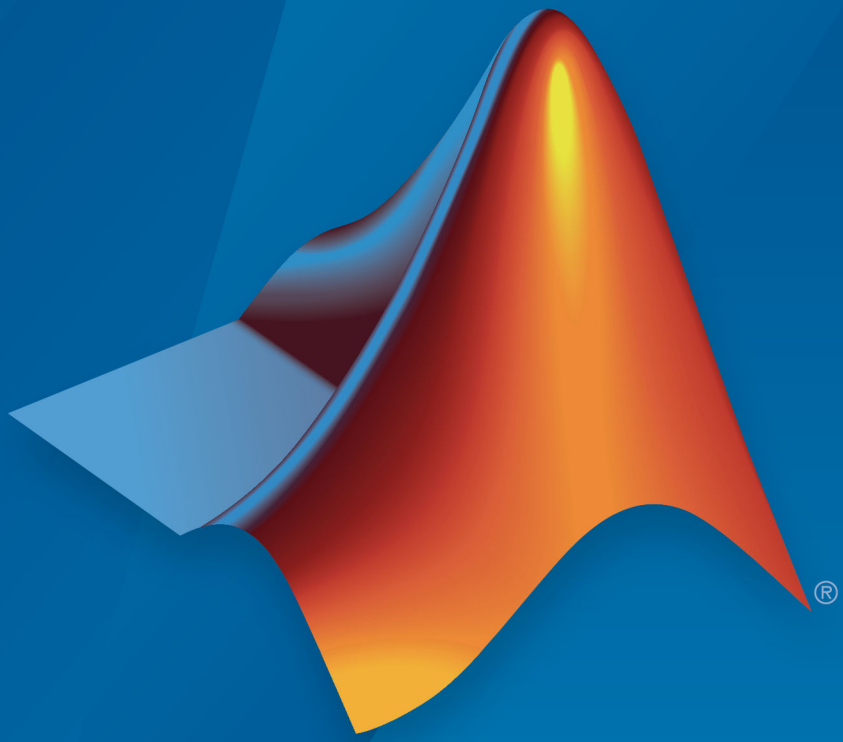
Embedded Coder® Release Notes



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Embedded Coder® Release Notes

© COPYRIGHT 2011–2018 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Contents

R2018b

Code Generation from MATLAB Code **1-2**

Column Limit in Generated Code: Generate more readable code by controlling line wrapping **1-2**
Static Code Metrics On Demand: Run static code metrics analysis when needed after code generation **1-2**
Single Instruction, Multiple Data (SIMD) Support: Generate Intel SSE/AVX intrinsic in MATLAB Coder **1-3**

Model Architecture and Design **1-5**

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 **1-5**
Code Preview in Embedded Coder Dictionary: Verify pseudocode preview as you select data, function, and memory section properties **1-5**
Embedded Coder Dictionary Mapping Control: Define storage classes that restrict mappings to parameters or signals . . . **1-6**
Embedded Coder Dictionary Version Handling: Use and export code definitions saved in previous releases with models created in later releases **1-6**
AUTOSAR Run-Time Calibration: Map internal signals, states, and model workspace parameters to AUTOSAR component memory and internal parameters for calibration **1-6**
AUTOSAR Memory Sections: Use SwAddrMethods to control memory placement of AUTOSAR runnable functions and internal data **1-8**
AUTOSAR XML Import and Export: Round trip imported arxml file structure and control packaging of new elements **1-8**
AUTOSAR XML Import: Changes to ArTypedPerInstanceMemory and StaticMemory import behavior **1-9**

Obsolete AUTOSAR signal and state map functions removed **1-10**
MISRA C:2012 and Secure Coding Standards: Improve compliance of generated code by using updated Model Advisor checks **1-11**

Data, Function, and File Definition **1-12**

Individual Function Mappings in Code Mapping Editor: Override default function mappings with individual function mappings **1-12**
Function Interface Control: Access Configure C Step Function Interface dialog box from Code Mapping Editor in code perspective **1-13**
Function Interface Control: Configure step functions for multi-instance, rate-grouped, single-tasking models **1-13**
Shared Default Code Configurations for Data and Functions: Share default code configuration settings between models **1-13**
Storage Class on Root-Level I/O: Access global data and functions in multi-instance models **1-14**

Code Generation **1-15**

Code View in Code Perspective: View generated code directly in Code Perspective **1-15**
Data Coherency: Generate one variable for each Data Store read and write operation **1-15**
AUTOSAR Code Generation: Automatically generate AUTOSAR platform data types in C code **1-16**
Data Type Replacement: Specify replacement types for 64-bit integers **1-16**
Multi-Dimensional Arrays: Preserve array dimensions for parameters and lookup tables in generated code **1-17**
Hardware Implementation Parameters: ProdHWDeviceType and TargetHWDeviceType are case-insensitive **1-19**
Enumerated Types: Optimizations in generated code **1-19**

Deployment **1-21**

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 **1-21**

Code Generation Assumptions: Use standalone workflow to run checks 1-21
Build Process: Library and header files for model reference hierarchy are not copied 1-21
Build Process: MATLAB_INCLUDES is not required in custom template makefiles 1-22
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) . . . 1-22

Performance 1-23

Execution-Time Profiling: Specify profiling granularity through model-wide and block-specific controls 1-23
Global Variable Caching: Reduce access for global variable arrays with custom storage classes 1-24
Data Copy Reduction: Eliminate unnecessary data copies for Mux blocks 1-25
Enhanced Buffer Reuse: Buffer reuse across the boundary of an Iterator subsystem 1-29
Code Replacement: Optimize generated code with SIMD and row-major order support and code replacement enhancements 1-31
Inplace Optimization for Assignment Blocks: Reduce data copies for Assignment blocks 1-32
Execution Speed: Eliminate redundant subexpressions 1-34
Single Instruction, Multiple Data (SIMD) Intrinsics: Generate code with optimized load and store operations for multidimensional signals and square root operations 1-37
Code Generation Report: Generate static code metrics reports faster 1-38
Functionality Being Removed or Changed 1-39
Cache Efficiency: Store global block signal and state data operating at the same rate in one data structure 1-39

Verification 1-42

SIL and PIL Simulations: Advanced custom storage classes support 1-42
SIL and PIL Simulations: Support for imported grouped custom storage classes 1-42
Model Block SIL and PIL: Accelerator mode SIM target is not built 1-42

Check bug reports for issues and fixes **1-44**

R2018a

Code Generation from MATLAB Code **2-2**

Interactive Traceability: Visualize mapping between MATLAB code and C code **2-2**
Polyspace Integration: Verify C/C++ code generated with MATLAB Coder by using simplified workflow **2-2**
Changes to Setup for MISRA C Compliance: Disable dynamic memory allocation and set C standard math library to C99 (ISO) **2-3**

Model Architecture and Design **2-4**

Embedded Coder Dictionary: Create custom code generation definitions for data and functions **2-4**
Multi-Instance Code Generation: Apply more control when generating reusable, reentrant code **2-5**
Variant Blocks Usability Enhancement: Generate Preprocessor Conditionals by using MATLAB variables as variant controls **2-6**
MISRA C:2012 Compliance and Deviation Considerations: Guidance for evaluating your generated code for compliance with MISRA C:2012 directives and rules **2-6**
Modeling Checks: Improve compliance of generated code by using Model Advisor check for MISRA C:2012 . . . . . **2-7**
AUTOSAR Release 4.3: Import and export AUTOSAR XML schema version 4.3 . . . . . **2-7**
AUTOSAR Perspective: Map and configure software components by using Code Mapping Editor and AUTOSAR Dictionary . . **2-8**
AUTOSAR XML Import and Export: Round-trip ComSpecs, import bitfield CompuMethods, export interface variation points, and automate more element creation **2-11**
AUTOSAR Signal Invalidation Block: Specify invalidation policy and initial value directly as block parameters **2-14**
AUTOSAR Basic Software: Use array and bus data types with NvMServiceCaller operations **2-15**
Obsolete AUTOSAR functions removed **2-15**

