export support AUTOSAR release 4.0.3 XML files.
- The AUTOSAR target generates AUTOSAR release 4.0.3 compliant C code.
- Selecting the value 4.0 for the AUTOSAR model parameter Generate XML file from schema version now selects schema revision 4.0.3, rather than 4.0.2. Also, the parameter now defaults to value 4.0, rather than 3.0 or an earlier version.

See also “Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior” on page 20-3.

Indent style and size control for code generation

R2013b adds options for customizing code appearance. The following new parameters are located in the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane.

- **Indent style:** Specify K&R or Allman style for the placement of braces.
- **Indent size:** Specify the number of characters per indent level. Choose from 2–8 characters.

For more information on configuring code style parameters, see Control Code Style.

Subsystem functions return value in generated code

In the Subsystem Block Parameters dialog box, on the **Code Generation** tab, if you set

- The **Function packaging** parameter for your subsystem to `Nonreusable function`
- The **Function interface** parameter to `Allow arguments`

The code generator might generate a subsystem function that returns a scalar output value. Previously, subsystem functions returned `void`.

Model reference step function void input and output arguments

Since R2010a, when a reusable subsystem fed the output, code generation might create output arguments for model reference step functions.

In R2013b, code generation produces model reference step functions with void input and void output when the model reference block:

- Is a single instance.
- Has exported globals on its input and output ports.

Deployment

ARM Cortex-M optimized code with STM32F4-Discovery board example

Build ARM Cortex-M optimized executables from Simulink models. Automatically run executables on STMicroelectronics STM32F4-Discovery boards.

Note In addition to the basic math optimizations provided by Embedded Coder Support Package for ARM Cortex-M Processors, you can obtain advanced optimizations for ARM DSP filters using the *DSP System Toolbox Support Package for ARM Cortex Processors*. For more information, see the DSP System Toolbox release notes for R2013b.

Support package for ARM Cortex processors

Use the Embedded Coder Support Package for ARM Cortex-M Processors to:

- Build and run CMSIS-optimized executables on ARM Cortex-M QEMU emulator.
- Use the capabilities and features described in Supported Features for ARM Cortex-M Processors

To download and install this feature, perform the steps described in Install Support for ARM Cortex-M Processors.

For more information, see the Support Package for ARM Cortex-M Processors topic.

Support package for STMicroelectronics STM32F4-Discovery Board

Use the Embedded Coder Support Package for STMicroelectronics STM32F4-Discovery Board to automatically build (makefile-based), download, and run an executable on Discovery board processors.

Use blocks from the Embedded Coder Support Package for STMicroelectronics STM32F4-Discovery Board block library:

- ADC — Convert analog signal to digital signal.
- GPIO Read — Configure input pin to read pin status.
- GPIO Write — Configure output pin to output pin status.

This support package automatically installs the following third-party software:

- STM32F4DISCOVERY peripheral firmware examples <http://www.st.com/internet/evalboard/product/252419.jsp>
- OpenOCD <http://www.freddiechopin.pl/en/download/category/4-openocd>
- GNU Tools for ARM Embedded Processors <https://launchpad.net/gcc-arm-embedded>
- QEMU <http://lassauge.free.fr/qemu/>
- CMSIS <http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php>

To download and install this support package, perform the steps described in Install Support for STMicroelectronics STM32F4 Discovery Board.

For more information, see the Support Package for STMicroelectronics STM32F4 Discovery Board topic.

Wind River VxWorks 6.9 support

You can automatically generate code from Simulink models and execute it on VxWorks 6.9 RTOS.

To use this feature, install the corresponding support package:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`.
- 2 Use Support Package Installer to install the Embedded Coder Support Package for Wind River VxWorks RTOS.

This feature includes the Embedded Coder Support Package for Wind River VxWorks RTOS block library, which contains the following blocks:

- UDP Send and UDP Receive — Enable UDP communication with networked devices using an Ethernet port.
- VxWorks Task — Spawn task function as a separate VxWorks thread.

For more information, see the Support Package for Wind River VxWorks RTOS topic.

Compatibility Considerations

Previous versions of Embedded Coder software had built-in support for the VxWorks 6.7 and 6.8. The current version of Embedded Coder does not have built-in support for VxWorks 6.7 and 6.8. To get support for VxWorks 6.7, 6.8, and 6.9, install the Embedded Coder Support Package for Wind River VxWorks RTOS.

Support package for Texas Instruments C2000 processors

You can automatically generate code from Simulink models and execute it on Texas Instruments C2000 processors.

To use this feature, install the corresponding support package:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`.
- 2 Use Support Package Installer to install Embedded Coder Support Package for Texas Instruments C2000 Processors.

This feature includes the Embedded Coder Support Package for Texas Instruments C2000 Processors block library, which contains:

- C2802x (c2802xlib) block library
- C2803x (c2803xlib) block library
- C2806x (c2806xlib) block library
- C280x (c280xlib) block library
- C281x (c281xlib) block library
- C2834x (c2834xlib) block library
- C28x3x (c2833xlib) block library

- Memory Operations block library
- Optimization — C28x DMC (c28xdmclib) block library
- Optimization — C28x IQmath (tiqmathlib) block library
- RTDX Instrumentation (rtDXBlocks) block library
- Scheduling block library
- Target Communication block library

For more information about this feature, see the Support Package for Texas Instruments C2000 Processors topic.

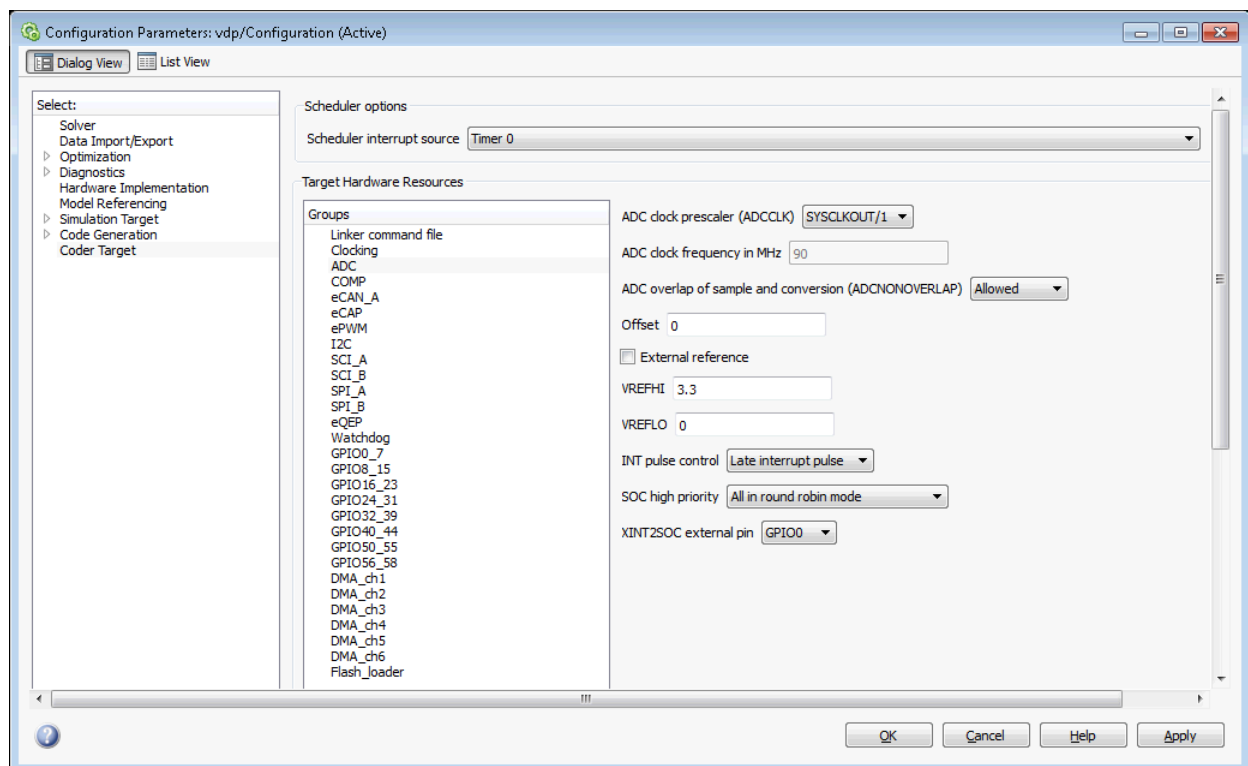
Compatibility Considerations

Previous versions of Embedded Coder software had built-in support for C2000 processors. The current version of Embedded Coder does not have built-in support for C2000 processors.

To get support for C2000 processors, install Embedded Coder Support Package for Texas Instruments C2000 Processors, as described in the preceding section.

Coder Target pane in Configuration Parameters dialog box

You can use the Coder Target pane to configure target hardware settings for your model.



This Coder Target pane has the same name as the Code Generation > Coder Target sub-pane that appears when the **System target file** parameter is `idelink_ert.tlc` or `idelink_grt.tlc`.

To use the Coder Target pane:

- 1 Open Configuration Parameter dialog box by entering **Ctrl+E**.
- 2 Select the Code Generation pane.
- 3 Set the **System target file** parameter to `ert.tlc`. Click **Apply**.
- 4 Set the **Target hardware** parameter to match your target hardware.

The Configuration Parameters dialog box displays the Coder Target pane with parameters for the specified target hardware.

ZedBoard hardware support

You can automatically generate code from Simulink models and execute it on ZedBoard™ hardware. Specifically, you can execute the code in the Linux environment on the ZedBoard's ARM Cortex-A9 processor.

To use this feature, install the corresponding support package:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`.
- 2 Use Support Package Installer to install Embedded Coder Support Package for Xilinx Zynq-7000 Platform.

This feature includes the Embedded Coder Support Package for Xilinx Zynq-7000 Platform block library, which contains:

- UDP Send and UDP Receive — Enable UDP communication with networked devices using an Ethernet port.
- Linux Task — Spawns task function as separate Linux thread.

For more information, see the Support Package for Xilinx Zynq-7000 Platform topic.

Note For more information about using HDL Coder™ software with the FPGA on the Avnet® ZedBoard hardware, see IP core integration into Xilinx EDK project for ZC702 and ZedBoard

Simplified multi-instance code interface and dynamic memory allocation for ERT targets

Embedded Coder now provides a simplified multi-instance code interface, with a dynamic memory allocation option, for ERT-based models. The new capabilities support easier integration of multi-instance code into applications. The new interface to generated model code features:

- Use of a single model entry-point function argument for instance data such as signals, states, parameters, and optionally root-level input and output.
- Configurable argument list for model root-level input and output.
- Option to generate a function that dynamically allocates memory for model instance data.

For more information, see model option **Generate reusable code**, Entry-Point Functions and Scheduling, and Generate Reentrant Code from a Top-Level Model.

For an example of an ERT-based model configured to generate reusable, reentrant code, see the example model `rtwdemo_reusable`.

Compatibility Considerations

Beginning in R2013b, when you select **Generate reusable code** for an ERT-based model, model data structures, such as Block I/O, DWork, and Parameters, are packaged into the real-time model data structure. The real-time model data structure is passed in a single argument to the model entry-point functions `model_initialize`, `model_step`, and `model_terminate`.

In earlier releases, when you selected **Generate reusable code** for an ERT-based model, model data structures were passed in separately as arguments to the model entry-point functions. The number of generated arguments varied, depending on the data requirements of the model.

If you have code that calls reusable code generated from ERT-based models, you should update the model entry-point function calls to use the new, simplified interface.

For example, consider model entry-point functions previously called as follows:

```
/* Step the model */
rtwdemo_reusable_step(&rtP, &rtDWork, rtU_In1, rtU_In2, &rtY_Out1);

/* Initialize model */
rtwdemo_reusable_initialize();
```

In R2013b or later, the corresponding calls might be as follows:

```
/* Step the model */
rtwdemo_reusable_step(rtM, rtU_In1, rtU_In2, &rtY_Out1);

/* Initialize model */
rtwdemo_reusable_initialize(rtM);
```

Beginning in R2013b, after selecting **Generate reusable code**, you also can select the model option **Generate function to allocate model data**, which generates a function to dynamically allocate memory (using `malloc`) for model data structures. If you do not select this option, the model instance data must be allocated either statically or dynamically by the calling code. For this case, an additional requirement beginning in R2013b is that pointers to the individual data structures (such as Block IO, DWork, and Parameters) must be set up in the top-level real-time model data structure.

Addition and Subtraction Operator Code Replacement Assumes Cast-Before-Operation Behavior

The type of algorithm that addition and subtraction operators apply for a given math library can be characterized as cast-before-operation (CBO) or cast-after-operation (CAO). In the CBO case, prior to performing the operation, the algorithm type casts input values to the output type. If the output data type cannot exactly represent the input values, losses can occur as a result of the cast to the output type. Additional loss can occur when the result of the operation is cast to the final output type.

In the CAO case, the algorithm computes the ideal result of the operation on the inputs and then type casts the result to the output data type. Loss occurs during the type cast. This algorithm behaves similarly to the C language except when the signedness of the operands does not match. For example, when you add a signed long operand to an unsigned long operand, standard C language rules convert the signed long operand to an unsigned long operand. The result is a value that is not ideal.

Starting in R2013b, the code generator assumes CBO behavior for replacement code defined for addition and subtraction operators.

Compatibility Considerations

In previous releases, the code replacement software did not take the Sum block configuration into account when making a replacement. Starting in R2013b, the code replacement software considers the Sum block for replacement if the block meets the CBO constraint. To meet that constraint, the block must be configured in one of the following ways:

- Input and output are the same type and the size of the accumulator type is equal to or greater than the size of that type
- Input and output types differ, but the size of the accumulator type is equal to the size of the output type

If the Sum block does not meet the CBO constraint, a replacement that occurred in a previous release might not occur.

Addition functions in libraries that implement full-precision addition, such as the ANSI C library, are not suitable as replacement functions.

When using code replacements, validate that the numerical results of the generated code match the results of a processor-in-the-loop (PIL) simulation.

Performance

Reusable custom storage class to reduce root I/O memory

In R2013b, if a pair of root-level model input and output signals uses the same storage class specification, code generation can reuse the root I/O signals in the generated code. The storage class specifications are the new custom storage class `Reusable(Custom)` or a custom storage class created from `Reusable(Custom)`. Reusing code for root input and output signals allows for further optimizations that reduce data copies, global variables, and ROM/RAM size. For more information, see [Signal Reuse for Root-Level Model Inputs and Outputs](#).

Subsystem functions reused independently of output connection

Previously, code generation used different criteria to determine when to reuse code.

- Code generation used the connection status to help determine the number of subsystem functions to generate.
- Code generation reused subsystem functions with varied connection status only when the outputs were passed by structure reference.

Code generation can now reuse subsystem functions regardless of:

- The connection state of the outputs. You can specify multiple outputs as unconnected or terminated across subsystems.
- Whether the reusable system outputs are passed as `Structure` reference or `Individual` arguments.

Verification

SIL and PIL support fixed-point data types wider than 32 bits

Use software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations to verify generated code that contains fixed-point data types wider than 32 bits.

A number of host and target platforms support 64-bit native data types. On these platforms, implementing a fixed-point data type wider than 32 bits as a single word is more efficient than the multiword fixed-point approach. Previously, data types wider than 32 bits, including multiword fixed-point, were supported internally within a SIL or PIL component. However, the data types were not supported in the communication between the MATLAB and Simulink host and the SIL or PIL component on the target. Now, the software supports 33-bit to 64-bit single word, fixed-point data types in host-target communication.

Data types that SIL and PIL support include the following:

- 64-bit `long` and `long long`
- 64-bit execution profiling timer data type — Previously, the target returned only the 32 least significant bits to the MATLAB host, with the possibility of timer wrapping.
- `int64` and `uint64` — Used in MATLAB Coder SIL execution.

The following constraints apply:

- For 64-bit data type support, the data type must be representable as `long` or `long long` on the MATLAB host *and* the target. Otherwise, the software uses the multiword fixed-point approach, which SIL and PIL do not support.
- 32-bit Windows does not support 64-bit `long` or `long long` data types. In this case, the software uses the multiword fixed-point approach which SIL and PIL do not support.
- The software does not support the 40-bit `long` data type of the TI's C6000 target.

Through the **Configuration > Hardware Implementation** pane, you can enable support for the 64-bit `long long` data type. However, for data types with widths between 33 and 40 bits (inclusive), the software implements the data types using the 40-bit `long` data type which SIL and PIL do not support.

SIL and PIL protected model support

Software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation modes are now supported for protected models. You can run models that contain protected models in SIL and PIL simulation modes if the protected models support code generation. In previous releases, the only supported simulation modes were Normal and Accelerator.

Code execution profiling improvements

Standalone code generation with function profiling

You can generate executable code (**Ctrl+B**) for your model even if function profiling is enabled. The software produces the following warning message:

Warning: Code profiling instrumentation is not supported for standalone builds (Ctrl+B). You can run the executable, but no profiling data will be collected.

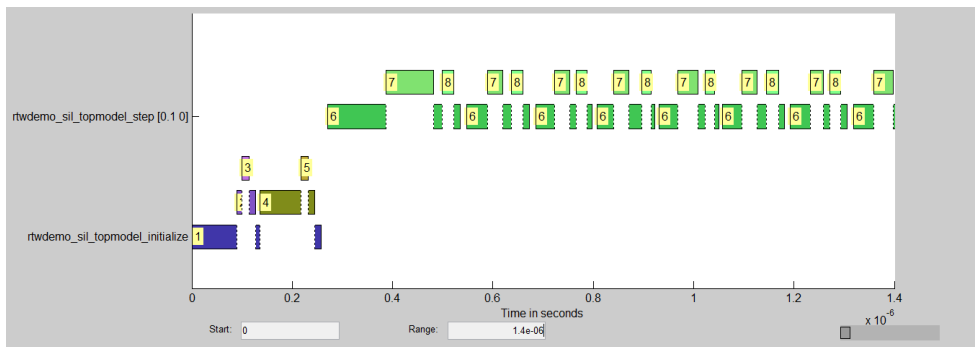
Previously, if function profiling was enabled for a SIL or PIL simulation, building your model produced an error message. For example:

Code profiling instrumentation within the generated code is not supported for top model standalone builds (Ctrl+B). You cannot build the top model `rtwdemo_sil_modelblock` in standalone mode because `rtwdemo_sil_modelblock` has code profiling instrumentation enabled. You must either simulate `rtwdemo_sil_modelblock` in SIL or PIL mode or disable code profiling instrumentation for `rtwdemo_sil_modelblock`. To disable code profiling instrumentation, clear the check box Simulation > Configuration Parameters > Code Generation > Verification > Measure function execution times.

For information about obtaining execution time profiles for generated code, see Code Execution Profiling.

Display of code section invocations

You can display code section invocations over the execution timeline.



For more information, see [timeline](#).

SampleOffset and SamplePeriod removed

The `coder.profile.ExecutionTimeSection` `SampleOffset` and `SamplePeriod` methods have been removed.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.4

New Features

Bug Fixes

Compatibility Considerations

Code Generation from MATLAB Code

Improved code replacement traceability for MATLAB code generation

In the R2013a release, there is now improved code replacement traceability for standalone code generated using MATLAB Coder. This capability is not available for generated MEX functions. When you choose to include code replacements in the code generation report:

- The code generation report includes a link to the Code Replacements Report.
- Code replacement trace information is generated for viewing in the **Trace Information** tab of the Code Replacement Viewer.
- The code replacement report lists replacement functions and their associated MATLAB code.

You can use the code replacement report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the MATLAB code that triggered the replacement.

For more information, see [Enable the Code Replacements Report and Viewing Code Replacements in the Generated Code](#).

Static code metrics report for MATLAB Coder

When you generate standalone C code with MATLAB Coder, the HTML code generation report now includes a static code metrics report. The static code metrics report is not available for generated MEX functions.

The static code metrics include the:

- Number of source code files.
- Number of lines of code.
- List of global variables.
- Functions in a call tree format.
- Estimated stack size required for a function.

You can use the information in the static code metrics report to:

- Find the number of files and lines of code in each file.
- Estimate the number of lines of code and stack usage per function.
- Compare how many files, functions, variables, and lines of code are generated every time you change the MATLAB algorithm.
- Determine a target platform and allocation of RAM to the stack, based on the size of global variables plus the estimated stack size.
- Determine possible performance slow points, such as the largest global variables or the most costly call path in terms of stack usage.
- View the cyclomatic complexity of a function, which counts the number of linearly independent paths through a function.
- View the function call tree.

- Determine the longest call path to estimate the worst-case execution timing.
- View how target functions, provided by the selected code replacement library, are used in the generated code.

For more information, see [Generate a Static Code Metrics Report for MATLAB Code and Static Code Metrics](#).

Model Architecture and Design

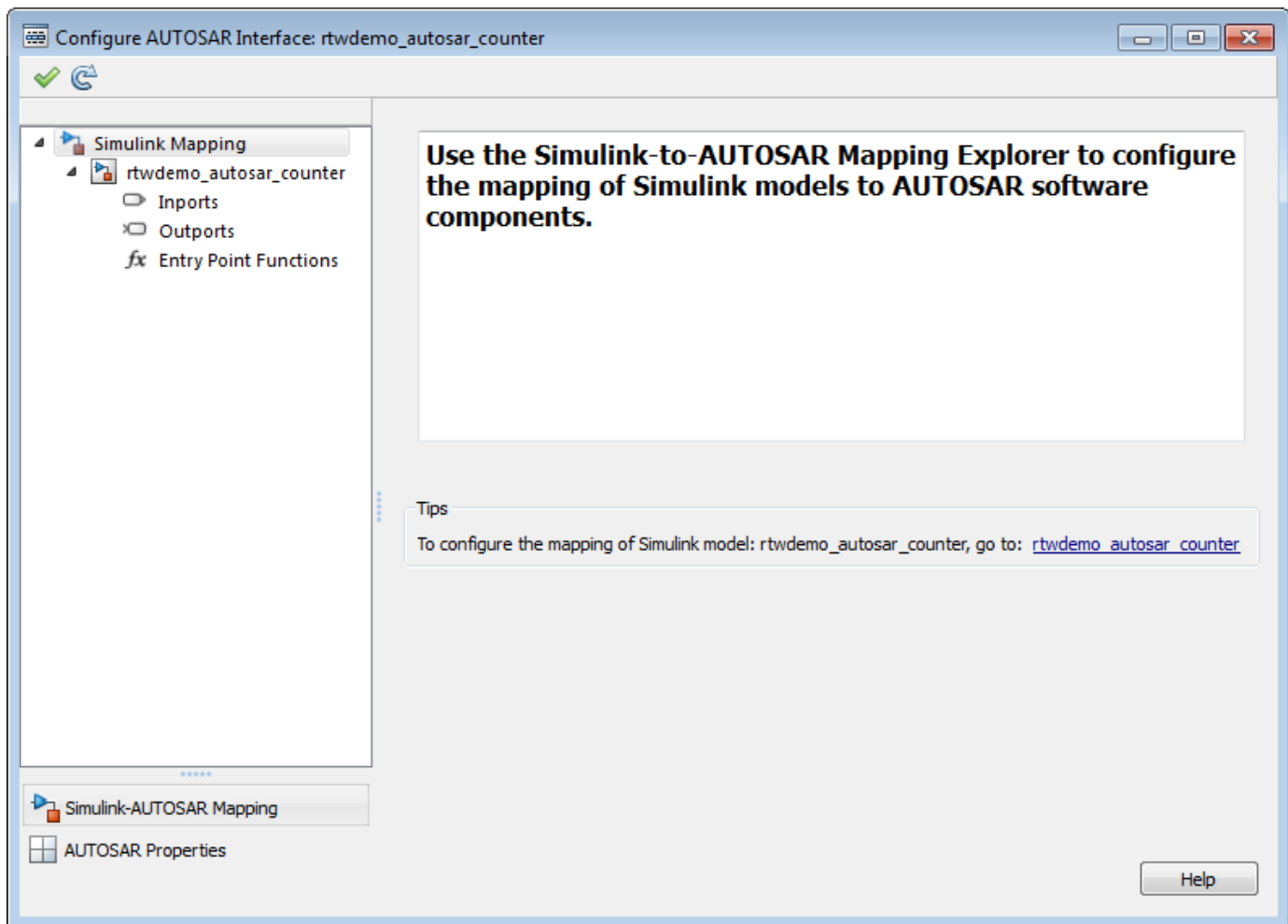
AUTOSAR user interface and round trip ARXML file import and export improvements

Improved graphical user interfaces for AUTOSAR configuration

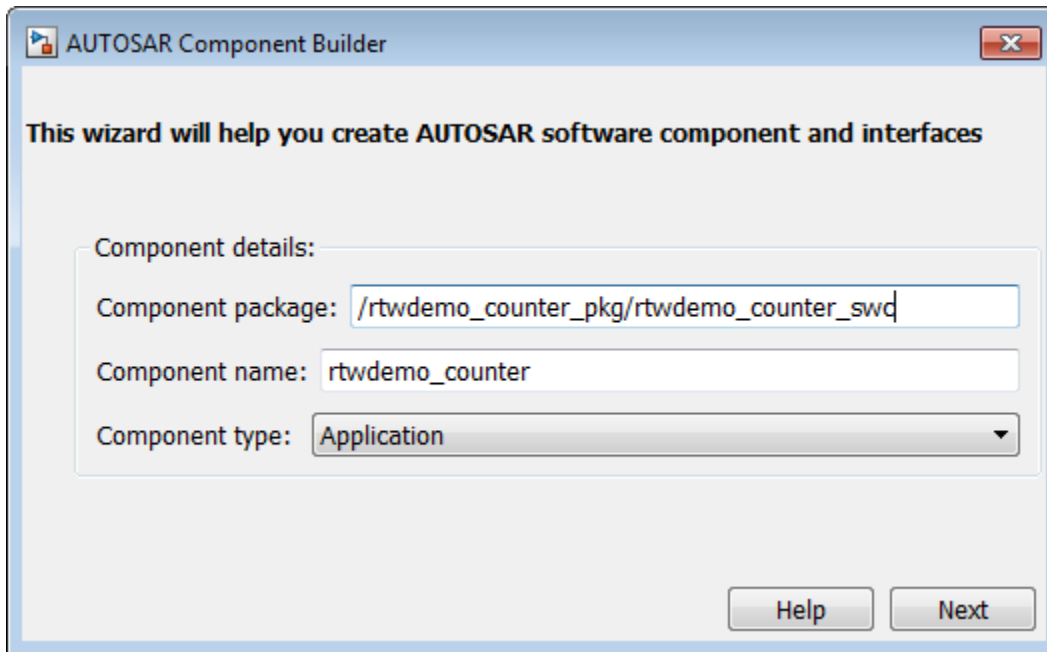
Embedded Coder software provides graphical user interfaces that allow you to add AUTOSAR elements to a Simulink model and map model components and interfaces to AUTOSAR components and interfaces. R2013a provides several improvements to the graphical user interfaces for AUTOSAR configuration:

- The Configure AUTOSAR Interface dialog box now provides separate **Simulink-AUTOSAR Mapping** and **AUTOSAR Properties** Explorers, which clearly distinguish mapping and editing activities.
- In both the Mapping and Properties Explorers:
 - Parameters that previously required text entry now offer selectable values or attributes.
 - Displays are more scalable (accommodating more elements) with fewer refresh issues.
 - Graphical layout reflects logical relationships between entities.
 - Filtering enhances element selection and modification.
- The Properties Explorer provides intuitive double-click and add/remove operations for configuring AUTOSAR ports, interfaces, data elements, runnables, and events.
- New check and synchronization icons provide single-click access to AUTOSAR configuration validation and Simulink model synchronization.
- A new AUTOSAR Component Builder dialog box allows you to interactively create a customized AUTOSAR component.

To explore the Configure AUTOSAR Interface dialog box, open a model that is already configured for AUTOSAR (such as the example model `rtwdemo_autosar_counter`). Select **Code > C/C++ Code > Configure Model as AUTOSAR Component** to open the dialog box. From there, you can select either the **Simulink-AUTOSAR Mapping** Explorer or the **AUTOSAR Properties** Explorer.



To explore the AUTOSAR Component Builder dialog box, open a model that is not already configured for AUTOSAR (such as the example model `rtwdemo_counter`). Select the AUTOSAR target (`autosar.tlc`) for the model, and then select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens a dialog box that allows you to select between creating a default AUTOSAR component or interactively creating an AUTOSAR component. To open the AUTOSAR Component Builder dialog box, click **Create Component Interactively**.



Round-trip preservation of AUTOSAR elements and UUIDs

To help support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink model-based design environment, Embedded Coder now preserves AUTOSAR elements and their UUIDs across arxml import and export, as follows:

- When arxml files created by an AAT are imported into a Simulink model, AUTOSAR element information is preserved, including UUIDs (for Identifiables), properties, and reference packages.
- When arxml files are exported from a Simulink model, the elements are generated back into arxml with their UUIDs and other information preserved.

As a result, the arxml files exported from Simulink can more easily be merged back into the AAT environment. Existing elements retain their UUIDs, while new elements created in Simulink get new UUIDs.

Code generation for variable-size scalar signals

Previously, a model that used a variable-size scalar signal (width equals 1) would cause an error during a model update. This limitation has been removed and the model now simulates and generates code for a variable-size scalar signal.

Data, Function, and File Definition

Shortened system-generated identifier names

In R2013a, you have the option to shorten the system-generated identifier names to allow more space for user names. This option also provides a more predictable and consistent naming system that uses camel case, no underscores or plurals, and consistent abbreviations for both a type and a variable.

To use the new names, open the Configuration Parameters dialog box, and on the **Code Generation > Symbols** pane, set the **System-generated identifiers** parameter to Shortened. When you open a new model in R2013a, the default setting for **System-generated identifiers** is set to Shortened. When you open an existing model in R2013a, **System-generated identifiers** is set as Classic. With this setting, the system-generated identifiers use the names from previous releases.

For more information, see System-generated identifiers and Customize Generated Identifier Naming Rules.

Improved data initialization with custom storage classes

Previously, Embedded Coder generated initialization code for these two cases, even though the **DataInitialization** parameter was set to None or Static.

- 1 Initial output of an Enabled Subsystem configured to reset when it is enabled.
- 2 Fixed-point data with bias, which cannot have zero ground value

Now, Embedded Coder will not generate dynamic initialization code for data that uses a custom storage class whose **DataInitialization** parameter is set to None or Static.

Default specification for global types

Previously, on the Configuration Parameters **Symbol** pane, the default for **Global types** was $\$N\$R\$M$. In R2013a, for new models, the default for **Global types** is $\$N\$R\$M_T$. For existing models opened in R2013a, **Global types** does not change.

Subsystem block parameter Function packaging option renamed

In the Subsystem block parameter dialog box, on the **Code Generation** tab, the Function packaging option Function is renamed to Nonreusable function.

Code Generation

Model Advisor checks for code generation

The Model Advisor **By Product** folder contains the following checks to replace **Identify questionable blocks within the specified system**:

- Check for blocks not supported by code generation
- Check for blocks not recommended for C/C++ production code deployment

To display the **By Product** folder, in the Model Advisor window select **Settings > Preferences**. In the Model Advisor Preferences dialog box, select **Show By Product Folder**.

Deployment

Concurrent execution API to target embedded multicore platforms

Semaphore and mutex code replacement for multicore target environments

Embedded Coder software now provides Simulink code replacement support for the following semaphore and mutex operations.

Mutex Destroy
Mutex Init
Mutex Lock
Mutex Unlock
Semaphore Destroy
Semaphore Init
Semaphore Post
Semaphore Wait

Semaphore and mutex code replacement is supported for:

- Simulink code generation for data transfer between tasks
- Code generation targets

Semaphore and mutex code replacement is not supported for:

- Stateflow charts, MATLAB Function blocks, and MATLAB functions
- Simulation targets

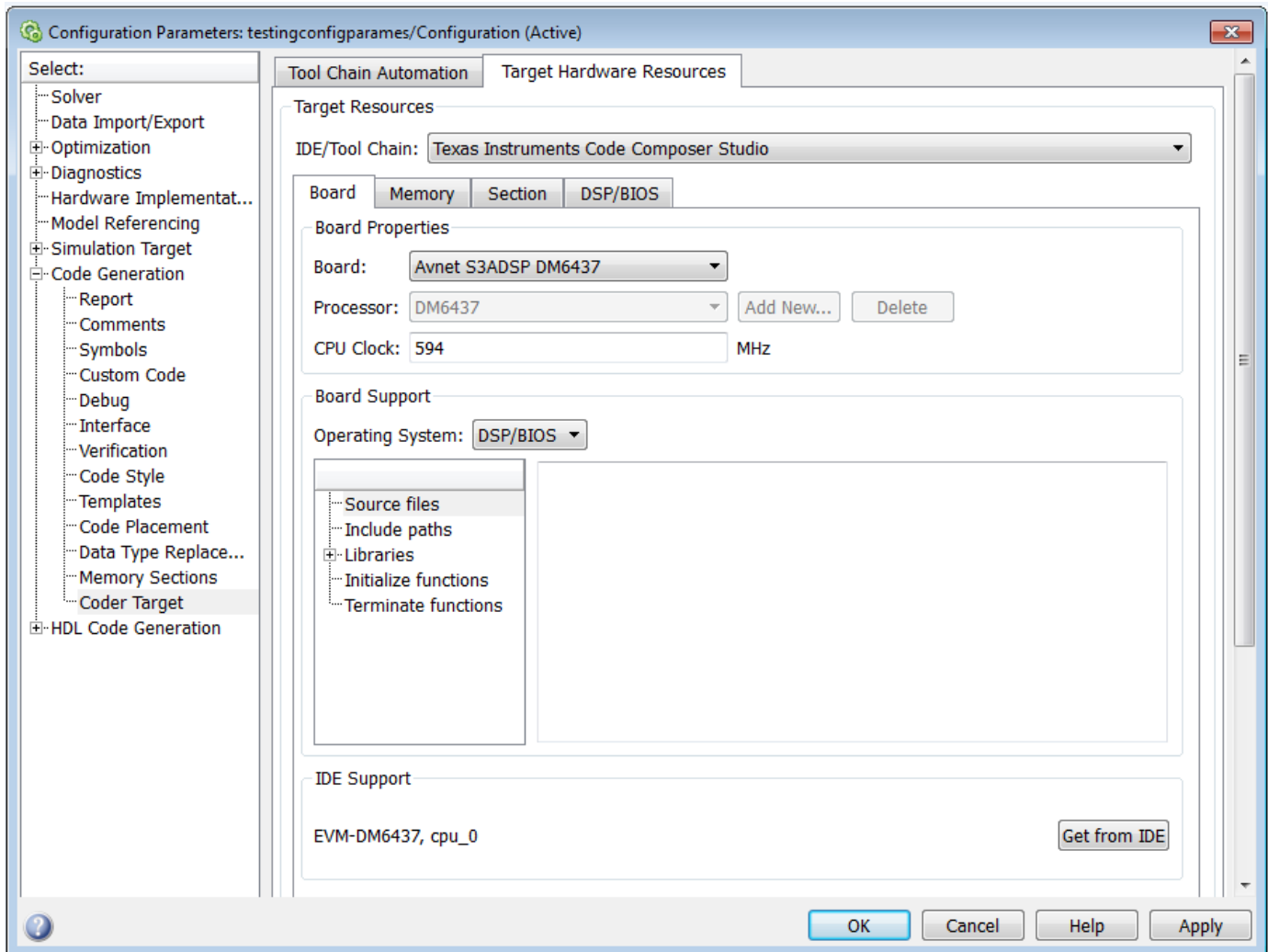
For more information, see Map Semaphore or Mutex Operations to Target-Specific Implementations.