Data, Function, and File Definition **2-18**

Function-Prototype Control: Configure step function name with void void interface **2-18**
Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model **2-18**
GetSet Custom Storage Class Enhancement: Improved readability for an array of buses **2-20**
Local Storage Class: Preserve local variables with Localizable storage class **2-21**
Accurate Header File Extension: Generate correct #include statements for imported data types **2-21**
Macro Access: Get data through a macro that your code define s **2-22**
Tokens for Memory Sections: Use $N token instead of identifie r **2-22**
Parameter Initialization: Statically initialize tunable parameters from system constants and other macros **2-22**
Model-Scoped Parameter Objects: Use FileScope to prevent name clashes between parameters in different models . . . **2-23**
File Packaging of Generated Code for Global Simulink Function Blocks: Code for function body placed in model.c **2-23**
Identifiers: Represent name of storage class in identifier naming rules by using new token $G **2-24**
Functionality Being Removed or Changed **2-25**

Code Generation **2-27**

Code Perspective: Customize Simulink desktop for code generation workflows **2-27**
Rate Transition Block Code Customization: Separate Rate Transition block code and data from algorithm code and data **2-28**
Generated Files: Customize generated file names with new token $E **2-28**
Hardware Implementation Settings: Inaccurate values corrected **2-29**
Cross-Release Code Integration: Reuse referenced model code with instance-specific parameters **2-30**
Cross-Release Code Integration: Import and simulate AUTOSAR code **2-30**

Traceability Comments: Specify Simulink identifier in comments for Simulink blocks, Stateflow objects, and MATLAB Function blocks **2-30**
Newline Style: Customize linefeed character irrespective of the operating system **2-31**
Export Functions: Generate ScratchModel file containing a Model block **2-32**

Deployment **2-33**

Build Process: Specify toolchain for template makefile **2-33**
Build Process Status for Parallel Builds: View and interact with build process status for parallel builds of referenced model hierarchies **2-33**
TI C2000 IPC Block: Support for Inter-Processor Communications for F2837xD in TI C2000 Support Package **2-33**
C2000 F28004x: Support for peripherals in Texas Instruments C2000 Support Package **2-33**
STM32F7 Audio: Multiple channel Mic-In, Line-In, and Speaker out for STM32F769I-Discovery in STM32 Support Package **2-34**
STM32F7 External Mode: Support for TCP/IP and Serial Communication for STM32F769I-Discovery board in STM32 Support Package **2-34**
External Mode Simulation: Upload execution-time metrics through XCP transport layer **2-34**

Performance **2-36**

Single Instruction, Multiple Data (SIMD) Intrinsics: Generate code with optimized load and store operations for use with Intel processors with SSE/AVX support **2-36**
Preprocessor Conditionals: Obtain better readability of generated code for variant systems **2-38**
Buffer Reuse: Prioritize buffer reuse based on signal labels in model diagram **2-39**
Configuration Set: New location and layout for optimization model configuration parameters **2-40**
Data Copy Reduction: Generate code with fewer data copies for writes to structure fields and matrix elements and for control flow patterns **2-42**
Code Size Reduction: Eliminate identical functions in the generated code **2-47**

Code Replacement: Optimize generated code with SIMD and row-major order support and improved library header file packaging 2-49
Execution Speed: Move invariant code containing global variables out of for loops 2-50

Verification 2-53

PIL Simulation: Verify initial values of global variables 2-53

Check bug reports for issues and fixes 2-54

R2017b

Code Generation from MATLAB Code 3-2

Setup for MISRA C Compliance: Configure code generation parameters to increase compliance with MISRA C:2012 guidelines 3-2
SIL/PIL Execution Performance: Speed up SIL or PIL execution by disabling constant input checking and global data synchronization 3-2
Execution-Time Profiling: Display time units in code execution profiling report 3-2
Default Case for Switch Statements: Increase generated code compliance with coding standards 3-2

Model Architecture and Design 3-4

Function Interfaces: Generate multi-instance functions from export-function models and control scope of Simulink functions 3-4
AUTOSAR Compositions and Basic Software: Import AUTOSAR compositions and simulate diagnostic and memory services 3-5
AUTOSAR Sender-Receiver Communication: Model AUTOSAR queued send and receive using Simulink messages 3-6
MISRA C: 2012 Modeling Checks: Improve compliance of generated code by using new MISRA C: 2012 standards checks 3-6

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 **3-8**
Code Reuse: Generate reusable code for subsystems that contain data objects with imported custom storage classes **3-10**

Data, Function, and File Definition **3-12**

Storage Class for Model Workspace Parameters: Apply custom storage classes to parameter objects in a model workspace **3-12**
Custom Storage Class Simplification: Default removed from drop-down lists **3-12**

Code Generation **3-13**

Cross-Release Code Integration: Reuse code from models containing model references, global I/O, data stores, and parameters **3-13**
Cross-Release Code Integration: Run all workflow tasks from current release **3-13**
AUTOSAR Run-Time Calibration: Measure and calibrate signal and discrete state data using arTypedPerInstanceMemory **3-14**
Stateflow Element Traceability: Obtain enhanced inline traceability **3-14**
Stateflow Objects and MATLAB User Comments: Configure comments flexibly **3-15**
Enhanced Shared Utilities Naming: Customize the names of shared utility functions that are inside MATLAB Function blocks **3-16**
Checksum Length: Specify the character length of the $C token **3-16**
Code Style: Generate static keyword for locally scoped functions **3-16**
Configuration Parameters Dialog Box: View your model and code generation configuration parameters in unified dialog box with search capability **3-17**
Improved Readability of the Generated Code: Include parentheses around compound expressions containing right-shift operators **3-18**

Deployment **3-20**

AUTOSAR Support Package: Run live-script examples for AUTOSAR compositions and Basic Software **3-20**
Support Package renamed to Embedded Coder Support Package for Intel SoC Devices **3-20**
Support Package renamed to Embedded Coder Support Package for Xilinx Zynq Platform **3-20**
Removed Support for Wind River VxWorks Hardware **3-20**

Performance **3-22**

RAM Reduction: Reduce data copies in For Each subsystems and reuse buffers of different sizes **3-22**
Reusable Storage Class: Specify reusable custom storage classes anywhere on a path **3-26**
Execution Speed: Eliminate redundant subexpressions **3-27**
Execution Speed: Convert data copies to pointer assignments for more modeling patterns **3-28**
Execution Speed: Move invariant code out of for loops **3-32**
Block Reordering for Improved Execution Efficiency: Change block execution order to enable buffer reuse and loop fusion **3-37**
MATLAB Function Block Buffer Reuse: Perform inplace assignment with root I/O **3-37**
Execution-Time Profiling: Display time units in code execution profiling report and Simulation Data Inspector **3-38**
memcpy and memset Optimization: Generate more efficient code for variable-size arrays **3-39**
Data Copy Reduction: Generate fewer data copies at function call sites **3-41**
Code Replacement: Apply MustHaveZeroNetBias and SlopesMustBeTheSame properties for fixed-point operator code replacement **3-43**
Enumerated Data Types Optimization: Improve the efficiency of the generated code for enumerated data types **3-43**

Verification **3-45**

Multiple Processor SIL/PIL Testing: Perform SIL or PIL component tests on different processors simultaneously . . **3-45**
SIL Simulation: Simplified configuration of hardware implementation settings **3-45**

SIL/PIL Configuration: Parent model code coverage, execution-time profiling, and SIL debugging settings apply to Model blocks with Top-model code interface **3-45**
Hardware Implementation Settings: SIL checks relaxed for data type sizes and byte ordering **3-46**

Check bug reports for issues and fixes **3-48**

R2017a

Code Generation from MATLAB Code **4-2**

SIL and PIL execution improvements for MATLAB Coder **4-2**
Verification of PIL target connectivity configuration **4-2**
Code Replacement for MATLAB Coder: Create code replacement library entries for target implementations that require data alignment **4-3**

Model Architecture and Design **4-4**

AUTOSAR arxml File Import: Flexibly model imported periodic, asynchronous, and initialization runnables **4-4**
AUTOSAR DESC elements populate Simulink Description field s **4-5**
External mode code generation for a model containing inline variant blocks **4-6**
Code generation support for Variant Subsystems containing global signals **4-6**
Preprocessor conditionals guard content inside and outside of function-call site **4-6**

Data, Function, and File Definition **4-9**

Function Interface: Return nonvoid type for scalar output of reusable functions **4-9**
Utility to generate Simulink representations of struct and enum types defined by external C code **4-13**

Code Generation **4-14**

Cross-Release Code Integration: Reuse model reference code generated from previous releases **4-14**
Code Replacement for Cast and Multiply Operations: Detect overflow and rounding mode equivalence for increased matches and code efficiency **4-14**
More information in code generation report summary **4-15**
Code Interface Report: Includes entry-point function for code generated from Reset Function block **4-15**
Shared utility memory section associated with subfunctions **4-16**
Inline traceability for generated code **4-16**
Clear file section content from TLC file **4-16**
Identifier case control with token decorators and custom text token $U **4-17**
Name change for AUTOSAR local temporary variables **4-18**
Additional checks against MISRA C:2012 guidelines in Code Generation Advisor **4-18**

Deployment **4-19**

TI Code Composer Studio (CCS): Generate projects for CCS versions 5 and 6 with Embedded Coder Target for TI C2000 . . . . . **4-19**
Customize generated makefiles for S-Functions **4-19**
Release notes and workflow overview documentation added to AUTOSAR support package **4-19**
SPI and I2C blocks added to TI C2000 support package **4-19**
CCS v3.3 IDE automation support for TI C2000 has been removed **4-20**
Real-time multitasking profiling for TI C2000 . . . . . **4-20**
TCP and UDP blocks added to STMicroelectronics STM32F746G-Discovery board **4-20**
MATLAB Coder PIL with STMicroelectronics STM32F4-Discovery Board **4-21**
External Mode and PIL supported over TCP/IP by STMicroelectronics STM32F746G-Discovery board **4-23**
Linux Support: Connect to ARM Cortex-M processor on Linux platform **4-23**
ARM Cortex-R optimized code **4-23**
Develop a Target for ARM Cortex-R processors **4-24**
Support for Wind River VxWorks RTOS will be removed **4-24**

Performance **4-25**

Data Copy Reduction: Generate fewer data copies and use less RAM for buses, data stores, and model blocks **4-25**
Code Efficiency: Improve loop fusion for Sum of Elements blocks and generate less code for temporal logic in Stateflow . . . **4-34**
Data copy reduction for Merge blocks **4-37**
More instances of buffer reuse for blocks and subsystems in a chain **4-40**
Improved buffer reuse due to changes in block execution order **4-44**
More efficient code for Bus Creator blocks **4-46**
Buffer reuse for Variant Source blocks **4-47**

Verification **4-50**

SIL and PIL Testing: Log signals inside exported functions and stream signals to Simulation Data Inspector during simulation **4-50**
Verification of PIL target connectivity configuration **4-50**

Check bug reports for issues and fixes **4-52**

R2016b

Code Generation from MATLAB Code **5-2**

Static code metrics report for C++ code **5-2**
Verification of size_t and ptrdiff_t hardware settings **5-2**
Verification of PIL target connectivity configuration **5-2**
Optimization for array indexing in loops **5-2**
Reduction of the Intel Performance Primitives (IPP) code replacement libraries (CRL) **5-3**

Model Architecture and Design **5-4**

AUTOSAR Basic Software (BSW) Services: Simulate BSW including Diagnostic Event Manager (DEM) and NVRAM Manager (NvM) **5-4**
AUTOSAR Parameters: Model STD_AXIS and COM_AXIS lookup table parameters, export SwRecordLayouts, and apply SwAddrMethods **5-4**

AUTOSAR startup, reset, and shutdown modeling **5-7**
AUTOSAR external trigger event communication **5-8**
AUTOSAR support for JMAAB model architecture **5-8**
AUTOSAR ExplicitReceiveByVal data access mode for receiver ports **5-9**
AUTOSAR ModeSenderPorts and ModeSwitchPoints for application mode management **5-10**
AUTOSAR reference element definitions for sharing among components and services **5-11**
ERT Target Code Generation: Remove unreachable reset and disable functions to reduce dead code **5-12**
Conditional compile time check for imported macros with ImportedDefine custom storage class **5-13**
Additional guarding of global data for variant systems **5-13**

Data, Function, and File Definition **5-18**

Simulink Function Code Interface: Configure generated C/C++ function interfaces for Simulink Function and Function Caller blocks **5-18**
ERT default value for configuration parameter ParameterTunabilityLossMsg **5-19**

Code Generation **5-20**

Cross-Release Code Integration: Reuse code generated from earlier releases **5-20**
Compound Operation Code Replacement: Replace "Multiply Shift Right Arithmetic" and "Multiply Divide" in generated code with a single custom operation **5-21**
ARXML import/export and C code generation for latest AUTOSAR 4.2 and 3.2 standard revisions **5-21**
AUTOSAR code replacement library enhancements **5-22**
Static code metrics report for C++ code **5-22**
Static code metrics data produced by Polyspace **5-22**
Streamlined report pane for easier model configuration **5-23**
Improved traceability between model and code **5-23**
Code replacement enhancements **5-24**
$I macro changed for argument names used as input and output **5-25**
Improved compliance with MISRA C:2012 Rules 10.1, 10.5, and 10.8 . . . . . **5-25**
Improved compliance with MISRA AC AGC Rule 12.6 . . . . . **5-27**

Use default installation folder on Windows system with ReFS file system 5-29

Deployment 5-30

Cortex-M7 Target Support Package: Generate code for STM32F746G-Discovery Board 5-30
Added Embedded Coder Support Package for ARM Cortex-R Processors 5-30
Improved External mode over serial communication 5-31
New blocks added to TI’s C2000 support package 5-31
Change in name and the base product for the FRDM-K64F and the FRDM-KL25Z support packages 5-31
Support for TI's C5000 DSPs has been removed 5-32
Support for TI’s C6000 has been removed 5-32
Support for Wind River VxWorks RTOS will be removed 5-32
Support for idelink_ert.tlc will be removed 5-32