Hardware timer function replacement

You can create a hardware-specific timer object for SIL and PIL simulations with your hardware target. See *Specification of hardware timer through the Code Replacement Tool* in “Code execution profiling improvements” on page 21-16.

Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box

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 Embedded 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 target hardware, see Model Setup.

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

Downloadable support and blocks for Analog Devices DSPs

If you have an Embedded Coder license, you can install support for Analog Devices VisualDSP++ IDE and DSPs as described in Install Support for Analog Devices DSPs. Support for Analog Devices VisualDSP++ IDE and DSPs includes the same capabilities that were previously available.

Use the “Embedded Coder Support Package for Analog Devices DSPs” block library to manage peripherals, scheduling, and memory on Blackfin®, SHARC®, and TigerSHARC® DSPs.

To get these capabilities, in a MATLAB Command Window, enter `supportPackageInstaller`. Then, use Support Package Installer to install the support package for Analog Devices DSPs. For more information, see the Working with Analog Devices VisualDSP++ IDE topic.

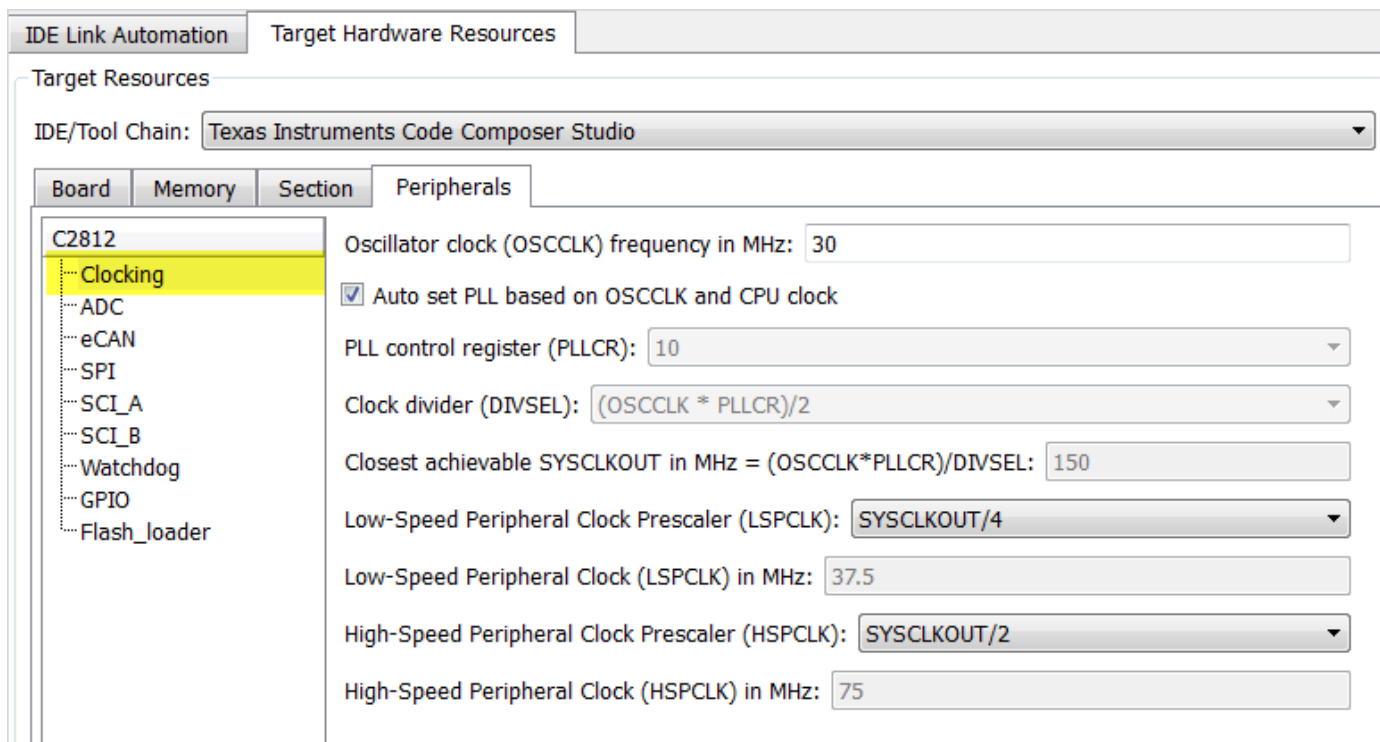
After installing the support package, you can open the block library. In the MATLAB Command Window, enter `adivdspLib`. The “Embedded Coder Support Package for Analog Devices DSPs” block library is also available in the Simulink Library Browser.

Compatibility Considerations

Previously, installing Embedded Coder software automatically installed support and blocks for Analog Devices DSPs. Effective this release, you must use Support Package Installer to install support before using Embedded Coder with Analog Devices DSPs.

Texas Instruments C2000 Clocking Options

In the Configuration Parameters dialog box, on the **Peripherals** tab, the new **Clocking** options help you to configure different timers that you use in the processor peripherals.



- The high-speed and low-speed clock settings help you to configure the baud rates for peripherals, such as SCI and SPI.
- You can specify the oscillator clock frequency used in the processor to set the system clock out parameter for the device. Based on the system clock out value, you can get feedback on the baud rate and the time settings.

- Automatic setting of the prescalers is done based on user-defined baud rate for peripherals, such as SCI and SPI.
- Based on the settings that you make in the Clocking peripheral, you can see the timing-related feedback for the peripherals, such as eCAN, I2C, ADC, and Watchdog.
- The parameter relationship is shown at the prompts in some of the peripherals. For example, in eCAN, at the baud rate parameter, you can see, CAN Module Clock/BRP/(TSEG1+TSEG2+1)) in bits/sec.

Support for Texas Instruments C2802x and Texas Instruments C2803x variants

You can now run models on the following variants of TI C2802x and TI C2803x processors:

- F28030
- F28031
- F28032
- F28033_cpu
- F28034
- F280200
- F28020
- F28021
- F28022
- F28026

You can use the following block libraries with these variants:

- C2802x (c2802xlib)
- C2803x (c2803xlib)

Downloadable support and blocks for Xilinx Zynq-7000 platform

Use the Embedded Coder Support Package for Xilinx Zynq-7000 Platform to automatically build (makefile-based), download, and run an executable on the Xilinx Zynq-7000 SoC ZC702 Evaluation Kit. The executable runs in the Linux environment on the ARM Cortex-A9 processor on the ZC702 Evaluation Kit.

Use blocks from the Embedded Coder Support Package for Xilinx Zynq-7000 Platform block library:

- The UDP Receive and UDP Send blocks enable communication with networked devices using an Ethernet port.
- The Linux Task block spawns a task function as separate Linux thread.

To download and install this feature, click **Add-Ons > Get Hardware Support Packages** on the MATLAB toolstrip. Then, use Support Package Installer to install the Embedded Coder Support Package for Xilinx Zynq-7000 Platform. For more information, see the Working with the Xilinx Zynq-7000 Platform topic.

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.

Support ending for remoteBuild method in a future release

Support for the `remoteBuild` method will end in a future release of the Embedded Coder products.

Compatibility Considerations

Use Support Package Installer to install the support package for BeagleBoard hardware, as described in [Install Support for BeagleBoard Hardware](#). Then, use the Simulink capability called “Run on Target Hardware” instead of `remoteBuild` to build and run models on BeagleBoard hardware.

For more information about using Run on Target Hardware with BeagleBoard, see the [BeagleBoard](#) topic.

Performance

Optimized function arguments for nonreusable subsystems

For nonreusable subsystems, you can specify the function interface in the generated code to use arguments. This specification reduces global RAM. It might reduce code size and improve execution speed, and allow the code generator to apply additional optimizations.

To optimize the function interface for a subsystem, in the Subsystem Block Parameter dialog box, on the **Code Generation** tab, set the **Function packaging** parameter to **Nonreusable function** (previously, named **Function**). The **Function packaging** parameter enables the **Function interface** parameter. Set the **Function interface** parameter to **Allow arguments**.

For more information, see [Function interface and Reduce Global Variables in Nonreusable Subsystem Functions](#).

Reduced data copies for tunable parameter expressions

Previously, in the generated code, tunable parameter expressions were copied to a temporary variable. In R2013a, the generated code removes this temporary variable. The removal of this unnecessary data copy improves execution speed, reduces code size and global RAM, and allows for additional code optimizations.

For example, for a tunable parameter, *b*, used in a Constant block, the code was:

```
/*Constant: '<Root>/Constant'*/
for (i=0; i<9; i++){
    tunable_expr_copy_B.Constant[i] = Param.b[i];
}
/*End of Constant: '<Root>/Constant'*/

/*S-Function(MySFun2D): '<Root>/S-Function Builder'*/
MySFun2D_Outputs_wrapper(tunable_expr_copy_B.Constant);
```

Now, the generated code is:

```
/*S-Function(MySFun2D): '<Root>/S-Function Builder'*/
MySFun2D_Outputs_wrapper(Param.b);
```

Removal of unused global variables

In R2013a, unused global variables generated from a For Each subsystem and bitfields are removed. This code generation enhancement reduces global RAM.

Verification

Debugging during SIL simulations

If you notice differences between the results of a Normal mode simulation and a SIL mode simulation, you can select the **Configuration Parameters > Verification > Enable source-level debugging for SIL** check box and rerun the SIL simulation. Then, from the Microsoft Visual Studio IDE, you can insert break points in the generated source code and step through the code during the SIL simulation. Observing code behavior in this way can help you to understand the differences in results. For example, when you are trying to integrate legacy code with generated code and the integration does not run as expected.

For more information, see [Debugging During SIL Simulations](#).

Simulation of multiple SIL Model blocks in a top model

If you have a top model containing Model blocks, you can simulate the model with multiple Model blocks in SIL mode. Previously, you could not simulate the top model with more than one Model block in SIL mode. To verify the different Model blocks, you had to run multiple simulations. Before each simulation, you had to specify the SIL mode for one Model block. The removal of this limitation reduces verification time.

If you specify code coverage or code execution profiling, the software does not support this feature.

API for testing rtiostream communications

To run PIL or External mode simulations with custom hardware, you write your own `rtiostream` implementations.

R2013a provides a test suite to debug and prove the behavior of custom `rtiostream` interface implementations.

This new API has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

This test suite has two parts. One part of the test suite runs on the target.

To launch this part, compile and link the following files, which are in `matlabroot/toolbox/coder/rtiostream/src/rtiostreamtest`.

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h`
- `rtiostream` implementation under investigation (e.g., `rtiostream_tcpip.c`)
- `main.c`

To run the second part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
function rtiostreamtest(connection,param1,param2)
```

- `connection` is a string indicating the communication method. It can have values `'tcp'` or `'serial'`.
- `param1` and `param2` have different values depending on the value of `connection`.
 - If `connection` is `'tcp'`: `param1`, `param2` are hostname and port, respectively.
 - If `connection` is `'serial'`: `param1`, `param2` are COM port and baud rate, respectively.

For example, you can run the second part of the test suite as follows:

```
function rtiostreamtest('tcp','localhost','2345')
```

SIL and PIL support for targets with multicore processors

R2013a allows you to run SIL and PIL simulations of models that are configured for targets with multicore processors:

- You can run SIL and PIL simulations of **single-rate** component models in a concurrent execution model hierarchy, without modifying models or regenerating code.
- Previously, the configuration parameters, `TargetOS` and `ConcurrentTasks`, had to be the same across a model hierarchy. This restriction has been removed.

Additional code annotation for justifying Polyspace checks

New Polyspace code annotations have been added to justify occurrences of `<<` and `+` inside fixed-point multiplication helper functions.

For more information, see [Code Annotation for Justifying Polyspace Checks](#).

Code execution profiling improvements

Comprehensive measurement and reporting of function execution times

R2013a provides comprehensive measurement and reporting of function execution times:

- The software measures execution times for initialization, shared utility and math library functions.
- The software inserts instrumentation probes around a function call site so that the measured time includes the time taken to call the function. Previously, the software inserted instrumentation probes inside the function. As a result, the measured time represented the execution time for only the function body.
- You can specify the time unit and numeric format for the time measurements in the code execution profiling report. Previously, the software reported execution times only in clock ticks. For information about the new default specifications for time unit and numeric format, see [report](#).
- The code execution profiling report contains hyperlinks to function call sites in the SIL/PIL test harness. Previously, the report provided hyperlinks to only source code files generated from the model.

For more information, see [Code Execution Profiling](#).

Viewing and comparing execution time plots with the Simulation Data Inspector

You can use the Simulation Data Inspector to view and compare plots of function execution times. If you select **All measurement and analysis data** from the **Configuration Parameters > Code Generation > Verification > Save options** drop-down list, the software automatically imports SIL simulation results into the Simulation Data Inspector. This feature allows you to plot execution times and manage and compare plots from various simulations.

For more information, see [Configure Code Execution Profiling](#) and [View and Compare Code Execution Times](#).

Specification of hardware timer through the Code Replacement Tool

In SIL and PIL simulations, if your hardware target does not have built-in timer support, you must create a timer object that provides details of the hardware-specific timer and associated source files. In R2013a, you can specify this hardware-specific timer using either the graphical user interface of the Code Replacement Tool or the corresponding command line API. The software stores the timer information as a Code Replacement Library (CRL) table.

Previously, you could specify the timer using the MATLAB function `coder.profile.Timer`. However, support for this function will cease in a future release.

For more information, see [Specify Hardware Timer](#).

Code-to-model traceability links for reusable subsystems in libraries

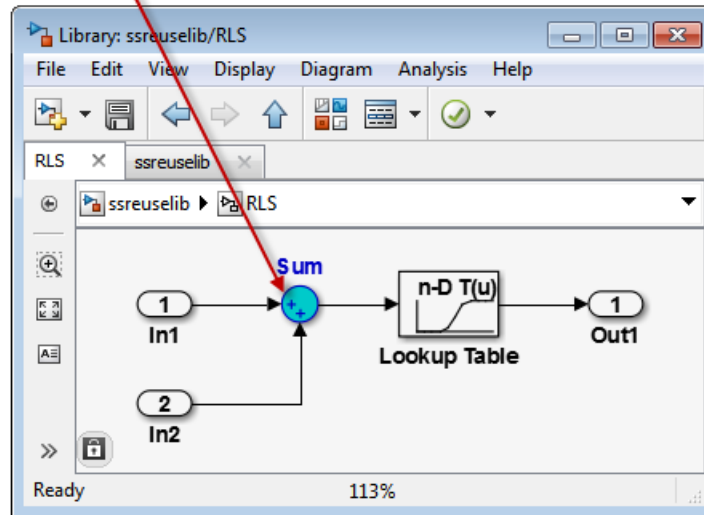
Code-to-model traceability links are now available in the generated code for a reusable library subsystem. Code-to-model traceability links for a reusable library subsystem appear in the comments of the generated code in the code generation report. The traceability link is the name of the library.

File: [RLS_HylfoiOq.c](#)

```

11
12 #include "RLS_HylfoiOq.h"
13
14 /* Output and update for atomic system: 'SS1' ('ssreuselib:1') */
15 void RLS_HylfoiOq(const real_T *rtu_In1, const real_T *rtu_In2, real_T *rty_Out1,
16                  const real_T rtp_y[11], const real_T rtp_x[11])
17 {
18     /* Lookup_n-D: 'Lookup Table' ('ssreuselib:4') incorporates:
19      * Sum: 'Sum' ('ssreuselib:5')
20      */
21     *rty_Out1 = look1_binlpx(*rtu_In1 + *rtu_In2, rtp_x, rtp_y, 10U);
22 }

```



To include traceability links in the generated code comments, see [Traceability in Code Generation Report](#).

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.3

New Features

Bug Fixes

Compatibility Considerations

Cyclomatic complexity measurement in static code metrics report

In R2012b, the static code metrics report includes a cyclomatic complexity measurement for each function. You can view the measurement in the **Complexity** column of the Function Information table. For more information, see Analyze Static Code Metrics.

Custom code substitution for MATLAB functions using code replacement libraries

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement library (CRL) function in the generated code. You can use `coder.replace` both in MATLAB code from which you want to generate C code using MATLAB Coder and in MATLAB code in a MATLAB Function block. For more information, see `coder.replace`, Replace MATLAB Function with Custom Code, and Replace MATLAB Function Block Code with Custom Code.

In addition, you can use the code replacement tool to create and register code replacement tables. These tables provide the basis for replacing default math functions and operators in your generated code with target-specific code. The ability to control function and operator replacements potentially allows you to optimize target speed and memory and better integrate generated code with external and legacy code.

Access the code replacement tool using one of these methods:

- At the MATLAB command line, enter:

```
crtool
```
- On the MATLAB Coder **Project Settings** dialog box **Hardware** tab, click the **Custom** link.

For more information, see Create Code Replacement Table for a Sample MATLAB Coder Project.

SIL and PIL support for signal logging, encapsulated C++, and AUTOSAR calibration parameters

Beginning in R2012b, Embedded Coder software supports using Simulink signal logging, encapsulated C++ code, and AUTOSAR calibration parameters in SIL and PIL mode simulations.

Signal logging for SIL and PIL simulations

In R2012b, Simulink signal logging is extended to the SIL and PIL simulation modes. This allows you to:

- Collect signal logging outputs (e.g., `logout`) during SIL and PIL simulations.
- Log the internal signals and the root-level outputs of a SIL or PIL component.
- Manage the SIL and PIL signal logging settings using the Simulink Signal Logging Selector.
- More easily compare logged signals between normal, SIL, and PIL simulations, for example, using Simulation Data Inspector.

Signal logging is supported with the following forms of SIL and PIL simulation:

- Top-model SIL or PIL
- Model block (referenced model) SIL or PIL

SIL or PIL signal logging requires the following model configuration settings:

- On the **Data Import/Export** pane of the Configuration Parameters dialog box, set **Signal logging format** to Dataset.
- On the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set **Interface** to C API.

Use SIL and PIL simulations to verify encapsulated C++ code

Previously, you could use SIL and PIL simulations to verify code generated with the model configuration **Language** setting C or C++. Beginning with R2012b, you can also use the **Language** setting C++ (Encapsulated).

Encapsulated C++ code is supported with the following forms of SIL and PIL simulation:

- SIL or PIL block
- Top-model SIL or PIL
- Model block (referenced model) SIL or PIL

Improved SIL and PIL verification for AUTOSAR-compliant code

The following forms of SIL and PIL simulation support AUTOSAR calibration parameters in generated code:

- SIL or PIL block
- Top-model SIL or PIL

You can use the calibration parameter custom storage classes `CalPrm` and `InternalCalPrm` to reference data.

AUTOSAR 4.0 nonscalar data support

R2012b extends Embedded Coder support for using nonscalar data in models from which AUTOSAR 4.0 compatible code is generated. Previously, you could use nonscalar data associated with port elements, calibration parameters, and per-instance memory. Beginning in R2012b, you also can use nonscalar inter-runnable variables (IRVs) in models configured for AUTOSAR.

For information about other AUTOSAR-related enhancements and changes, see “AUTOSAR software component import and export enhancements” on page 22-7.

Code annotation for justifying Polyspace checks

You can apply Polyspace verification to generated code using the Polyspace Model Link™ SL product. The software detects run-time errors in the generated code. It also helps you to locate and fix model faults.

Because of the way Embedded Coder implements certain operations, Polyspace might indicate potential overflows for operators or operations that are actually legitimate.

Previously, you manually justified the associated orange checks in the Polyspace verification environment.

Now, if you select the new check box, **Configuration Parameters > Code Generation > Comments > Auto generate comments > Operator annotations**, the Embedded Coder software annotates the generated code with comments for Polyspace. When you run a Polyspace verification, the Polyspace software uses the comments to justify overflows associated with legitimate operations and assigns the `Not a Defect` classification to the corresponding checks.

For more information, see [Code Annotation for Justifying Polyspace Checks](#).

Texas Instruments Code Composer Studio IDE 5.1 support

This release adds support for version 5.1 of the Texas Instruments Code Composer Studio IDE (CCS) to existing support for CCS versions 3.3 and 4.1.

Support for CCS version 5.1 includes the following capabilities:

- Automatic creation of makefile projects
- Support for DSP/BIOS™ version 5.41.xx
- Support for C6000 Compiler version 7.3.x

For more information, see [Working with Texas Instruments Code Composer Studio IDE](#).

External mode support for ERT targets with static main

Previously, Embedded Coder software supported External mode for ERT targets only if the associated main program was automatically generated by the model build process. Beginning in R2012b, the software also supports External mode for ERT targets with a static main program. Specifically, the static main file `matlabroot/rtw/c/src/common/rt_main.c` has been enhanced to support External mode.

If you have authored a custom ERT-based target, you can support External mode with your custom main program by updating your main program, using the code in `rt_main.c` as an example.

Downloadable support for Green Hills MULTI

If you have an Embedded Coder license, you can install support for Green Hills MULTI IDE (MULTI) as described in [Install Support for Green Hills MULTI IDE](#). Support for MULTI includes the same capabilities that were previously available.

After installing support for MULTI, you can use the “Target for Use with Green Hills MULTI IDE” block library, located in the Simulink Library Browser. You can open this block library by entering `idelinklib_ghsmulti` in the MATLAB Command Window.

The block library contains blocks for:

- Analog Devices Blackfin processors
 - Memory Allocate
 - Memory Copy
 - Blackfin Hardware Interrupt
 - Idle Task

- Freescale MPC55xx and MPC74xx processors
 - Memory Allocate
 - Memory Copy
 - Idle Task
 - MPC5500 Interrupt
 - MPC7400 Hardware Interrupt

Compatibility Considerations

Previously, Embedded Coder software included support for MULTI. Now, use Target Installer to install support before using Embedded Coder with MULTI.

Support for Texas Instruments C2806x processors

This release adds support for Texas Instruments C2806x processors to Embedded Coder.

This support adds the C2806x (c2806xlib) block library to the Simulink Library Browser. The C2806x block library includes the following blocks:

- C2802x/C2803x/C2806x ADC
- C2802x/C2803x/C2806x AnalogIO Input
- C2802x/C2803x/C2806x AnalogIO Output
- C28x CAN Calibration Protocol
- C2802x/C2803x/C2806x COMP
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x GPIO Digital Input
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x GPIO Digital Output
- C28x I2C Receive
- C28x I2C Transmit
- C28x SCI Receive
- C28x SCI Transmit
- C28x SPI Receive
- C28x SPI Transmit
- C28x Software Interrupt Trigger
- C28x Watchdog
- C28x eCAN Receive
- C28x eCAN Transmit
- C28x eCAP
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x ePWM
- C28x eQEP

For more information, see C2806x (c2806xlib).

Performance enhancement of Simulink data objects

In R2012b, Simulink can create and load subclasses of Simulink data classes more efficiently. To take advantage of this enhancement, use the `setupCoderInfo` method to configure the `CoderInfo` object of your class. The `setupCoderInfo` method is called once during object construction.

Consider the example of the `ECoderDemos.Parameter` class. Previously, this class was defined as follows. Notice how the `CoderInfo` object is configured in the class constructor.

```
classdef Parameter < Simulink.Parameter
% ECoderDemos.Parameter Class definition.

    methods
        function h = Parameter(optionalValue)
            % Use custom storage classes from this package
            useLocalCustomStorageClasses(h, 'ECoderDemos');

            % Set up object to use custom storage classes by default
            h.CoderInfo.StorageClass = 'Custom';

            % Initialize Value property
            switch nargin
                case 0,
                    % No action
                case 1,
                    h.Value = optionalValue;
            end
        end
    end % methods
end % classdef
```

In this release, the `ECoderDemos.Parameter` class is defined as follows. Notice the use of the `setupCoderInfo` method to configure the `CoderInfo` object. The rest of the constructor method is unchanged.

Note You can access this class definition at `matlabroot/toolbox/rtw/targets/ecoder/ecoderdemos/dataclasses/+ECoderDemos/@Parameter/Parameter.m`.

```
classdef Parameter < Simulink.Parameter
% ECoderDemos.Parameter Class definition

    methods
        function setupCoderInfo(h)
            % Use custom storage classes from this package
            useLocalCustomStorageClasses(h, 'ECoderDemos');

            % Set up object to use custom storage classes by default
            h.CoderInfo.StorageClass = 'Custom';
        end

        function h = Parameter(optionalValue)
            % Initialize Value property
            switch nargin
                case 0,
                    % No action
                case 1,
                    h.Value = optionalValue;
            end
        end
    end % methods
end % classdef
```

AUTOSAR software component import and export enhancements

R2012b adds AUTOSAR workflow improvements, including import validation and faster import and export of arxml files. See also “AUTOSAR 4.0 nonscalar data support” on page 22-3.

Import validation

Beginning in R2012b, the AUTOSAR software component importer validates the XML in the imported arxml files. If XML validation fails for a file, the importer displays errors. For example:

```
Error
The IsService attribute is undefined for interface /mtest_pkg/mtest_if/In1
in file hArxmlFileErrorMissingIsService_SR_3p2.arxml:48.
Specify the IsService attribute to be either true or false
```

In this example message, the file name is a hyperlink, and you can click the hyperlink to see the location of the error in the arxml file.

Faster import and export of arxml files

Beginning in R2012b, Embedded Coder software provides up to 20 times faster import and export of AUTOSAR software component descriptions.

Explicit access mode for AUTOSAR Sender and Receiver ports

Previously, the AUTOSAR software component importer did not support explicit data access modes for AUTOSAR component Sender and Receiver ports. It issued a warning for an explicit data access mode and set the port data access mode to implicit. Beginning in R2012b, the importer analyzes the AUTOSAR software component to determine whether the data access mode for a port is implicit or explicit. The importer honors an explicit access mode setting. However, if conflicting data access modes are detected, the importer issues a warning and sets the data access mode to implicit.

Import port-based calibration parameters

The AUTOSAR software component importer has been enhanced to import any port-based calibration parameters referenced in the AUTOSAR software component. For each imported parameter, the importer creates a data object in the MATLAB base workspace.

Highlight virtual blocks in model Web view of code generation report

In the model Web view of the code generation report, when tracing between the model and the code, if you click a virtual block and no code is highlighted in the generated code pane, the virtual block is highlighted yellow.

Code Execution Profiling Improvements

Updated Code Execution Profiling API

The existing code execution profiling APIs, `rtw.pil.ExecutionProfile` and `rtw.pil.ExecutionProfileSection`, have been replaced with `coder.profile.ExecutionTime` and `coder.profile.ExecutionTimeSection` respectively.

Compatibility Considerations

The old class names and methods forward to the corresponding new class names and methods. A warning is not issued. The old method names are hidden and no longer documented.

New Properties and Methods

The following new methods and properties have been added:

Interface	Method or Property
coder.profile.Timer	coder.profile.Timer
coder.profile.ExecutionTime	display
	Sections
	TimerTicksPerSecond
	report
coder.profile.ExecutionTimeSection	ExecutionTimeInTicks
	MaximumExecutionTimeCallNum
	MaximumExecutionTimeInTicks
	MaximumSelfTimeCallNum
	MaximumSelfTimeInTicks
	Name
	Number
	NumCalls
	SampleOffset
	SamplePeriod
	SelfTimeInTicks
	TotalExecutionTimeInTicks
	TotalSelfTimeInTicks

Functionality Being Removed or Changed

The following functionality is being removed or changed:

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.connectivity.Timer	Call is forwarded to coder.profile.Timer without warning message.	coder.profile.Timer	All methods are the same as rtw.connectivity.Timer.
rtw.pil.ExecutionProfile.display	Call is forwarded to coder.profile.ExecutionTime.display without warning message.	display	None

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.pil.ExecutionProfile.report	Call is forwarded to coder.profile.ExecutionTime.report without warning message.	report	None
rtw.pil.ExecutionProfile.getSectionProfile rtw.pil.ExecutionProfile.getNumSectionProfiles	Call is forwarded to coder.profile.ExecutionTime.Sections without warning message.	Sections	Uses property syntax
rtw.pil.ExecutionProfile.getTimerTicksPerSecond rtw.pil.ExecutionProfile.setTimerTicksPerSecond	Calls are forwarded to property coder.profile.ExecutionTime.TimerTicksPerSecond without warning message.	TimerTicksPerSecond	Uses property syntax
rtw.pil.ExecutionProfile-Section.getMaxTicks	Call is forwarded to coder.profile.ExecutionTimeSection.MaximumExecutionTimeInTicks without warning message.	MaximumExecutionTimeInTicks	Uses property syntax
rtw.pil.ExecutionProfile-Section.getName	Call is forwarded to coder.profile.ExecutionTimeSection.Name without warning message.	Name	Uses property syntax
rtw.pil.ExecutionProfile-Section.getNumCalls	Call is forwarded to coder.profile.ExecutionTimeSection.NumCalls without warning message.	NumCalls	Uses property syntax
rtw.pil.ExecutionProfile.getSectionNumber	Call is forwarded to coder.profile.ExecutionTime.Number without warning message.	Number	Uses property syntax
rtw.pil.ExecutionProfile-Section.getTicks	Call is forwarded to coder.profile.ExecutionTimeSection.ExecutionTimeInTicks without warning message.	ExecutionTimeInTicks	Uses property syntax
rtw.pil.ExecutionProfile.getTimes	Call is forwarded to the legacy getTimes function without warning message.	Calculate execution time in seconds by the formula $ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond.$	No equivalent to getTimes in new interface.
rtw.pil.ExecutionProfile-Section.getTotalTicks	Call is forwarded to coder.profile.ExecutionTimeSection.TotalExecutionTimeInTicks without warning message.	TotalExecutionTimeInTicks	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.pil.ExecutionProfile-Section.getSampleOffset	Call is forwarded to coder.profile.Execution-TimeSection.SampleOffset without warning message.	SampleOffset	Uses property syntax
rtw.pil.ExecutionProfile-Section.getSamplePeriod	Call is forwarded to coder.profile.Execution-TimeSection.SamplePeriod without warning message.	SamplePeriod	Uses property syntax
rtw.pil.ExecutionProfile-Section.getTotalSelfTicks	Call is forwarded to coder.profile.Execution-TimeSection.TotalSelf-TimeInTicks without warning message.	TotalSelfTimeInTicks	Uses property syntax

Code Execution Profiling Supports Single Object Output

Code execution profiling during a SIL or PIL simulation honors the **Save simulation output as a single object** setting.

If the **Measure task execution time** check box is selected in the **Verification** pane and the **Save simulation output as a single object** check box is selected in the **Data Import/Export** pane, then the **Workspace variable** defined in the **Verification** pane is saved in the single output object instead of in the base workspace.

Incremental Compilation with Changes in Code Coverage Settings

If only code coverage settings have changed and the generated code is otherwise up to date, code is not regenerated. Instead, the existing up-to-date code is recompiled using the new code coverage settings.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.2

New Features

Bug Fixes

Compatibility Considerations

AUTOSAR Enhancements

AUTOSAR Release 4.0

R2012a supports AUTOSAR Release 4.0 (version 4.0.2), which includes:

- Import and export of AUTOSAR R4.0 XML files
- Generation of AUTOSAR R4.0 code
- Support for *application* and *implementation* data types and *base* types. For more information, see Data Type Support for Release 4.0.
- **Code replacement library** (CRL) support for over 300 routines from the following AUTOSAR libraries:
 - Floating-Point Math (AUTOSAR_SWS_MFLLibrary)
 - Fixed-Point Math (AUTOSAR_SWS_MFXLibrary)

Support for Schema 2.0 Removed

Support for AUTOSAR schema version 2.0 has been removed from R2012a. The software now supports the following schema versions:

- 4.0 (4.0.2)
- 3.2 (3.2.1)
- 3.1 (3.1.4) — Default
- 3.0 (3.0.2)
- 2.1 (XSD rev 0017)

Code Efficiency Enhancements

For Each Subsystem Loop Bound Passed by Value

The generated code of the For Each subsystem includes a loop bound that was previously passed by a pointer. In R2012a, the loop bound is passed by value which improves memory usage and execution speed.

For example, if you have a For Each subsystem with a **Function name**, `myFcnVectorized`, the generated code for the function prototype is:

```
void myFcnVectorized(int32_T NumIters, ...) {  
    for (ForEach_itr = 0;  
         ForEach_itr < NumIters;  
         ForEach_itr++) { ...
```

The argument `NumIters` is passed by value, instead of by pointer. The function is called as follows:

```
myFcnVectorized(3, ...
```

For more information, see For Each Subsystem.

Fully Inlined S-functions from Legacy Code Tool

The Legacy Code Tool now automatically generates fully inlined S-functions for legacy code. Previously, the generated code included an unnecessary data copy for the function-call input. In

R2012a, these temporary variables are no longer generated. This enhancement reduces memory usage and improves execution speed, as well as enabling other optimizations and a consistent coding style.

For example, temporary variables, `tmp` and `tmp_0`, were used for the generated function-call input:

```
int32_T i;
real_T tmp[6];
real_T tmp_0[6];
for (i = 0; i < 6; i++) {
/* S-Function (rtwdemo_sfundarray_add):'<S1>/rtwdemo_sfundarray_add' */
array3d_add(rtb_Output1,tmp,tmp_0,1,2,3);
```

Now, the generated code is:

```
int32_T i;
/* S-Function (rtwdemo_sfundarray_add):'<S1>/rtwdemo_sfundarray_add' */
array3d_add(rtb_Output1, rtwdemo_lct_ndarray_ConstP.Constant_Value,
rtwdemo_lct_ndarray_ConstP.Constant1_Value, 1, 2, 3);
```

For more information, see [Integrate External Code Using Legacy Code Tool](#).

Element-Wise Operations as Inputs to Intrinsic Functions

In previous releases, element-wise operations were performed in temporary variables before being used as inputs in an intrinsic function call. In R2012a, element-wise operations are performed within the intrinsic function call to improve memory usage and execution speed.

For example, in previous releases when you generated code for the following MATLAB code:

```
function y = matrixExpand(u1, u2)
    eml.varsize('u1', [4, 8, 10]);
    eml.varsize('u2', [4, 8, 10]);
    y = isnan(u1 + u2);
```

element-wise operations were stored in a temporary variable, `x_data`, which became the input to the generated intrinsic function, `muDoubleScalarIsNaN`:

```
for (i = 0; i <= loop_ub; i++) {
    x_data[i] = u1_data[i] + u2_data[i];
}
...
for (i = 0; i <= loop_ub; i++) {
    y_data[i] = muDoubleScalarIsNaN(x_data[i]);
}
```

In R2012a, the temporary variable is eliminated in the generated code and the element-wise operations occur in the function call input:

```
for (i = 0; i <= loop_ub; i++) {
    y_data[i] = muDoubleScalarIsNaN(u1_data[i] + u2_data[i]);
}
```

Enhancements to Custom Storage Classes in Simulink and mpt Packages

In this release, enhancements have been made to the following custom storage classes (CSCs) in the Simulink package.

- **Owner** property added to Const, Volatile, ConstVolatile, ExportToFile
- **Definition file** property added to Const, Volatile, ConstVolatile, ExportToFile
- **Header file** property added to Const, Volatile, ConstVolatile, Define

The following enhancements have been made to CSCs in the mpt package

- **Owner** property has been added to ExportToFile
- Settings for the **Owner** and **Definition file** properties for Global, Custom, Volatile, and ConstVolatile CSCs have been moved from the **Other Attributes** tab to the **General** tab of the Custom Storage Class Designer.

Code Generation Report Includes Simulink Web View

R2012a supports integration of the Simulink Web view into the code generation report. You can view the generated code and model in a single web browser window without MATLAB and Simulink installed on your computer.

To generate a code generation report with the model Web view, on the **Code Generation > Report** pane of the model configuration parameters, select:

- **Create code generation report**
- **Generate model Web view**
- **Open report automatically** (optional)

For navigation between the generated code and the model in the Web view, select

- **Code-to-model**
- **Model-to-code**

For more information, see Include Model Web View in HTML Code Generation Report. The model Web view requires a Simulink Report Generator license.

LDRA Testbed Code Coverage Annotations in Code Generation Report

If you specify the LDRA Testbed code coverage tool for a SIL/PIL simulation, the code generation report provides summary data and code annotations with LDRA Testbed coverage information. Each code annotation is associated with a code feature and indicates the nature of the feature coverage during code execution. See Code Coverage Summary and Annotations in Code Generation Report.

You should not use the code generation report alone to check that your coverage goals have been achieved. You must refer to the LDRA Testbed Report. See View Code Coverage Information at the End of SIL or PIL Simulations.

Generated Identifiers Enhancements

Simplified Identifiers for Model Reference Code

Previously, model reference identifiers were generated with the `mr_` prefix. In R2012a, code generation no longer includes the `mr_` prefix to identifiers. This naming convention is now consistent with the code generation of subsystem identifiers and other identifiers. For more information, see Configuring Generated Identifiers.

Consistent Identifiers for Comparing Generated Code

To generate unique identifiers in the generated code, the code generation process inserts a mangling string in an identifier name. Previously, the mangling string was generated using the full block path name, which included the model name. In R2012a, the mangling string uses the Simulink Identifier (SID), which is unique within the model. This mangling string allows for consistent identifiers for similar or derived models, because the SID is persistent even if you change the name of the model. If you create another model using **Save As**, the SID is preserved for each block. For blocks in a subsystem, the SID is preserved whether you build the subsystem or build the model containing the subsystem.

For example, you might want to make a structural change to a model and then see the impact of the change on the generated code. You can save your model using **Save As** and make a change to the saved model. To see only the change in the generated code due to the change in the model, you can compare the generated code from the original and derived model. Before R2012a, the identifiers from the derived model were different, because the mangling string included the different model names. It was difficult to see only the difference in the generated code from the change in the model. Now, when you compare the generated code for the two models, the difference is just the code resulting from the change in the derived model.

If you have an Embedded Coder license, see **Configure Generated Identifiers in Embedded System Code** for more information on customizing generated identifiers.

Code Replacement Enhancements

R2012a provides the following enhancements to code replacement library support.

Target Function Libraries Renamed to Code Replacement Libraries

In R2012a, target function libraries (TFLs) are renamed to code replacement libraries (CRLs). The change is reflected in software, demos, and documentation. The changes include the following:

- The model configuration parameter **Target function library** (TargetFunctionLibrary) is renamed to **Code replacement library** (CodeReplacementLibrary). The command line parameter TargetFunctionLibrary is still supported, but when you save a model, the library value is saved using the parameter CodeReplacementLibrary.
- The code replacement demo `rtwdemo_tfl_script` is renamed to `rtwdemo_crl_script`, and the `rtwdemo_tfl*` models associated with the demo are renamed to `rtwdemo_crl*`. For example, the model `rtwdemo_tfladdsub` is renamed to `rtwdemo_crladdsub`.
- The code replacement demo `coderdemo_tfl` is renamed to `coderdemo_crl`.
- The Target Function Library (TFL) Viewer is renamed to Code Replacement Viewer.

Code replacement related items that have *not* been renamed include code replacement classes, functions, and commands. Examples include the `RTW.TfLCOperationEntry` class, the `setTfLCFunctionEntryParameters` function, and the `RTW.viewTfl` command.

Enhanced Code Replacement Traceability

R2012a provides enhanced code replacement traceability, using the model option **Summarize which blocks triggered code replacements**, which is located on the **Code Generation > Report** pane of the Configuration Parameters dialog box. When you select **Summarize which blocks triggered code replacements**:

- Code generation includes a code replacement report in the HTML code generation report for your model.
- Code replacement trace information is generated for viewing in the **Trace Information** tab of the Code Replacement Viewer.

The code replacement report lists replacement functions and their associated blocks. You can use the report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the Simulink block that triggered the replacement.

For more information, see [Analyze Code Replacements in the Generated Code](#)

The **Trace Information** tab of the Code Replacement Viewer lists **Hit Source Locations** and **Miss Source Locations**. The Viewer provides links to each source location (the source block for which code replacement was considered) and, for misses, lists a **Miss Reason**. For example, if a rounding mode setting did not match between a CRL entry and a block, the Viewer displays a reason similar to the following: “Mismatched rounding mode: actual 'RTW_ROUND_SIMPLEST', expected 'RTW_ROUND_CEILING'.” After generating code for your model, you can open the Code Replacement Viewer for viewing hits and misses using the following commands:

```
>> crl=get_param('model','TargetFcnLibHandle')
>> RTW.viewTfI(crl)
```

When debugging a CRL entry, you can use code replacement report information together with hits and misses information in the Code Replacement Viewer to determine why a replacement function was not used in the generated code.

For more information, see [Trace Code Replacements Generated Using Your Code Replacement Library and Determine Why Code Replacement Functions Were Not Used](#).

Code Replacement Support for Simulink Matrix Division and Inversion Operators

Embedded Coder software now provides Simulink code replacement support for the following nonscalar division and inversion operators:

Operator	Key
Matrix right division (/)	RTW_OP_RDIV
Matrix left division (\)	RTW_OP_LDIV
Matrix inversion (inv)	RTW_OP_INV

For more information, see [Map Nonscalar Operators to Target-Specific Implementations](#).

Code Replacement Support for MATLAB Coder fix, hypot, round, and sign Functions

Embedded Coder software now provides MATLAB Coder code replacement support for `fix`, `hypot`, `round`, and `sign` functions.

Integer Functions Now Return Real-World Values

The following functions now return real-world values instead of stored integer values: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`.

Compatibility Considerations

In code generation with MATLAB Coder or Simulink Coder, if you used a CRL to replace a cast in your replacement function, silent incorrect numerical results may occur. The numerical results will not change if the input fi object has binary-point scaling and zero fractional length. To optimize code generation, these integer functions now use floor rounding, instead of nearest rounding, when the input fraction length equals 0. You should reevaluate your integer cast replacement functions and update their replacement tables.

SIL and PIL Enhancements

R2012a supports the following enhancements for software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations.

SIL and PIL Test Harness Files in Code Generation Report

For top-model and Model block SIL and PIL simulations, the software now displays test harness files and the corresponding static code metrics in the code generation report.

3. Function Information [\[hide\]](#)

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: [Call Tree](#) | [Table](#)

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines
[-] main	1,552	12	22	32
[+] xilInit	1,540	4	7	10
[+] xilTerminateComms	540	8	7	9
[-] xilRun	430	29	191	222
[+] processData	401	9	49	61
[+] finalizeCommandResponse	393	1	14	21
[+] xilReadData	384	24	22	28
[+] xilInitialize	4	4	8	12
xilInitializeConditions	4	4	7	11
xilGetDataTypeInfo	0	0	1	3
xilProcessParams	0	0	4	7
[-] xilOutput	0	0	11	14
[-] rtwdemo_sil_topmodel_step	0	0	5	26
CounterTypeA	0	0	11	43
CounterTypeB	0	0	8	31
xilTerminate	0	0	4	8
xilEnable	0	0	5	9
xilDisable	0	0	5	9
xilUpdate	0	0	5	9

This feature helps you to:

- Understand and review the SIL and PIL verification process.
- See how your registered custom target connectivity files fit into the target application that runs during a SIL or PIL simulation.

This feature is not available for simulations that you run with the PIL block. For more information, see [View Test Harness Files in Code Generation Report](#).

PIL Support for Code Coverage with LDRA Testbed

The target connectivity API supports code coverage with LDRA Testbed for the following types of PIL simulation:

- Top-Model PIL
- Model block PIL

Previously, support for code coverage during a PIL simulation was only available in special cases, where your PIL application could write directly to the host file system.

You can run PIL simulations on simulator or target hardware and collect code coverage metrics to support high integrity workflows, for example, DO-178B and ISO 26262. For more information, see [Use a Code Coverage Tool in SIL and PIL Simulations](#).

Seamless Switching Between SIL and PIL for Top-Model and Model Block

If you select **Configuration Parameters > SIL and PIL Verification > Enable portable word sizes**, you can switch between the SIL and PIL simulation modes without:

- Changing configuration parameters of your model
- Regenerating code (if your model is up-to-date)

This feature:

- Applies only to top-model and Model block SIL/PIL
- Requires that the code can be compiled by both the host computer and the target platform

If your target uses code that cannot be compiled on the host, then you see compilation errors when you try to simulate the model in SIL mode. You might be able to work around this problem by adding the source code files to the `SkipForSil` group in the build information object `RTW.BuildInfo`. The SIL build on the host platform does not compile source files present in the `SkipForSil` group. See [Code that the Host Cannot Compile](#).

Enhanced Hardware Implementation Support

Host and Target Floating Point Data Type Sizes

The host and target floating point data type sizes must be the same. Previously, a mismatch would produce undefined behaviour resulting in a simulation failure. Now, the software generates an error with a clear message when the host and target data types are *not*:

- 32 bits for single
- 64 bits for double

For more information, see [Hardware Implementation Support](#).

Word-Addressable Targets

Previously, the target connectivity API did not support word-addressable targets for PIL simulations or SIL simulations with `PortableWordSizes` enabled. This limitation has been removed.

In addition, data type sizes that are smaller than the target word sizes are now supported. See [Hardware Implementation Support](#).

The software uses the MATLAB host byte order when sending words through the `rtIOStream` API. For information about host byte ordering, see [computer](#).

Top-Model Output Limitations Removed

Previously, in a top-model SIL/PIL simulation, not all signal and output logging fields matched the fields produced by a Normal simulation. For example:

- With signal logging, the software would add the suffix `_wrapper` to the block path for signals in `logout`.
- With output logging, if the save format was `Structure` or `Structure` with `time`, the software would add the suffix `_wrapper` to the block name for signals in `yout`.

These limitations are not present in R2012a, except if you do one of the following:

- Specify the signal logging format to be `ModelDataLogs`. In this case, `yout` will still contain references to the wrapper model. You should use the `Dataset` signal logging format. See `Simulink.SimulationData.Dataset` in the Simulink reference documentation.
- Run command line simulations using the `sim` command but without specifying the single-output format. See [Using the sim Command](#).

Model Block SIL/PIL Support for Absolute Time

Previously, you could not run a Model block in the SIL or PIL mode if the Model block contained Simulink blocks that depended on absolute time. Now, Model block SIL/PIL supports absolute time except for the following case: the Model block contains Simulink blocks that require absolute time **and** the Model block is conditionally executed. See [Configuration Parameters Support](#).

Changes for ERT and ERT-Based Targets

In R2012a, the simplified model call interface used by ERT targets has been further streamlined. (The simplified call interface also is now available to GRT target users — see [Simplified Call Interface for Generated Code](#) in the R2012a Simulink Coder Release Notes.) With the call interface enhancements come some compatibility considerations for static ERT main program (`ert_main.c`) files created before R2012a.

Compatibility Considerations

ERT Main Programs Now Include `rtmodel.h` Instead of `autobuild.h`

- In previous releases, GRT-based main programs such as `grt_main.c` and `grt_malloc_main.c` included `rtmodel.h` (which includes `model.h`) to access model-specific data structures and entry points. However, the static ERT main program `ert_main.c` included a different file, `autobuild.h`.

- Beginning in R2012a, GRT and static ERT main programs include `rtmodel.h`. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to include `rtmodel.h` instead of `autobuild.h`.

tid Argument to Model Step or Model Output/Update Function No Longer Generated As part of streamlining the model call interface, code generation no longer generates the *tid* argument to *model_step* or *model_output/model_update* functions in multirate, single-tasking models. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to remove the *tid* argument in model function calls.

firstTime Argument to Model Initialize Function No Longer Generated As part of streamlining the model call interface, code generation no longer generates the *firstTime* argument to the *model_initialize* function. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to remove the *firstTime* argument in *model_initialize* function calls.

Note The target configuration parameter `ERTFirstTimeCompliant` and the model configuration parameter `IncludeERTFirstTime` will be removed from the Embedded Coder software in a future release.

MAT-file Logging and External Mode Calls Moved from Model Code to Main Program As part of streamlining the model call interface, some MAT-file logging and External mode calls have been moved from the generated model code in `model.c` or `.cpp` to the main program code in `ert_main.c`. MAT-file logging and External mode calls are not heavily used in production code environments. However, if you have a static ERT main program created before R2012a that you want to use with R2012a generated code, and if you do want to support MAT-file logging or External mode, update the main program to add the MAT-file logging and External mode calls.

Changes for Embedded IDEs and Embedded Targets

- “Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE” on page 23-10
- “Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors” on page 23-10
- “Support Removed for Freescale MPC5xx” on page 23-11
- “Limitation: Parallel Builds Not Supported for Embedded Targets” on page 23-11

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

Embedded Coder software now supports version 4.4 of GCC 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.

Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors

You can now perform PIL simulation over a SCI interface with Texas Instruments C280x, C2802x, C2803x, C28x3x, c2834x processors. Previously, this capability was supported only for TI C28035 and C28335 processors.

Support Removed for Freescale MPC5xx

This release removes support for the Freescale MPC5xx processor family from the Embedded Coder product.

Attempting to generate code from models that contain blocks for Freescale MPC5xx hardware produces an error message.

Limitation: Parallel Builds Not Supported for Embedded 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 Embedded Targets.

New and Enhanced Demos

The following demos have been added in R2012a:

Demo...	Shows How You Can...
rtwdemo_roll_axis	Generate code for a roll axis autopilot control system. The <code>rtwdemo_roll</code> model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. <code>rtwdemo_roll</code> replaces <code>rtwdemo_f14</code> .
c28335_pmsmfoc_script	Schedule a multi-rate controller for a permanent magnet synchronous machine (PMSM) motor control application that runs on a Texas Instruments F28335 processor. To get this demo, use <code>targetinstaller</code> or <code>supportPackageInstaller</code> to install the <i>Embedded Coder Support Package for Texas Instruments C2000 Processors</i> .

The following demos have been enhanced in R2012a:

Demo...	Now...
coderdemo_crl	Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs).
rtwdemo_crl_script	<ul style="list-style-type: none"> Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs). Illustrates code replacement for Simulink matrix division and inversion operators.
rtwdemo_pmsmfoc_script	Added torque and position control modes to controller, parameterized motor and sensor data, and added support for specifying baud rate in example PIL implementation.
rtwdemo_radar	Shows how to simulate and generate code for the model <code>rtwdemo_eml_aero_radar</code> , which contains a MATLAB script.

Demo...	Now...
rtwdemo_configuration_set	Shows how to use the Code Generation Advisor and to automate the process of configuring a model for simulation and code generation.

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

R2011b

Version: 6.1

New Features

Bug Fixes

Compatibility Considerations

Static Code Metrics in Code Generation Report

The HTML code generation report now includes a static code metrics report. The static code metrics include: number of source code files, number of lines of code, list of global variables, functions in a call tree format, and the estimated stack size required for a function.

To generate the static code metrics report, on the **Code Generation > Report** pane of the Configuration Parameters dialog box, select the **Static code metrics** parameter and build your model. For more information, see Analyze Static Code Metrics of the Generated Code.

AUTOSAR Enhancements

Import and Export of AUTOSAR Sensor/Actuator Components

Embedded Coder now supports Sensor/Actuator Software Components. The key difference between a sensor/actuator component and an application component is that a sensor/actuator component can access the I/O hardware abstraction part within the ECU abstraction layer.

This support allows you to import sensor/actuator components, implement and test designs within Simulink, and export sensor/actuator components. For more information, see Use the Configure AUTOSAR Interface Dialog Box.

Improved Simulink Library Support for Multiple Runnables

Previously, Embedded Coder did not support the creation of multiple runnables from subsystems with links to Simulink library blocks. For example, you had to disable and break links to library blocks in order to configure and validate the subsystems as AUTOSAR runnables.

Now, the software supports the creation of multiple runnables when:

- The wrapper subsystem (containing function-call subsystems) is a link to a library block
- The function-call subsystems (within the wrapper subsystem) are links to library blocks

For more information, see Configure Multiple Runnables.

AUTOSAR Schema Version 3.2

The software now supports AUTOSAR schema version 3.2 (3.2.1). See Select an AUTOSAR Schema.

Export AUTOSAR XML as Single File

When you export an AUTOSAR Software Component, you can generate XML as either a set of files (default) or a single file. The latter option is new. For more information, see Use the Configure AUTOSAR Interface Dialog Box.

SIL and PIL Enhancements

R2011b supports the following enhancements for software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations.

Code Execution Profiling of Functions in Subsystems and Model Blocks

Previously, you could generate a profile of code execution times only for tasks within your generated code (for example, the step function for a sample rate). Now, you can also produce a profile of code

execution times for functions generated from atomic subsystems and model reference hierarchies within the top model. The software places instrumentation probes inside these functions and calculates execution times during a SIL or PIL simulation. At the end of the simulation, you can view an HTML report and analyze execution times within the MATLAB environment:

- The HTML report provides a summary of maximum and average execution times, which allows you to identify code that requires optimization
- The supplied APIs allow you to carry out further analysis of time measurements.

For more information, see Code Execution Profiling.

Code Coverage with LDRA Testbed

You can measure code coverage using the LDRA Testbed from LDRA Software Technology. For more information, see Code Coverage.

BitField and GetSet Custom Storage Classes

The software previously did not support the `BitField` and `GetSet` custom storage classes. Now, the software supports these custom storage classes for all types of SIL and PIL simulations, with one limitation. `GetSet` behavior for the SIL block is different from top-model SIL/PIL, Model block SIL/PIL, and PIL block:

- SIL block — The C definitions of the `Get` and `Set` functions that you provide form part of the algorithm under test.
- Other types of SIL/PIL — The SIL/PIL test harness automatically provides C definitions of the `Get` and `Set` functions that are used during SIL/PIL simulations. In addition, the software supports only *scalar* signals, parameters and global data stores.

For more information, see I/O Support and GetSet Custom Storage Class.

Model Blocks with Variable-Size Signals

You can run Model block SIL and PIL simulations where the Model block contains variable-size signals. On the **Simulation > Configuration Parameters > Model Referencing** pane, in the **Propagate sizes of variable-size signals** field, you must specify `During execution`. See I/O Support.

Verification of Generated C++ Code

Previously, support for C++ was restricted to simulations with the SIL block. Now, you can verify generated C++ code using all types of SIL and PIL:

- Top-model
- Model block
- SIL or PIL block

As before, only the SIL block supports C++ encapsulation. See Configuration Parameters Support.

Generate Multitasking Code for Concurrent Execution on Multicore Processors

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

See *Configuring Models for Targets with Multicore Processors*.

Changes for Embedded IDEs and Embedded Targets

- “64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0” on page 24-4
- “Support Added for Wind River VxWorks 6.8” on page 24-4
- “Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335” on page 24-5
- “New Target Function Library for Intel IPP/SSE (GNU)” on page 24-5
- “Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors” on page 24-5
- “Support Removed for Altium TASKING” on page 24-5
- “Support Removed for Infineon C166” on page 24-5
- “Support Ending for Green Hills MULTI in a Future Release” on page 24-6
- “Support Ending for Freescale MPC5xx in a Future Release” on page 24-6

64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0

Installing MATLAB & Simulink on a 64-bit Windows computer automatically installs the 64-bit versions of your MathWorks products, including Embedded Coder software. Now, you can use the 64-bit version of Embedded Coder software with the following 32-bit IDEs/tool chains:

- Texas Instruments Code Composer Studio 3.3
- Texas Instruments Code Composer Studio 4.0
- Analog Devices VisualDSP++ 5.0 (update 8)

Previously, you had to install the 32-bit versions of your MathWorks products to use Embedded Coder software with these IDEs.

For more information, see <https://www.mathworks.com/hardware-support/texas-instruments.html> and <https://www.mathworks.com/hardware-support/analog-devices.html>.

Also, check the Texas Instruments and Analog Devices Web sites for support information about using their tools on 64-bit Windows platforms.

Support Added for Wind River VxWorks 6.8

You can automatically generate and integrate code with the Wind River VxWorks 6.8 RTOS using makefiles via the XMakefiles feature. For more information, see *Choosing an XMakefile Configuration and Working with Wind River VxWorks RTOS*.

Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335

This release adds support for Serial Communication Interface (SCI) communications during processor-in-the-loop (PIL) simulations with Texas Instruments™ C28035 and C28335 microcontrollers. Using SCI for PIL simulations is much faster than using an IDE debugger for PIL.

For more information, see Serial Communication Interface (SCI) for Texas Instruments C2000, Example — Performing a Model Block PIL Simulation via SCI Using Makefiles, and the `fuelsys_pildemo`.

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.

For more information, see Code Replacement Library (CRL) and Embedded TargetsDesktop Targets.

Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors

This release adds support for the Single Instruction Multiple Data (SIMD) capabilities of the ARM Cortex-A8, ARM Cortex-A9 , and Intel processors. The use of SIMD instructions increases throughput compared to traditional Single Instruction Single Data (SISD) processing.

The following TFLs (code replacement libraries) optimize generated code for SIMD:

- GCC ARM Cortex-A8 — The GCC compiler and the ARM Cortex-A8 embedded processor
- GCC ARM Cortex-A9 — The GCC compiler and the ARM Cortex-A9 embedded processor
- Intel IPP/SSE (GNU) — The GCC compiler and the Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE)

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 SIMD capabilities, enable the corresponding TFLs as described in Code Replacement Library (CRL) and Embedded TargetsDesktop Targets.

Support Removed for Altium TASKING

Support for the Altium® TASKING IDE has been removed from the Embedded Coder product.

Support Removed for Infineon C166

Support for the Infineon® C166® processor family has been removed from the Embedded Coder product.

Support Ending for Green Hills MULTI in a Future Release

Support for the Green Hills MULTI IDE will end in a future release of the Embedded Coder product.

Support Ending for Freescale MPC5xx in a Future Release

Support for the Freescale MPC5xx processor family will end in a future release of the Embedded Coder product.

Saturation Control of Stateflow Data

A new property for Stateflow charts, **Saturate on integer overflow**, enables you to control the behavior of data with signed integer types when overflow occurs. This check box appears in the Chart properties dialog box.

Check Box	When to Use This Setting	Overflow Handling	Example of a Result
Selected	Overflow is possible for data in your Stateflow chart and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer saturates to -128 or +127.
Cleared	You want to optimize efficiency of the generated code.	The behavior depends on the C compiler you use for generating code.	The number 130 does not fit in a signed 8-bit integer and wraps to -126.

Arithmetic operations in the chart for which you can enable saturation protection are:

- Unary minus: $-a$
- Binary operations: $a + b$, $a - b$, $a * b$, a / b , $a \wedge b$
- Assignment operations: $a += b$, $a -= b$, $a *= b$, $a /= b$

For new charts, this check box is selected by default. When you open charts saved in previous releases, the check box is cleared to maintain backward compatibility.

For more information, see Handling Integer Overflow for Chart Data in the Stateflow User's Guide.

Custom Storage Class Properties for Managing Data Ownership and Definition

In R2011b, use the **Owner** and **Definition File** properties of custom storage classes to manage the definition and ownership of mpt data objects in generated code.

Previously, you could include the data definitions in generated code but could not specify the model that defined the data. Now, Embedded Coder creates the data definitions in the generated code according to the **Owner** property.

The **Owner** property of a custom storage class specifies the model that owns and defines the data in the generated code. The **Definition File** property specifies a name for the data definition file that Embedded Coder generates.

Compatibility Considerations

- If your legacy code exports data definitions to generated code and you now specify the **Owner** property, your generated code might have duplicate data definitions. This duplication causes a link error. In this case, remove the data definitions from the legacy code.
- If your legacy code does not export data definitions to generated code and you now specify the **Owner** property, your generated code might not contain data definitions. This mismatch causes a link error. In this case, add the missing data definitions to your legacy code.

Export Data Declarations to Shared Header File for Code Generation with Model Reference

When generating code with model reference, you can export shared data declarations to a specific header file in a shared directory.

Specify a data declaration header file in the following ways:

- For a data object: In the **Code generation** options section of the data object dialog
- For a model: In the **Code Generation > Code Placement** section of the **Configuration Parameters** dialog

Specify the option to use a **Shared location** in the field **Shared code placement** in **Code Generation > Interface** section of the **Configuration Parameters** dialog.

Target Function Library Code Replacement Enhancements

R2011b provides the following enhancements to code replacement using target function libraries (TFLs).

Code Replacement Tool for Creating and Managing TFL Tables

R2011b provides the Code Replacement Tool, which helps you create and manage the code replacement tables that make up a TFL. You can:

- Create a new code replacement table or import existing tables.
- Add, modify, and delete table entries. Each table entry represents a potential code replacement for a single function or operator. You can manage multiple tables together and copy and paste entries between tables.
- Validate tables and table entries.
- Save code replacement tables as MATLAB files.
- Generate the customization file you use to register your code replacement tables with code generation software.

Each code replacement table contains one or more table entries. Each table entry represents a potential replacement, during code generation, of a single function or operator by a custom implementation. For each table entry, you provide:

- **Mapping Information**, which relates a conceptual view of the function or operator (similar to the Simulink block view of the function or operator) to a custom implementation of that function or operator.

- **Build Information**, which provides header, source, or link information required for building the custom implementation.

You can open the Code Replacement Tool in the following ways:

- Go to the **Interface** pane of the Configuration Parameters dialog box and click the **Custom** button, which is located to the right of the **Target function library** parameter.
- Use the MATLAB command `crtool`.

For more information about creating code replacement tables for TFLs, see [Create and Manage Code Replacement Tables Using the Code Replacement Tool](#).

Ability to Align Data Objects to TFL-Specified Boundaries to Boost Code Performance

R2011b provides the ability to align data objects passed into a TFL replacement function to a specified boundary. This allows you to take advantage of target-specific function implementations that require data to be aligned in order to optimize application performance. To configure data alignment for a function implementation:

- 1 Specify the data alignment requirements in a TFL table entry. Alignment can be specified separately for each implementation function argument or collectively for all function arguments.
- 2 Specify the data alignment capabilities and syntax for one or more compilers, and include the alignment specifications in a TFL registry entry in an `sl_customization.m` or `rtwTargetInfo.m` file.

For more information on specifying data alignment requirements and compiler alignment attributes, see [Configure Data Alignment for Function Implementations](#).

For additional examples of configuring data alignment for function implementations, see the demo `rtwdemo_tfl_script`.

Support for Replacing Element-wise Matrix Multiply

TFLs support several nonscalar operators for replacement with custom library functions in generated model code. R2011b adds support for replacing element-wise matrix multiplication operations (`.*` operator in element-wise mode) with custom implementations. For more information, see [Map Nonscalar Operators to Target-Specific Implementations](#).

Code Generation Enhancements

Redundant Condition Checks

Multiple checks of the same condition are difficult to avoid in modeling. For example, a common modeling pattern is Switch blocks sharing the same condition check. Previously, the generated code for multiple Switch blocks produced multiple `if` statements.

```
if (cond) {
    true_statement1;
} else {
    false_statement1; }
if (cond) {
    true_statement2;
} else {
    false_statement2;
}
```

In R2011b, the generated code combines these condition checks. For example, the generated code for Switch blocks with a common condition combines these multiple `if` statements.