Performance 5-33

Data Reuse and Memory Reduction: Reuse global data for nonreusable subsystems and reduce data copies with user-specified buffers 5-33
Code Optimizations: Generate more efficient code with select-assign-iterator pattern and matrix padding operations . . . 5-36
Display of code execution times for model component 5-40
More efficient code for array element assignments 5-41
Loop fusion for nested for loops 5-43
More efficient initialization code for root-level inports 5-45
More efficient code for Boolean expressions 5-49

Verification 5-51

Verification of size_t and ptrdiff_t hardware settings 5-51
Verification of PIL target connectivity configuration 5-51
Signal range checking in SIL and PIL simulations 5-51
SIL and PIL block support for Simulink Function and Function Caller blocks 5-51

Check bug reports for issues and fixes 5-52

Code Generation from MATLAB Code **6-2**

Export data by using ExportedDefine storage class **6-2**
SIL execution returns standard output and standard error streams **6-2**

Model Architecture and Design **6-4**

Compile-Time Dimensions: Generate compiler directives (#define) for implementing signal dimensions **6-4**
Compile-Time Variants: Generate compiler directives (#if) for variant choices specified with Variant Source and Variant Sink blocks **6-5**
C++ Code Generation: Use referenced models with multitasking, export-functions, and virtual buses **6-5**
MISRA C:2012 Compliance: Check block names and Assignment blocks by using the Model Advisor **6-6**
AUTOSAR Round Trip: Automate model additions for update and merge of ARXML files **6-6**
Comment change in generated code **6-7**
Variants in AUTOSAR component modeling **6-7**
AUTOSAR DataReceivedEvents for receiver ports in ImplicitReceive data access mode **6-9**
AUTOSAR LiteralPrefix for enumerations in IncludedDataTypeSets **6-10**
Programmatic validation and synchronization of AUTOSAR model configurations **6-10**

Data, Function, and File Definition **6-11**

In/Out Arguments: Specify same variable name for in/out arguments of MATLAB Function and Model blocks **6-11**
Custom Storage Class Type AccessFunction **6-15**
Creation of custom storage classes for macros defined by compiler options **6-15**
Generation of ERT S-functions that represent variant controls as preprocessor conditionals **6-16**

Code Generation **6-17**

Default style C++ interface replaces the void-void style C++ interface **6-17**
Compiler warning limitation removed for portable word sizes in SIL simulations **6-17**
AUTOSAR arxml round trip **6-18**
Improved AUTOSAR library support for Mfx functions **6-20**
AUTOSAR target no longer supports building wrapper subsystem as AUTOSAR SW-Component **6-21**
Root model name in generated identifier for shared utility files **6-21**
Improved configuration parameter defaults for Embedded Coder targets **6-21**
Streamlined code generation panes for easier model configurati on **6-22**
Build button removed from Configuration Parameters dialog box **6-27**
Improved web view for code generation report **6-28**
Dependent parameters not added to custom code generation objective **6-28**
Removal of leading underscore character in macro type definitio ns **6-28**

Deployment **6-30**

Hardware implementation parameters enabled by default . . **6-30**
MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with Altera SoC and Xilinx Zynq hardware **6-30**
Updates to support package for Texas Instruments C2000 processors **6-30**
Support package for Freescale FRDM-K64F board **6-31**
Support for TI’s C5000 DSPs will be removed **6-31**
Support for TI’s C6000 DSPs will be removed **6-31**
Change in base product for ARM Cortex-Based VEX Microcontroller support package **6-31**

Performance **6-32**

Data Buffer Reuse: Use same variable for multiple signals in a path by using the same Reusable storage class specificatio n **6-32**
Reuse input, output, and state of Delay block **6-32**
Initialization code occurs once after start code in model_initialize function **6-32**

Reset function improves initialization code optimization **6-36**
Removal of unnecessary rtmIsFirstInitCond flag **6-38**
Optimized code for models containing logical operator blocks **6-40**
Improved code for conditional expressions involving Boolean expressions **6-41**
memset Optimization for more scenarios **6-43**
Changes to meaning of createCRLEntry wildcard syntax for fixed-point data **6-50**
Code replacements involving root-level I/O variables and data alignment **6-51**

Verification **6-53**

SIL/PIL Data Access: Use vector Get/Set custom storage class and C++ parameter access methods **6-53**
SIL/PIL support for variant condition propagation **6-53**
SIL simulation returns standard output and standard error streams **6-53**
Linux SIL/PIL support for LDRA Testbed **6-54**

Check bug reports for issues and fixes **6-55**

R2015aSP1

Bug Fixes

Check bug reports for issues and fixes **7-2**

R2015b

Code Generation from MATLAB Code **8-2**

MATLAB Coder Storage Classes: Easily import and export data by using storage classes **8-2**

MATLAB Coder PIL With ARM Cortex-A: Verify and profile ARM optimized code with BeagleBone Black hardware **8-3**
Code generation assumptions verified during PIL execution . . **8-3**
Control of signed right shifts in generated code **8-4**
Detection of multiword operations **8-5**

Model Architecture and Design **8-6**

MISRA-C 2012: Comply with mandatory and required rules . . **8-6**
AUTOSAR 4.1.3 and 4.2: Import and export ARXML and generate code for latest AUTOSAR standard **8-7**
AUTOSAR sender-receiver modeling **8-7**
AUTOSAR client-server modeling **8-10**
AUTOSAR nonvolatile data communication modeling **8-12**
AUTOSAR component behavior modeling **8-14**
AUTOSAR COM_AXIS lookup table modeling **8-16**
Embedded Coder model templates **8-17**
Removal of uncalled Disable functions from generated code **8-17**
Enhancement to option for generating preprocessor conditionals **8-17**

Data, Function, and File Definition **8-19**

Tokenized function names for custom storage class GetSet . . **8-19**

Code Generation **8-20**

Embedded Coder Quick Start: Quickly configure model to generate reusable and efficient code **8-20**
Internationalization: Generate and review code containing mixed languages for different locales **8-20**
MISRA C:2012 code generation objective **8-21**
AUTOSAR arxml round-trip **8-21**
Toolchain controls for AUTOSAR code generation **8-24**
AUTOSAR RTE file generation enhanced for SIL and PIL . . . **8-24**
Lookup table blocks with new even spacing specification generate AUTOSAR compatible IFX library routines **8-26**
Control use of signed shifts in generated code **8-26**
Code generation report with operator traceability **8-27**

Deployment **8-28**

Hardware Implementation Selection: Quickly generate code for popular embedded processors **8-28**
Code Replacement Tool uses simplified specification **8-30**
Code replacement support for new lookup table breakpoint specification **8-31**
Support for Analog Devices VisualDSP++ will be removed . . **8-31**

Performance **8-32**

RAM/ROM Optimization Improvements: Generate more efficient code using reusable storage class and converting data copies to pointer assignments **8-32**
Live Execution Profiling: View PIL profile results during run-time **8-34**
Enhanced support for buffer reuse at the root-level input and output ports **8-34**
More efficient code for small subsystems **8-37**
More efficient code for Simulink.Bus objects **8-39**
Enhanced local variable reuse **8-41**
Enhanced consolidation of for loops **8-43**

Verification **8-46**

Faster SIL and PIL Verification Workflow **8-46**
Code generation assumptions verified during PIL simulation **8-46**
SIL and PIL support for C++ class root-level I/O access methods **8-46**
Removal of Generate code only parameter restriction **8-47**
Removal of scheduling limitations that caused algebraic loops **8-47**

Check bug reports for issues and fixes **8-2**

R2015a

Code Generation from MATLAB Code **9-2**

Indent style and size control for generated C/C++ code **9-2**

Improved MISRA-C compliance for bitwise operations on signed integers **9-3**
Improved MISRA-C type cast compliance **9-4**

Model Architecture and Design **9-6**

AUTOSAR improvements including multi-runnable modeling and code efficiency **9-6**
Combined input/output arguments with function prototype control **9-6**
Improved MISRA-C compliance for bitwise operations on signed integers **9-7**
AUTOSAR multi-runnable modeling using Simulink rate-based multitasking **9-7**
Enhanced modeling with AUTOSAR system constants **9-8**
AUTOSAR CompuMethod enhancements **9-8**
Preprocessor conditionals for single variant choice **9-9**

Data, Function, and File Definition **9-10**

Control of Boolean and data type limit identifiers in generated code **9-10**
Names of built-in storage classes reserved **9-10**

Code Generation **9-12**

Simplified Code Replacement Library specification plus more replacements involving integer operations **9-12**
Improved readability for shared header file 'rtwtypes.h' **9-13**
New and enhanced Model Advisor checks for MISRA-C compliance **9-15**
Improved traceability for AUTOSAR RTE implicit read **9-15**
Configurable aliveTimeout value for AUTOSAR ports **9-16**
AUTOSAR calibration parameter export for COM_AXIS lookup tables **9-17**
Fixed-point scaling information in Code Interface Report . . . **9-17**
Unsigned integer minimum data limit identifiers **9-17**
Default iteration variable data type **9-18**

Deployment **9-20**

Code Replacement Viewer enhanced **9-20**
Model configuration parameter considered for division operator code replacements **9-20**

Lookup table algorithm parameter specification enhancements 9-20
Header file for Basic Linear Algebra Subroutine (BLAS) multiplication function code replacement example changed 9-21
Code replacement detection of overflow and rounding mode equivalence 9-21
Feature being removed in a future release 9-21

Performance 9-22

More efficient code involving model references, unit delays, and global data references 9-22
Conditional compilation of Data Store Memory block memory definition and declaration 9-28
Ternary Boolean expressions transformed into assignment statements 9-30

Verification 9-31

SIL/PIL for protected models and SIL source code debugging using Microsoft Visual Studio Express 9-31
Model block SIL/PIL parameter renamed 9-32
ERT S-Function block no longer supported for AUTOSAR . . . 9-32
SIL/PIL support for replacing boolean data type with int8 . . . 9-32
SIL/PIL support for generated access methods for C++ model class root-level I/O signals 9-33

Check bug reports for issues and fixes 9-34

R2014b

Code Generation from MATLAB Code 10-2

Processor-in-the-loop (PIL) verification and execution profiling for MATLAB code 10-2
Software-in-the-loop verification improvements for MATLAB Coder 10-2
Additional options for custom banners and comments in C and C ++ code generated from MATLAB code 10-3

Highlighting of potential data type issues in code generation reports **10-4**

Model Architecture and Design **10-9**

AUTOSAR targeting updates including 4.1 ARXML, client/server with Simulink Functions, multi-instance components, and IFL/IFX libraries **10-9**
AUTOSAR client and server modeling **10-10**
Global From and Goto blocks for AUTOSAR modeling **10-10**
AUTOSAR IRV branch from outport signal allowed outside runnable **10-10**

Data, Function, and File Definition **10-12**

Constant sample time limitation for AUTOSAR models **10-12**
Iteration variable in For Iterator block uses signal name . . . **10-12**
Data type replacement specification can be used across models **10-12**
Definition file for grouped custom storage classes **10-12**
Type definition location for custom storage classes **10-13**
GetFunction and SetFunction included in checks for identifier clash **10-13**

Code Generation **10-14**

Enhanced reporting of eliminated blocks **10-14**
Improved MISRA-C type cast compliance **10-14**
Support Package for AUTOSAR Standard **10-14**
AUTOSAR help navigation enhancements **10-15**
Support for AUTOSAR Release 4.1 . . . . . **10-16**
Multi-instance AUTOSAR atomic software components **10-17**
AUTOSAR arxml import and export **10-17**
AUTOSAR addPackageableElement replaces add*Interface functions **10-22**
Code generation report with enhanced navigation and integrated access to code metrics data **10-23**
Updated license requirements for viewing code generation report **10-23**
Option for doxygen style comments in generated code **10-24**
Dynamic memory allocation parameters renamed **10-25**
Template makefile compatibility with execution time profiling **10-25**

Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation **10-26**

Deployment **10-27**

Embedded Coder support packages for AUTOSAR, TI Concerto, and Freescale FRDM-KL25Z **10-27**
Relational operator replacement **10-27**
Code replacement involving vector and matrix data **10-27**
Algorithm specification for addition and subtraction operator replacement **10-29**
Improved code replacement with output type cast absorption **10-29**
Lookup table function code replacement extended to 30 dimensions **10-30**
Rounding mode support for lookup table function replacement **10-30**
Algorithm parameter value sets in code replacement table entries **10-30**
coder.replace support for functions specified with varargin input variable **10-31**
Documentation installation with hardware support package **10-31**
Support package for Altera SoC platform **10-31**
Support package for BeagleBone Black hardware **10-31**
Support for Eclipse IDE has been removed **10-32**
Support for Green Hills MULTI IDE has been removed **10-32**
Support for Texas Instruments C5000 DSPs will be removed **10-32**

Performance **10-33**

Reduced RAM and faster execution for modeling patterns including select-assign-iterate blocks, subsystem interfaces, and model references **10-33**
Global variable localization optimizations **10-39**

Verification **10-41**

Top-model code testing with Model block SIL and PIL **10-41**
SIL/PIL support for Simulink Function and Function Caller blocks **10-42**
SIL debugging support for Linux **10-42**

PIL support for test hardware approach **10-42**
SIL/PIL support for model initialization dynamic memory allocation **10-42**

Check bug reports for issues and fixes **10-43**

R2014a

Code Generation from MATLAB Code **11-2**

Template to customize code generation output for MATLAB Coder **11-2**
In-place function replacement with coder.replace in MATLAB **11-2**
Single-line (///) comment style available for generated code **11-2**
Software-in-the-loop verification for MATLAB Coder **11-3**
Change of default value for MATLABFcnDesc **11-5**

Model Architecture and Design **11-6**

Capability to merge AUTOSAR authoring tool changes into Simulink models as part of round-trip iterations **11-6**
AUTOSAR 4.0 static and constant memory, AUTOSAR-typed per-instance memory, and VariationPointProxy **11-8**
Specify AUTOSAR runnable symbol name distinct from short-name **11-9**
Improved AUTOSAR arxml support for measurement and calibration **11-10**
AUTOSAR data dictionary support **11-10**
Configure AUTOSAR Interface button removed from AUTOSAR Code Generation Options **11-11**
Subsystem methods of AUTOSAR arxml.importer class removed **11-11**

Data, Function, and File Definition **11-13**

Custom storage class and optimized class declarations for C++ class code generation **11-13**

Constant sample time limitations for root-level Outport blocks 11-13
Example model rtwdemo_cppencap renamed to rtwdemo_cppclass 11-14
Unit Delay block optimization 11-14