```
if (cond) {
    true_statement1;
    true_statement2;
}
else {
    false_statement1;
    false_statement2;
}
```

This optimization reduces code size and execution time. As a result, other optimizations for condition expressions or merged branches are enabled which reduce data copies and RAM usage.

Loop Fusion

R2011b provides more precise data dependency analysis of the data and signals of a nested Simulink bus. This enhancement enables more loop fusion in the generated code which reduces code execution time and ROM, and improves code readability.

Invariant Condition Check Lifting

When a condition check is invariant to the enclosing loop and you specify loops to be unrolled, the code generator lifts the check out of the loop. This enhancement reduces ROM, enables additional optimizations, and improves execution speed and code readability. For more information on loop unrolling, see [Configure Loop Unrolling Threshold](#).

Parameter Pooling for Stateflow and Interpreted MATLAB Function Blocks

Parameter pooling now occurs for Simulink matrix constants used as Stateflow graphical function arguments. This enhancement reduces RAM and ROM, and improves thread safety.

Readability Improvement for Reusable Subsystem Input and Output

The generated code for reusable subsystem input and output now eliminates redundant operators and unnecessary parentheses. This enhancement improves code readability.

Enhanced Code Generation Optimization Using Minimum and Maximum Values

The **Optimize using specified minimum and maximum values** code generation option now takes into account the minimum and maximum values specified for `Simulink.Parameter` objects even if the object is part of an expression. For example, consider a Gain block with a gain parameter specified as an expression such as `k1 + 5`, where `k1` is a `Simulink.Parameter` object with `k1.min = -10` and `k1.max = 10`. If minimum and maximum values of the parameter specified in the parameter dialog box are 0 and 20, the range calculated for this parameter expression is 0 to 15.

For more information, see [Optimize Generated Code Using Specified Minimum and Maximum Values](#).

New Model Advisor Check for Code Efficiency of Logic Blocks

The Simulink Model Advisor includes the following new check for code efficiency of logic blocks: Check output types of logic blocks. The following blocks in the Simulink Logic and Bit Operations library can use boolean or another setting for the output data type:

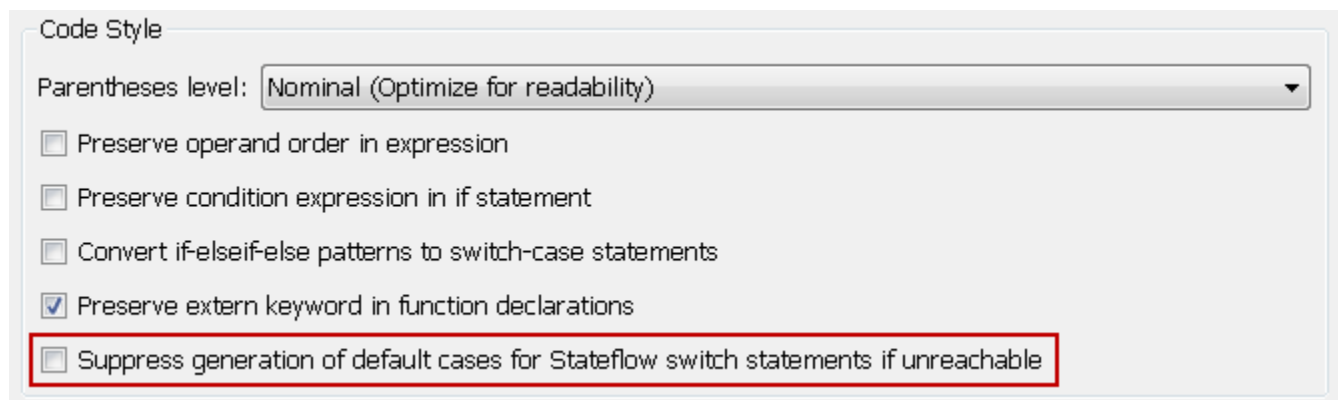
- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

Running this Model Advisor check helps you identify logic blocks that do not use boolean for the output data type.

For more information about the Model Advisor, see Consulting the Model Advisor.

Control of Default Case Generation for Switch Statements in Generated Code for Stateflow Charts

You can specify whether or not to generate default cases for switch statements in the generated code for Stateflow charts. This optimization works on a per-model basis and applies to the code generated for a state that has multiple substates. Use the following check box on the **Code Generation > Code Style** pane of the Configuration Parameters dialog box:



Check Box	When to Use This Setting	Format of Switch Statements
Selected	Provide better code coverage by checking that every branch in the generated code is falsifiable.	Exclude the default case when it is unreachable.
Cleared	Check for MISRA C compliance and provide a fallback in case of RAM corruption.	Include a default case.

For new models, this check box is cleared by default. When you open models saved in previous releases, the check box is also cleared to maintain backward compatibility.

For more information, see Code Generation Pane: Code Style in the Embedded Coder Reference documentation.

Improvement to Build Process for Conflicting Identifiers

Previously, if your model contained two referenced models with the same input (or output) port names, the model might not build because of potentially conflicting identifiers. The failure to build happens when the generated identifiers exceed the Maximum identifier length. In R2011b, the build process is improved to handle more cases when two referenced models have the same input (or output) port names. For more information, see Model Referencing Considerations.

Update to Code Generation Verification Class `cgv.Config`

Compatibility Considerations

The Connectivity `cgv.Config` parameter has the following updates:

- `pil` replaces the `custom` value. In R2011b, you can use `custom` without producing a warning or error message.
- The `tasking` value is not available. Specifying `tasking` produces an error.

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...
<code>rtwdemo_pmsmfoc_script</code>	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine
<code>rtwdemo_sil_pil_script</code>	Incorporates code execution profiling
<code>rtwdemo_tfl_script</code>	Shows how you can align nonscalar data passed into a target function library (TFL) code replacement function
<code>fuelsys_pil</code>	Incorporates using serial communication interface to communicate during PIL simulation

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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: 6.0

New Features

Bug Fixes

Compatibility Considerations

Coder Product Restructuring

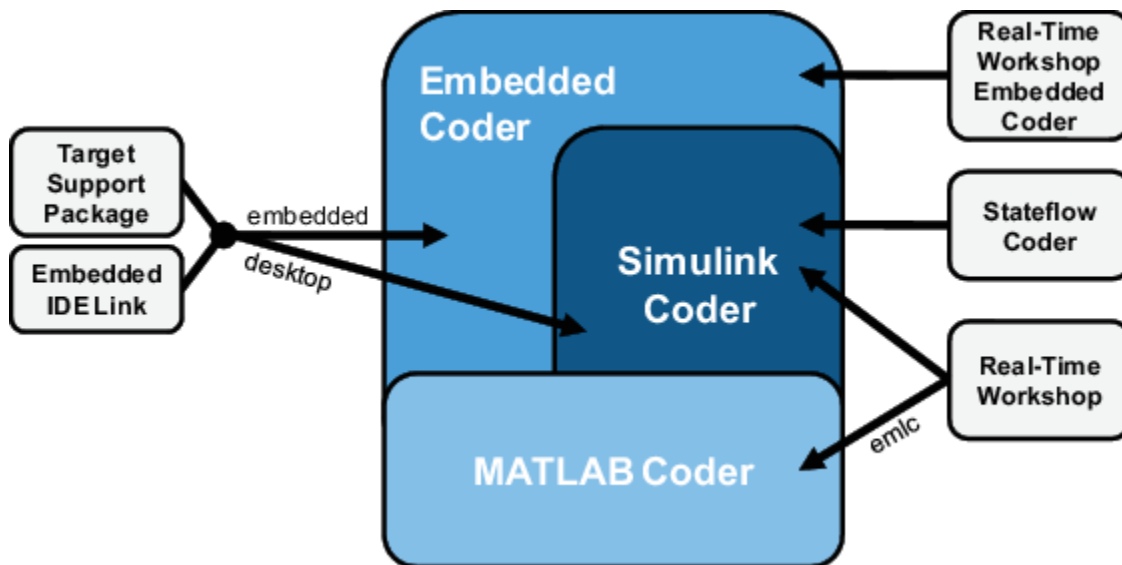
- “Product Restructuring Overview” on page 25-2
- “Resources for Upgrading from Real-Time Workshop Embedded Coder” on page 25-2
- “Migration of Embedded MATLAB Coder Features to MATLAB Coder” on page 25-3
- “Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder” on page 25-3
- “Interface Changes Related to Product Restructuring” on page 25-4
- “Simulink Graphical User Interface Changes” on page 25-4

Product Restructuring Overview

In R2011a, the Embedded Coder product replaces the Real-Time Workshop® Embedded Coder product. Additionally,

- The Simulink Coder product combines and replaces the Real-Time Workshop and Stateflow Coder products
- 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 Embedded IDE Link™ and Target Support Package™ products 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.



Resources for Upgrading from Real-Time Workshop Embedded Coder

If you are upgrading to Embedded Coder from Real-Time Workshop Embedded Coder, review information about compatibility and upgrade issues at the following locations:

- *Release Notes for Embedded Coder* (latest release), “Compatibility Summary” section

- On the MathWorks web site, in the Archived documentation, select R2010b, and view the following tables, which are provided in the release notes for Real-Time Workshop Embedded Coder: *Compatibility Summary for Real-Time Workshop Embedded Coder Software*:

This table provides 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 and view the corresponding tables for Embedded IDE Link or Target Support Package:
 - *Compatibility Summary for Embedded IDE Link (R2010b)*
 - *Compatibility Summary for Target Support Package (R2010b)*

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 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 *Generating C/C++ Code from MATLAB Code*.

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 following table summarizes the transition of the Embedded IDE Link and Target Support Package supported hardware and software 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
	Texas Instruments C6000		x
	Linux OS	x	x
	Windows OS	x	
	VxWorks RTOS		x

Interface Changes Related to Product Restructuring

You will see interface changes as part of restructuring the Coder products.

- 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, changes to code generation related elements
- In error messages, tool tips, demos, and product documentation, references to Real-Time Workshop Embedded Coder, Real-Time Workshop, and Stateflow Coder and related terms are 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
Subsystem Parameter dialog box	Following parameters on main pane: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Function packaging • Function name options • Function name • File name options • File name (no extension)

Compatibility Considerations

In the Help browser **Contents** pane, Embedded Coder is now listed with the products for MATLAB, because Embedded Coder now supports both MATLAB Coder and Simulink Coder workflows.

Data Management Enhancements and Changes

- “Memory Section Enhancements” on page 25-5
- “No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects” on page 25-5
- “Parts of Data Class Infrastructure Not Available” on page 25-5
- “No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer” on page 25-6

- “Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release” on page 25-6

Memory Section Enhancements

- Pragmas are now added to data and function declarations (prior to R2011a they were added to definitions only); at compile time, this makes the compiler aware of memory locations for functions and data, potentially optimizing generated code
- New function category is available for shared utilities on the **Code Generation > Memory Sections** pane: Shared utility
- Referenced models can have a memory section that is different from that of the top model for the InitTerm and Execute function categories

No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects

You cannot set the RTWInfo or CustomAttributes property of a Simulink data object from the MATLAB Command Window or a MATLAB script. Attempts to set these properties generate an error.

Although you cannot set RTWInfo or CustomAttributes, you can still set subproperties of RTWInfo and CustomAttributes.

Compatibility Considerations

Operations from the MATLAB Command Window or a MATLAB script, which set the data object property RTWInfo or CustomAttributes, generate an error.

For example, a MATLAB script might set these properties by copying a data object as shown below:

```
a = Simulink.Parameter;
b = Simulink.Parameter;
b.RTWInfo = a.RTWInfo;
b.RTWInfo.CustomAttributes = a.RTWInfo.CustomAttributes;
.
.
.
```

To copy a data object, use the object's deepCopy method.

```
a = Simulink.Parameter;
b = a.deepCopy;
.
.
.
```

Parts of Data Class Infrastructure Not Available

Simulink has been generating warnings for usage of the following data class infrastructure features for several releases. As of R2011a, the features are not supported.

- Custom storage classes not captured in the custom storage class registration file (csc_registration) - *warning displayed since R14SP2*
- Built-in custom data class attributes BitFieldName and FileName+IncludeDelimiter - *warning displayed since R2008b*

Instead of...	Use...
BitFieldName	StructName
FileName+IncludeDelimiter	HeaderFile

- Initial value of MPT data objects inside `mpt.CustomRTWInfoSignal` - *warning displayed since R2006a*

Compatibility Considerations

- When you use a removed feature, Simulink now generates an error.
- When loading a MAT-file that uses an unsupported feature, the load operation suppresses the generated error such that it is not visible. In addition, MATLAB silently deletes data that had been associated with the unsupported feature. To prevent loss of data when loading a MAT-file, load and resave the file with R2010b or earlier.

No Longer Generating Pragma for Data Defined with Built-In Storage Class `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer`

The code generator no longer generates a pragma around definitions or declarations for data that has the following built-in storage classes:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

Prior to R2011a, based on model configuration parameters for specifying memory sections and the built-in storage class defined for data, the code generator would do the following:

For Built-In Storage Class...	Generate pragma Around...
<code>ExportedGlobal</code>	Data definition and declaration
<code>ImportedExtern</code>	Data declaration
<code>ImportedExternPointer</code>	Data declaration

The code generator now treats data with these built-in storage classes like custom storage classes with no memory section specified.

Compatibility Considerations

To work around this change, select a custom storage class that uses the memory section of interest for the data.

Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release

In a future release, data classes `Simulink.CustomParameter` and `Simulink.CustomSignal` will no longer be supported because they are equivalent to `Simulink.Parameter` and `Simulink.Signal`.

Compatibility Considerations

If you use the data class `Simulink.CustomParameter` or `Simulink.CustomSignal`, Simulink posts a warning that identifies the class and describes one or more techniques for eliminating it. You can ignore these warnings in R2011a, but consider making the described changes now because the classes will be removed in a future release.

AUTOSAR Enhancements

The following enhancements are available in R2011a.

Calibration Parameters

Previously, the software supported only calibration parameters that were defined by a calibration component. These parameters could be accessed by all AUTOSAR Software Components. The AUTOSAR standard also specifies an internal calibration parameter that is defined and accessed by only one AUTOSAR Software Component. The software now supports:

- AUTOSAR internal calibration parameters, including the import and export of initial values of these parameters.
- A bus object data type (AUTOSAR record type) to import and export both kinds of calibration parameters.

For more information, see [Calibration Parameters](#) and [Configure Calibration Parameters](#).

Multiple Runnables from Virtual Subsystems

Previously, if a wrapper subsystem had virtual subsystems containing function-call subsystems, you could not export the function-call subsystems as AUTOSAR runnables from the wrapper subsystem level. Now, within a wrapper subsystem, you can group function-call subsystems into virtual subsystems and generate runnables for these function-call subsystems. See [Configure Multiple Runnables](#) and [Export AUTOSAR Software Component](#).

Support for Code Descriptor Elements

The AUTOSAR standard specifies that the XML description of an AUTOSAR Software Component implementation must contain code descriptor elements to describe generated source files and include header files. This feature allows AUTOSAR authoring tools that import software components to automate the building process for source code.

Previously, the software did not generate the software component implementation file (`modelname_implementation.arxml`) with these code descriptor elements. Now, when you build a Simulink model for an AUTOSAR target, the software generates a `CODE-DESCRIPTORS` element within the `SWC_IMPLEMENTATION` element. The `CODE-DESCRIPTORS` element contains `XFILE` elements that provide descriptions of the generated code.

For example, if you build the model `rtwdemo_autosar_counter`, the generated file `rtwdemo_autosar_counter_implementation.arxml` has the following `SWC_IMPLEMENTATION` element:

```
....
<SWC-IMPLEMENTATION>
  <SHORT-NAME>rtwdemo_autosar_counter</SHORT-NAME>
  <CODE-DESCRIPTORS>
    <CODE>
```