Code Generation 11-15

In-place function replacement with coder.replace in MATLAB and lookup table code replacement for Simulink 11-15
Global variable usage available in the static code metrics report 11-15
Single-line (///) comment style available for generated code 11-16
Code indentation support for namespace declarations in generated code 11-16
AUTOSAR C code generation enhancements 11-17
Static main program module for C++ class code generation 11-18
Error message for data type replacement and classic call interface conflict 11-18

Deployment 11-20

ARM Cortex-A optimized code generation using Ne10 library 11-20
Lookup table code replacement for Simulink 11-20
Replacement of functions that take vector and matrix arguments 11-21
Logical data type support for arguments of replaced functions 11-21
Code replacement data alignment for complex types 11-21
Intel IPP (ANSI) and Intel IPP (ISO) code replacement libraries are combined 11-22
Support for Eclipse IDE will be removed 11-22
Support for Green Hills MULTI IDE will be removed 11-23
Support package for ARM Cortex-A processors 11-23
Support package for Texas Instruments C6000 processors . 11-23
Updates to support package for Texas Instruments C2000 processors 11-24
Updates to support package for Xilinx Zynq-7000 platform 11-25
Updates to support package for STMicroelectronics STM32F4 Discovery board 11-25

Wind River Tornado (VxWorks 5.x) example main program option to be removed in future release **11-26**

Performance **11-27**

Additional options for reuse of global variables **11-27**
Enhanced global variable optimization options **11-27**
for loops used to initialize arrays to zero **11-27**

Verification **11-29**

Software-in-the-loop simulation for physical models **11-29**
SIL verification for subsystem code generation **11-29**
SIL and PIL support for fixed-point data type override **11-32**
SIL and PIL support for Invoke AUTOSAR Server Operation block **11-33**
SIL and PIL support for structure parameters with storage class SimulinkGlobal **11-33**
Model block SIL and PIL with export-function and asynchronous function-call models **11-33**
Model block SIL and PIL with disabled inline parameters . . **11-34**
SIL and PIL block improvements **11-34**

Check bug reports for issues and fixes **11-35**

R2013b

Code Generation from MATLAB Code **12-2**

Software-in-the-loop verification for MATLAB Coder **12-2**
Custom generated identifiers for emxArray utility functions **12-2**

Model Architecture and Design **12-4**

Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior **12-4**
Reorganization of Model Advisor Embedded Coder checks **12-6**

Model Advisor fixed-point checks with additional coverage and optimization awareness **12-7**
Protected model Web view **12-7**
RTW.AutosarInterface class to be removed in a future release **12-7**
Subsystem methods of arxml.importer class to be removed in a future release **12-8**

Data, Function, and File Definition **12-9**

Simplified global types file rtwtypes.h with invariant content **12-9**
C++ encapsulation support for name space control and template-based file customization **12-9**
Shared utility naming control **12-10**
Expanded support for identifier names **12-11**
Terminate function setting honored for subsystems and referenced models **12-11**

Code Generation **12-12**

Support for AUTOSAR release 4.0.3 XML and generated code **12-12**
Indent style and size control for code generation **12-12**
Subsystem functions return value in generated code **12-12**
Model reference step function void input and output arguments **12-13**

Deployment **12-14**

ARM Cortex-M optimized code with STM32F4-Discovery board example **12-14**
Wind River VxWorks 6.9 support **12-15**
Support package for Texas Instruments C2000 processors . **12-16**
Coder Target pane in Configuration Parameters dialog box **12-17**
ZedBoard hardware support **12-18**
Simplified multi-instance code interface and dynamic memory allocation for ERT targets **12-19**
Addition and Subtraction Operator Code Replacement Assumes Cast-Before-Operation Behavior **12-20**

Performance **12-22**

Reusable custom storage class to reduce root I/O memory . **12-22**
Subsystem functions reused independently of output connection **12-22**

Verification **12-23**

SIL and PIL support fixed-point data types wider than 32 bits **12-23**
SIL and PIL protected model support **12-24**
Code execution profiling improvements **12-24**

Check bug reports for issues and fixes **12-26**

R2013a

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

Improved code replacement traceability for MATLAB code generation **13-2**
Static code metrics report for MATLAB Coder **13-2**

Model Architecture and Design **13-4**

AUTOSAR user interface and round trip ARXML file import and export improvements **13-4**
Code generation for variable-size scalar signals **13-7**

Data, Function, and File Definition **13-8**

Shortened system-generated identifier names **13-8**
Improved data initialization with custom storage classes . . . **13-8**
Default specification for global types **13-8**
Subsystem block parameter Function packaging option renamed **13-9**

Code Generation **13-10**

Model Advisor checks for code generation **13-10**

Deployment **13-11**

Concurrent execution API to target embedded multicore platforms 13-11
Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box 13-12
Downloadable support and blocks for Analog Devices DSPs 13-13
Texas Instruments C2000 Clocking Options 13-14
Support for Texas Instruments C2802x and Texas Instruments C2803x variants 13-15
Downloadable support and blocks for Xilinx Zynq-7000 platform 13-15
Support ending for Eclipse IDE in a future release 13-16
Support ending for remoteBuild method in a future release 13-16

Performance 13-17

Optimized function arguments for nonreusable subsystems 13-17
Reduced data copies for tunable parameter expressions . . . 13-17
Removal of unused global variables 13-18

Verification 13-19

Debugging during SIL simulations 13-19
Simulation of multiple SIL Model blocks in a top model . . . 13-19
API for testing rtiostream communications 13-19
SIL and PIL support for targets with multicore processors . 13-20
Additional code annotation for justifying Polyspace checks 13-21
Code execution profiling improvements 13-21
Code-to-model traceability links for reusable subsystems in libraries 13-22

Check bug reports for issues and fixes 13-24

R2012b

Cyclomatic complexity measurement in static code metrics report 14-2

Custom code substitution for MATLAB functions using code replacement libraries **14-2**

SIL and PIL support for signal logging, encapsulated C++, and AUTOSAR calibration parameters **14-2**
Signal logging for SIL and PIL simulations **14-3**
Use SIL and PIL simulations to verify encapsulated C++ code **14-3**
Improved SIL and PIL verification for AUTOSAR-compliant code **14-3**

AUTOSAR 4.0 nonscalar data support **14-4**

Code annotation for justifying Polyspace checks **14-4**

Texas Instruments Code Composer Studio IDE 5.1 support **14-5**

External mode support for ERT targets with static main **14-5**

Downloadable support for Green Hills MULTI **14-5**

Support for Texas Instruments C2806x processors **14-6**

Performance enhancement of Simulink data objects **14-7**

AUTOSAR software component import and export enhancements **14-8**
Import validation **14-8**
Faster import and export of arxml files **14-9**
Explicit access mode for AUTOSAR Sender and Receiver ports **14-9**
Import port-based calibration parameters **14-9**

Highlight virtual blocks in model Web view of code generation report **14-9**

Code Execution Profiling Improvements **14-9**
Updated Code Execution Profiling API **14-9**
Code Execution Profiling Supports Single Object Output . . **14-13**

Incremental Compilation with Changes in Code Coverage Settings **14-13**

Check bug reports for issues and fixes **14-14**

R2012a

AUTOSAR Enhancements **15-2**
AUTOSAR Release 4.0 . . . . . **15-2**
Support for Schema 2.0 Removed **15-2**

Code Efficiency Enhancements **15-2**
For Each Subsystem Loop Bound Passed by Value **15-2**
Fully Inlined S-functions from Legacy Code Tool **15-3**
Element-Wise Operations as Inputs to Intrinsic Functions . . . **15-3**

Enhancements to Custom Storage Classes in Simulink and mpt Packages **15-4**

Code Generation Report Includes Simulink Web View **15-5**

LDRA Testbed Code Coverage Annotations in Code Generation Report **15-5**

Generated Identifiers Enhancements **15-6**
Simplified Identifiers for Model Reference Code **15-6**
Consistent Identifiers for Comparing Generated Code **15-6**

Code Replacement Enhancements **15-6**
Target Function Libraries Renamed to Code Replacement Libraries **15-7**
Enhanced Code Replacement Traceability **15-7**
Code Replacement Support for Simulink Matrix Division and Inversion Operators **15-8**
Code Replacement Support for MATLAB Coder fix, hypot, round, and sign Functions **15-8**
Integer Functions Now Return Real-World Values **15-9**

SIL and PIL Enhancements **15-9**
SIL and PIL Test Harness Files in Code Generation Report **15-9**
PIL Support for Code Coverage with LDRA Testbed **15-11**

Seamless Switching Between SIL and PIL for Top-Model and Model Block 15-11
Enhanced Hardware Implementation Support 15-11
Top-Model Output Limitations Removed 15-12
Model Block SIL/PIL Support for Absolute Time 15-12

Changes for ERT and ERT-Based Targets 15-13

Changes for Embedded IDEs and Embedded Targets 15-14
Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE 15-14
Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors 15-14
Support Removed for Freescale MPC5xx 15-14
Limitation: Parallel Builds Not Supported for Embedded Targets 15-15

New and Enhanced Demos 15-15

Check bug reports for issues and fixes 15-17

R2011b

Static Code Metrics in Code Generation Report 16-2

AUTOSAR Enhancements 16-2
Import and Export of AUTOSAR Sensor/Actuator Components 16-2
Improved Simulink Library Support for Multiple Runnables 16-2
AUTOSAR Schema Version 3.2 16-3
Export AUTOSAR XML as Single File 16-3

SIL and PIL Enhancements 16-3
Code Execution Profiling of Functions in Subsystems and Model Blocks 16-3
Code Coverage with LDRA Testbed 16-3
BitField and GetSet Custom Storage Classes 16-3

Model Blocks with Variable-Size Signals 16-4
Verification of Generated C++ Code 16-4

Generate Multitasking Code for Concurrent Execution on Multicore Processors 16-4

Changes for Embedded IDEs and Embedded Targets 16-5
64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0 16-5
Support Added for Wind River VxWorks 6.8 16-6
Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335 16-6
New Target Function Library for Intel IPP/SSE (GNU) 16-6
Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors . . . 16-6
Support Removed for Altium TASKING 16-7
Support Removed for Infineon C166 16-7
Support Ending for Green Hills MULTI in a Future Release 16-7
Support Ending for Freescale MPC5xx in a Future Release 16-7

Saturation Control of Stateflow Data 16-7

Custom Storage Class Properties for Managing Data Ownership and Definition 16-8

Export Data Declarations to Shared Header File for Code Generation with Model Reference 16-9

Target Function Library Code Replacement Enhancements 16-9
Code Replacement Tool for Creating and Managing TFL Tables 16-9
Ability to Align Data Objects to TFL-Specified Boundaries to Boost Code Performance 16-10
Support for Replacing Element-wise Matrix Multiply 16-11

Code Generation Enhancements 16-11
Redundant Condition Checks 16-11
Loop Fusion 16-12

Invariant Condition Check Lifting **16-12**
Parameter Pooling for Stateflow and Interpreted MATLAB Function Blocks **16-12**
Readability Improvement for Reusable Subsystem Input and Output **16-12**

Enhanced Code Generation Optimization Using Minimum and Maximum Values **16-12**

New Model Advisor Check for Code Efficiency of Logic Blocks **16-13**

Control of Default Case Generation for Switch Statements in Generated Code for Stateflow Charts **16-13**

Improvement to Build Process for Conflicting Identifiers . . **16-14**

Update to Code Generation Verification Class cgv.Config . . **16-15**

License Names Not Yet Updated for Coder Product Restructuring **16-15**

New and Enhanced Demos **16-15**

Check bug reports for issues and fixes **16-17**

R2011a

Coder Product Restructuring **17-2**
Product Restructuring Overview **17-2**
Resources for Upgrading from Real-Time Workshop Embedded Coder **17-3**
Migration of Embedded MATLAB Coder Features to MATLAB Coder **17-4**
Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder **17-4**
Interface Changes Related to Product Restructuring **17-5**
Simulink Graphical User Interface Changes **17-5**

Data Management Enhancements and Changes **17-6**
Memory Section Enhancements **17-7**
No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects **17-7**
Parts of Data Class Infrastructure Not Available **17-8**
No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer **17-8**
Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release **17-9**

AUTOSAR Enhancements **17-9**
Calibration Parameters **17-9**
Multiple Runnables from Virtual Subsystems **17-10**
Support for Code Descriptor Elements **17-10**

SIL and PIL Enhancements **17-11**
Code Execution Profiling **17-11**
PIL Block Parameter Tuning **17-11**
Top-Model SIL/PIL and PIL Block Parameter Initialization . . **17-12**
Model Block Parameter Tuning and Model Initialization . . . **17-12**

Code Generation Enhancements **17-12**
Improved Code for Data Store Memory In-place Assignment **17-12**
Improvements to Target Function Library Replacements . . **17-12**
Improved Loop Fusion **17-13**
Improved Array Indexing **17-13**
Improvement on Matrix Parameter Pooling **17-13**
Readability Improvements Involving Data References **17-13**

Code Generation Verification (CGV) API Updates **17-13**
Support for Adding Multiple Callback Functions **17-13**
New Functionality Added to the cgv.CGV Class **17-14**

MISRA-C Code Generation Objective **17-17**

New Model Advisor Check for Code Efficiency of Lookup Table Blocks **17-17**

Enhanced Code Generation Optimization **17-18**

Target Function Library Replacement Based on Computation Method for Reciprocal Sqrt, Sine, and Cosine **17-18**

Target Function Library Support for abs, min, max, and sign functions **17-19**

C++ Encapsulation Allowed for Referenced Models in For Each Subsystems **17-19**

Improved Code Generation for Portable Word Sizes **17-19**

Improved Comments in the Generated Code **17-19**

Replacement Data Types and Simulation Mode for Referenced Models **17-20**

Changes for Embedded IDEs and Embedded Targets **17-20**
Feature Support for Embedded IDEs and Embedded Targets **17-21**
Execution Profiling during PIL Simulation **17-21**
Location of Blocks for Embedded Targets **17-21**
Location of Demos for Embedded IDEs and Embedded Targets **17-23**
Multicore Deployment with Rate-Based Multithreading . . . **17-24**
Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard) **17-24**
Changes to Frame-Based Processing **17-24**
New Support for Analog Devices Blackfin BF50x and BF51x Processors **17-26**
Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors **17-26**
Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI **17-26**
Support for Texas Instruments Delfino C2834x Processors . **17-26**
Ending Support for Altium TASKING in a Future Release . . **17-27**
Ending Support for Freescale MPC5xx in a Future Release **17-27**
Ending Support for Infineon C166 in a Future Release **17-27**
Removed Methods and Arguments **17-27**

Changes to ver Function Product Arguments **17-28**

New and Enhanced Demos **17-28**

Check bug reports for issues and fixes **17-30**

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 theEmbedded 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 `SharedParameter`s 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 `SharedParameter`s 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 (`SwAddrMethod`s) 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 `SwAddrMethod`s in Simulink.
- Specify `SwAddrMethod`s for runnable functions.
- Specify `SwAddrMethod`s 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.