```

<SHORT-NAME>Code</SHORT-NAME>
<TYPE>SRC</TYPE>
<XFILES>
  <XFILE>
    <SHORT-NAME>rtwdemo_autosar_counter_c</SHORT-NAME>
    <CATEGORY>GeneratedFile</CATEGORY>
    <URL>rtwdemo_autosar_counter_autosar_rtw\rtwdemo_autosar_counter.c</URL>
    <TOOL>Embedded Coder</TOOL>
    <TOOL-VERSION>5.6</TOOL-VERSION>
  </XFILE>
  <XFILE>
    <SHORT-NAME>rtwdemo_autosar_counter_h</SHORT-NAME>
    <CATEGORY>GeneratedFile</CATEGORY>
    <URL>rtwdemo_autosar_counter_autosar_rtw\rtwdemo_autosar_counter.h</URL>
    <TOOL>Embedded Coder</TOOL>
    <TOOL-VERSION>5.6</TOOL-VERSION>
  </XFILE>
  . . .
</XFILES>
</CODE>
</CODE-DESCRIPTORS>
<CODE-GENERATOR>Embedded Coder 5.6 (R2011a) 26-Aug-2010</CODE-GENERATOR>
<PROGRAMMING-LANGUAGE>C</PROGRAMMING-LANGUAGE>
</SWC-IMPLEMENTATION>
. . . .

```

SIL and PIL Enhancements

Code Execution Profiling

You can collect execution time measurements in a specified base workspace variable during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. At the end of the simulation, you can view or analyze the measurements within the MATLAB environment. This feature allows you to collect an execution time profile for each task within your generated code.

The software supports code execution profiling for all types of SIL or PIL simulations except the SIL block.

For more information, see Code Execution Profiling.

PIL Block Parameter Tuning

R2011a supports parameter tuning for the PIL block, which allows you to change tunable workspace parameters between or during simulations without regenerating code. This feature also includes support for tunable structure parameters. For more information, see I/O Support and Tunable Parameters and SIL/PIL.

Top-Model SIL/PIL and PIL Block Parameter Initialization

R2011a supports automatic definition and initialization of parameters with imported storage classes. For more information, see I/O Support and Imported Data Definitions.

Model Block Parameter Tuning and Model Initialization

Previously, the software did not support the following features for Model block SIL/PIL:

- Simplified initialization mode
- Tunable structure parameters

R2011a now supports these features. For more information, see Configuration Parameters Support, I/O Support, and Tunable Parameters and SIL/PIL.

Code Generation Enhancements

Improved Code for Data Store Memory In-place Assignment

Previously, the generated code for a Data Store Memory block used data copies to perform data store assignments. The generated code now eliminates the data copies and performs an in-place assignment. This improvement generates less code, uses less memory, and provides faster execution.

Improvements to Target Function Library Replacements

Enhancements to Target Function Library Replacements (TFL) include:

- If multiple TFL replacements occur within a function, temporary variables are now reused instead of creating extra temporary variables. This enhancement reduces the stack size during TFL replacement.
- During TFL replacement, if unnecessary temporary variables are introduced when block output is not the returned value of the function but one of the input arguments, code generation now removes the temporary variable. This enhancement improves execution speed and requires less memory.

For more information, see Introduction to Code Replacement Libraries.

Improved Loop Fusion

Code generation now includes the following:

- An improved loop fusion algorithm that reduces data copies. This enhancement decreases stack size, ROM consumption, and code generation time.
- Selectively fuses loops when the loop count is larger than the Loop unrolling threshold. In these cases, loop unrolling allows the code generator to perform more optimizations. In addition, the code generator groups the statements together to assign values to the elements of a signal or parameter array, which improves data access and code readability.

Improved Array Indexing

The generated code is optimized for more efficient array indexing. When a complex instruction is used repeatedly in an array index, the instruction is replaced with a temporary variable to perform the calculation more efficiently. This enhancement improves execution speed and reduces code size.

Improvement on Matrix Parameter Pooling

For matrix parameters with the same flattened value, the generated code now pools the matrix parameters even when they have different shapes. This enhancement reduces ROM consumption.

Readability Improvements Involving Data References

For references to the root inport and outport, as well as DWork, unnecessary parentheses are removed from the generated code. This enhancement produces more readable code.

Code Generation Verification (CGV) API Updates

Support for Adding Multiple Callback Functions

In R2011a, the `cgv.CGV` class includes new methods to add callback functions. These methods replace the `cgv.CGV.addCallback` method which added only a pre-execution callback function. Now, the new methods allow CGV to invoke callback functions at several stages of the `cgv.CGV.run` execution. The new methods are:

- `cgv.CGV.addHeaderReportFcn` adds a callback function invoked before executing input data in the `cgv.CGV` object.
- `cgv.CGV.addPreExecReportFcn` adds a callback function invoked before executing each input data file in the `cgv.CGV` object.
- `cgv.CGV.addPreExecFcn` adds a callback function invoked before executing each input data file in the `cgv.CGV` object.
- `cgv.CGV.addPostExecReportFcn` adds a callback function invoked after executing each input data file in the `cgv.CGV` object.
- `cgv.CGV.addPostExecFcn` adds a callback function invoked after executing each input data file in the `cgv.CGV` object.
- `cgv.CGV.addTrailerReportFcn` adds a callback function invoked after executing input data in the `cgv.CGV` object.

New Functionality Added to the `cgv.CGV` Class

The `cgv.CGV` class now includes the following methods:

- `cgv.CGV.activateConfigSet` activates the configuration set of a model.
- `cgv.CGV.addBaseline` adds a file of baseline data for comparison.
- `cgv.CGV.copySetup` creates a copy of a `cgv.CGV` object.
- `cgv.CGV.setMode` specifies the mode of execution (`sim`, `sil`, or `pil`).
- `cgv.CGV.copySetup` returns the status of the execution of the `cgv.CGV` object.

The `cgv.CGV` class now includes the following properties:

- Name
- Description

Compatibility Considerations

Previously, the `cgv.CGV` class included parameters that you set to perform automatic configuration checks of your model. In R2011a, `cgv.CGV` class does not perform automatic configuration checks. Instead, you can use the `cgv.Config` class to perform a manual configuration check of your model. Before calling `cgv.CGV.run`, perform a manual configuration check of your model. Otherwise, an error might occur later in the process. For more information, see Programmatic Code Generation Verification.

Changes to the `cgv.CGV` class parameters are listed in the following table.

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
LogMode removed from <code>cgv.CGV</code>	Errors	LogMode parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the LogMode parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.
Processor removed from <code>cgv.CGV</code>	Errors	Processor parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the Processor parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.
SaveModel removed from <code>cgv.CGV</code>	Errors	SaveModel parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the SaveModel parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.
ConfigModel removed from <code>cgv.CGV</code>	Warns if set to off Errors if set to on	<code>cgv.Config.configModel</code> method	To check your model before running CGV, replace the <code>cgv.CGVConfigModel</code> parameter with a call to the <code>cgv.Config.configModel</code> method
CheckInterface parameter from <code>cgv.CGV</code>	Warns if set to off Errors if set to on	CheckOutputs parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the CheckOutputs parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
tasking and custom values removed from the Connectivity parameter of <code>cgv.CGV</code>	Errors	<code>pil</code> , a new value for the <code>cgv.CGV</code> Connectivity parameter	Replace calls to the <code>cgv.CGV</code> constructor using the parameter-value arguments, ('Connectivity', 'tasking') or ('Connectivity', 'custom'), with ('Connectivity', 'pil').

Changes to the `cgv.Config` class parameters are listed in the following table:

Parameter	What Happens When You Use Parameter?	Compatibility Considerations
CheckOutports parameter added to <code>cgv.Config</code>	Defaults to on. Compiles the model. Then checks that the model output configuration is compatible with the <code>cgv.CGV</code> object.	If your script fixes errors reported by <code>cgv.Config</code> , you can set CheckOutports to off.
LogMode parameter from <code>cgv.Config</code>	Change in behavior	If you do not give a value for LogMode, logging changes are not made to the configuration parameters.

MISRA-C Code Generation Objective

The Code Generation Advisor now includes a new objective for MISRA-C:2004 guidelines. To set the new objective, open the Configuration Parameters dialog box and select the **Code Generation** pane. In the Code Generation Advisor section, click the **Set objectives** button to open the Code Generation Advisor dialog box. In the **Available objectives** list, select MISRA-C:2004 guidelines and click the select button (arrow pointing right) to move the objective to the **Selected objectives** list. For more information on setting objectives, see Application Objectives.

New Model Advisor Check for Code Efficiency of Lookup Table Blocks

The Simulink Model Advisor includes the following new check for code efficiency of lookup table blocks: Identify lookup table blocks that generate expensive out-of-range checking code. By default, the following blocks generate code that checks for out-of-range breakpoint inputs:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

Similarly, the Interpolation Using Prelookup block generates code that checks for out-of-range index inputs. Running this Model Advisor check helps you identify lookup table blocks that generate out-of-range checking code for breakpoint or index inputs.

For more information about the Model Advisor, see [Consulting the Model Advisor](#).

Enhanced Code Generation Optimization

The **Optimize using specified minimum and maximum values** code generation option now takes into account the minimum and maximum values specified for:

- A `Simulink.Parameter` object provided that it is used on its own. It does not use these minimum and maximum values if the object is part of an expression. For example, if a Gain block has a gain parameter specified as `K1`, where `K1` is defined as a `Simulink.Parameter` object in the base workspace, the optimization takes the minimum and maximum values of `K1` into account. However, if the Gain block has a gain parameter of `K1+5` or `K1+K2+K3`, where `K2` and `K3` are also `Simulink.Parameter` objects, the optimization does not use the minimum and maximum values of `K1`, `K2` or `K3`.
- Design ranges specified on block outputs in a conditionally-executed subsystem, except for the block outputs that are directly connected to an Output block.

For more information, see [Optimize Generated Code Using Specified Minimum and Maximum Values](#).

Target Function Library Replacement Based on Computation Method for Reciprocal Sqrt, Sine, and Cosine

Target function libraries (TFLs) now support the ability to control replacement of certain math functions using their computation method as a distinguishing attribute. For example,

- The `rSqrt` block can be configured to use either of two computation methods, `Newton-Raphson` or `Exact`.
- The `Trigonometric Function` block, with **Function** set to `sin` or `cos`, can be configured to use either of two approximation methods, `CORDIC` or `None`.

You can configure TFL table entries to replace these functions for one or all of the available computation methods. For example, you could replace only `Newton-Raphson` instances of the `rSqrt` function.

For more information, see [Replace Math Functions Based on Computation Method](#).

Target Function Library Support for abs, min, max, and sign functions

Embedded Coder software now supports target function library customization control for fixed-point `abs`, `min`, `max`, and `sign` functions.

For more information, see [Register Code Replacement Libraries](#).

C++ Encapsulation Allowed for Referenced Models in For Each Subsystems

In previous releases, due to a code generation limitation, code could not be generated for a `For Each Subsystem` block under the following conditions:

- The `For Each Subsystem` block directly or indirectly contains a `Model` block.

- The Model block references a model for which C++ encapsulation is selected.

R2011a removes this limitation. You can now generate code for a For Each Subsystem in which a referenced model uses C++ encapsulation.

Improved Code Generation for Portable Word Sizes

In the software-in-the-loop (SIL) simulation work flow, the model option **Enable portable word sizes** allows you to take code intended for a specific target platform and compile and run the same code on a MATLAB host platform that uses different processor word sizes. R2011a enhances the code generated for portable word sizes by inserting explicit casts to help protect against integral promotion differences and other behavior differences between host and target. This potentially can reduce the incidence of numerical differences due to host/target behavior differences. For more information, see *Configure Hardware Implementation Settings for SIL and Portable Word Sizes Limitations*.

Improved Comments in the Generated Code

R2011a provides improvements to comment generation for better readability and understanding of the generated code. Specifically, comments are located closer to the referring code and reflect the intent of the code. An end comment is now included at the end of a control flow block of code. For information on customizing comments in the generated code, see *Configure Code Comments in Embedded System Code*.

Replacement Data Types and Simulation Mode for Referenced Models

To replace built-in data type names with user-defined data type names in the generated code for a referenced model, you must set the **Simulation mode** parameter for the Model block to one of the following:

- Normal
- Software-in-the-loop (SIL)
- Processor-in-the-loop (PIL)

For more information, see *Data Types and Referenced Model Simulation Modes*.

Changes for Embedded IDEs and Embedded Targets

- “Feature Support for Embedded IDEs and Embedded Targets” on page 25-15
- “Execution Profiling during PIL Simulation” on page 25-15
- “Location of Blocks for Embedded Targets” on page 25-15
- “Location of Demos for Embedded IDEs and Embedded Targets” on page 25-16
- “Multicore Deployment with Rate-Based Multithreading” on page 25-17
- “Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)” on page 25-17
- “Changes to Frame-Based Processing” on page 25-17
- “New Support for Analog Devices Blackfin BF50x and BF51x Processors” on page 25-18

- “Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors” on page 25-19
- “Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI” on page 25-19
- “Support for Texas Instruments Delfino C2834x Processors” on page 25-19
- “Ending Support for Altium TASKING in a Future Release” on page 25-20
- “Ending Support for Freescale MPC5xx in a Future Release” on page 25-20
- “Ending Support for Infineon C166 in a Future Release” on page 25-20
- “Removed Methods and Arguments” on page 25-20

Feature Support for Embedded IDEs and Embedded Targets

The Embedded Coder software provides the following features as implemented in the former Target Support Package and former Embedded IDE Link products:

- Automation Interface
- Processor-in-the-Loop (PIL) Simulation
- Execution Profiling
- Execution Profiling during PIL Simulation
- Stack Profiler
- External Mode
- Schedulers and Timing
- Makefile Generation (XMakefile)
- Target Function Library (TFL) Optimization
- Multicore Deployment for Rate Based Multithreading

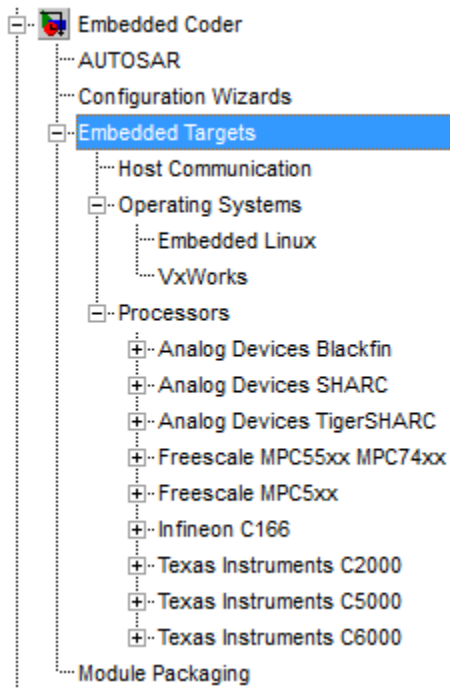
Note You can only use these features in the 32-bit version of your MathWorks products. To use these features on 64-bit hardware, install and run the 32-bit versions of your MathWorks products.

Execution Profiling during PIL Simulation

During Processor-in-the-loop (PIL) simulation, you can profile synchronous tasks in code running on the target. For more information, see Execution Profiling during PIL Simulation

Location of Blocks for Embedded Targets

Blocks from the former Target Support Package product and Embedded IDE Link product now reside under Embedded Coder in the Embedded Targets block library, as shown.

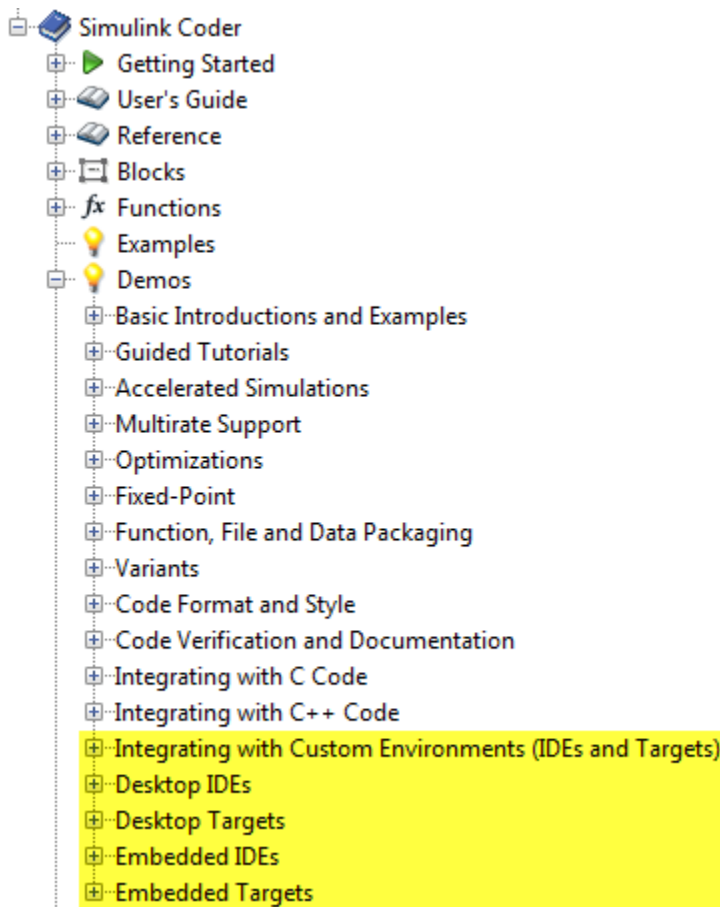


Embedded Targets includes the following types of blocks:

- Host Communication
- Operating Systems
 - Embedded Linux
 - VxWorks
- Processors
 - Analog Devices Blackfin
 - Analog Devices SHARC
 - Analog Devices TigerSHARC
 - Freescale MPC5xx MPC74xx
 - Freescale MPC5xx
 - Infineon C166
 - Texas Instruments C2000
 - Texas Instruments C5000
 - Texas Instruments C6000

Location of Demos for Embedded IDEs and Embedded 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.



Multicore Deployment with Rate-Based Multithreading

You can deploy rate-based multithreading applications to multicore processors running Embedded Linux and

VxWorks. This feature improves performance by taking advantage of multicore hardware resources.

Also see the [Running Target Applications on Multicore Processors user's guide](#) topic.

Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)

You can generate a makefile project on a Windows host machine, transfer the makefile project to an remote target running Linux, such as a BeagleBoard, and then build the executable on the remote target.

Changes to Frame-Based Processing

Signal processing applications often process sequential samples of data at once as a group, rather than one sample at a time. MathWorks documentation refers to the former as frame-based processing and the latter as sample-based processing. A frame is a collection of samples of data, sequential in time. To perform frame-based processing in MathWorks products, you must have a DSP System Toolbox license.

Historically, Simulink-family products that can perform frame-based processing propagate frame-based signals throughout a model. The frame status is an attribute of the signals in a model, just as

data type and dimensions are attributes of a signal. The Simulink engine propagates the frame attribute of a signal with a frame bit, which can either be on or off. When the frame bit is on, Simulink interprets the signal as frame-based, and displays it as a double line, rather than as a single line.

Beginning in R2010b, MathWorks started to change the handling of frame-based processing significantly. In the future, signal attributes will not include frame status. Instead, individual blocks will control whether they treat data inputs as frames or as samples.

To transition to this new paradigm, blocks that can perform sample- and frame-based processing contain a new **Input processing** parameter that specifies the processing behavior. You can set **Input processing** to `Columns as channels` (frame based) or `Elements as channels` (sample based). The third option, `Inherited` (this choice will be removed - see release notes), is a temporary selection. This third option helps you migrate your existing models from the old paradigm to the new paradigm.

In R2011a, the following Embedded Coder blocks received a new **Input processing** parameter:

- C62X Real Forward Lattice All-Pole IIR
- C62X Complex FIR
- C62X General Real FIR
- C62X Real IIR
- C64X Real Forward Lattice All-Pole IIR

Compatibility Considerations

When you load an existing model in R2011a, blocks with the new **Input processing** parameter shows a setting of `Inherited` (this choice will be removed - see release notes). This setting enables your existing models to work as expected until you upgrade them. Upgrade your models as soon as possible.

To upgrade your existing models, use the `supdate` function. This function detects blocks that have **Input processing** set to `Inherited` (this choice will be removed - see release notes). The function asks you whether to upgrade each block. If you select yes, the function detects the status of the frame bit on the input port of the block. If the frame bit is 1 (frames), the function sets the **Input processing** parameter to `Columns as channels` (frame based). If the bit is 0 (samples), the function sets the parameter to `Elements as channels` (sample based).

A future release will remove the frame bit and the `Inherited` (this choice will be removed - see release notes) option. At that time, if you have not updated the model, the software automatically sets the **Input processing** parameter. The software uses the library default setting of the block to select either `Columns as channels` (frame based) or `Elements as channels` (sample based). If the library default setting does not match the parameter setting in your model, your model will produce unexpected results. Additionally, after the removal of the frame bit, you will no longer be able to upgrade your models using the `supdate` function. Therefore, upgrade your existing models using `supdate` as soon as possible.

New Support for Analog Devices Blackfin BF50x and BF51x Processors

You can now generate code for the following embedded processors when you use Embedded Coder software:

- BF504

- BF504F
- BF506F
- BF512
- BF514
- BF516
- BF518

Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors

You can use new Target Function Libraries (TFLs) to generate efficient fixed-point code for the ARM Cortex-M3, Cortex-A8, and Cortex-A9 processors. These TFLs include GCC compiler extensions and intrinsic functions that optimize the code Embedded Coder generates for these processors.

Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI

Support for Green Hills MULTI software now includes versions 5.0.6 and 5.1.6.

Support for Texas Instruments Delfino C2834x Processors

You can now generate code for the following embedded processors when you use Embedded Coder software with Texas Instruments Code Composer Studio software:

- C28341
- C28342
- C28343
- C28344
- C28345
- C28346

The new C2834x (c2834xlib) block library contains the following blocks:

- C2000 CAN Calibration Protocol
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger
- C28x Watchdog
- C280x/C2803x/C28x3x/c2834x eCAN Receive
- C280x/C2803x/C28x3x/c2834x eCAN Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x eCAP

- C280x/C2802x/C2803x/C28x3x/c2834x ePWM
- C280x/C2803x/C28x3x/c2834x eQEP

Ending Support for Altium TASKING in a Future Release

Support for the Altium TASKING IDE will end in a future release of the Embedded Coder product.

Ending Support for Freescale MPC5xx in a Future Release

Support for the Freescale MPC5xx processor family will end in a future release of the Embedded Coder product.

Ending Support for Infineon C166 in a Future Release

Support for the Infineon C166 processor family will end in a future release of the Embedded Coder product.

Removed Methods and Arguments

Deprecated the `type` property for the Code Composer Studio IDE object. For example, entering the following text generates an error message:

```
infolist = IDE_Obj.list(type)
```

Changes to ver Function Product Arguments

The following changes have been made to `ver` function arguments related to embedded code generation products:

- The new argument `'embeddedcoder'` returns information about the installed version of the Embedded Coder product.
- The argument `'ecoder'`, which previously returned information about the installed version of the Real-Time Workshop Embedded 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 `'ecoder'` argument, update the script appropriately. For example, you can update the `ver` call to use the `'embeddedcoder'` argument.

New and Enhanced Demos

The following demos have been added in R2011a:

Demo...	Shows How You Can...
<code>coderdemo_tfl</code>	Use target function libraries (TFLs) to replace operators and functions in code generated by MATLAB Coder.

Demo...	Shows How You Can...
rtwdemo_code_coverage_script	Generate model coverage and code coverage reports, and use these reports to compare model coverage and code coverage results for parts of a model.
rtwdemo_pmsmfoc_script	Perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine.

The following demos have been enhanced in R2011a:

Demo...	Now...
vipstabilize_fixpt_beagleboard	Uses 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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.