`getDataDefaults`	Get AUTOSAR memory type for Simulink signals or discrete states
`mapDataDefaults`	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: • **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: • 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`

Data Type Limit Identifier Setting	Default Identifier	Command-Line Parameter
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 =
        1        2        3
        4        5        6
```

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

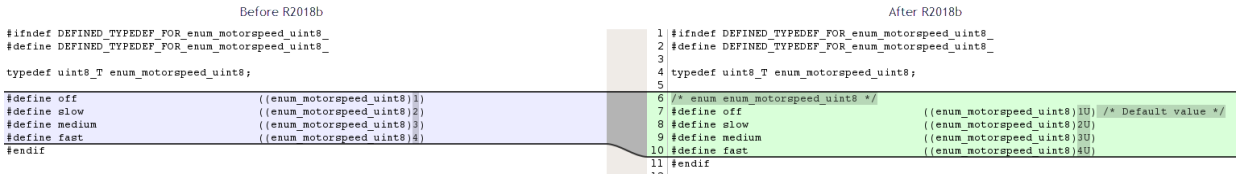
```
#ifndef DEFINED_TYPEDEF_FOR_enum_colors_int32_
#define DEFINED_TYPEDEF_FOR_enum_colors_int32_

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
```

After R2018b

```
#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
```

Appearance of unsigned integer

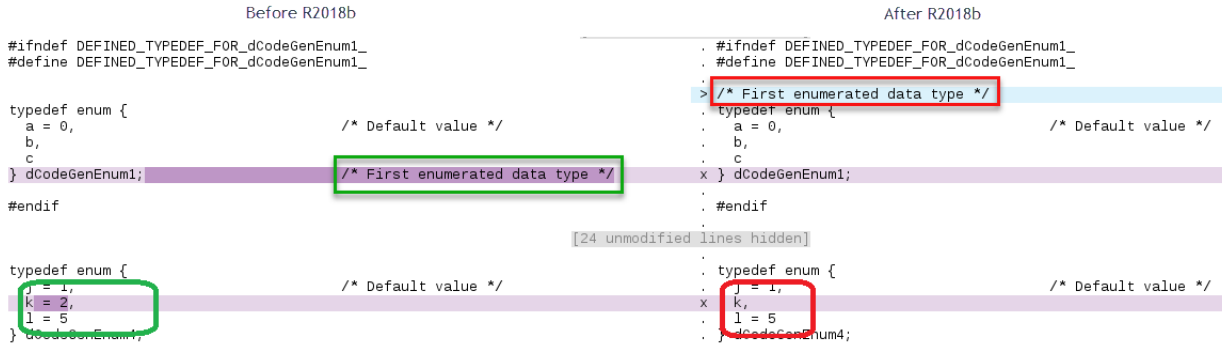
If the base type of an enumerated constant is an unsigned integer, the enumerated constant value might contain the appendix `U`. For example:



Enumerated element values and comments

Inferred values of enumerated elements might not be in the generated code.

For example, since `j=1`, `k=2`, you can infer the value of `k`, so it is not in the generated code. Comments for enumerated elements appear before the `typedef`. For example:



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
`'off'`	`'off'`
`'on'`	`'detailed'`

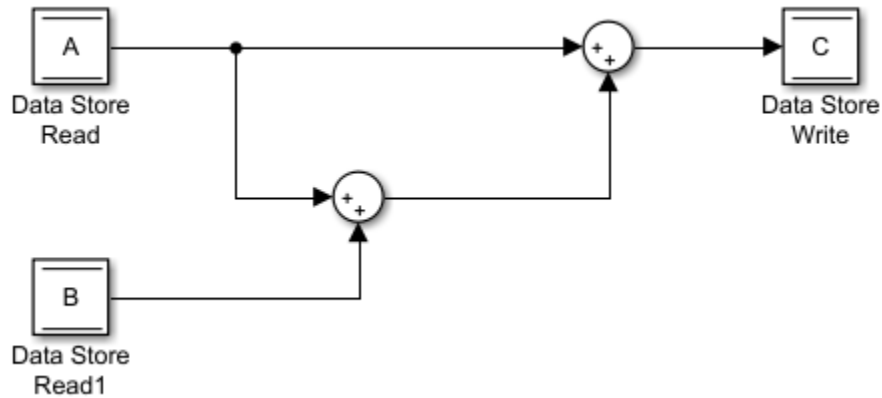
In R2018b, if you run `set_param(modelName, 'CodeProfilingInstrumentation', 'on')`, the function sets `'CodeProfilingInstrumentation'` to `'detailed'` and produces a warning.

`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 `caching_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 `caching_example.c` file.

```
static int16_T A[2];
static int16_T B;
static int16_T C[2];
RT_MODEL_caching_example_T caching_example_M_;
RT_MODEL_caching_example_T *const caching_example_M = &caching_example_M_;
void caching_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 `caching_example.c` file.

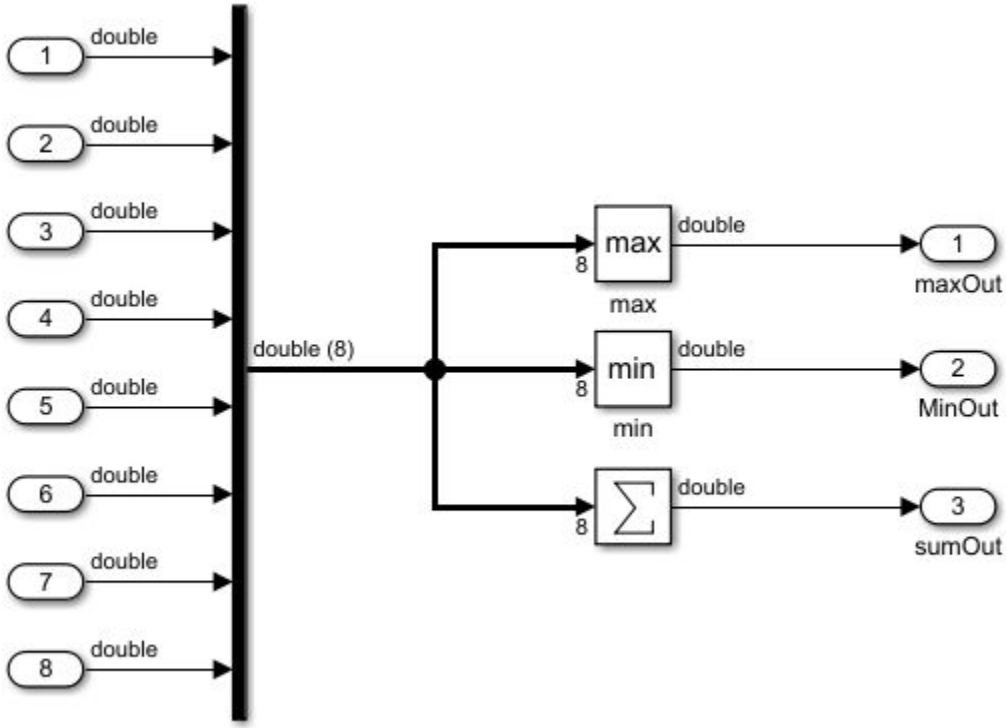
```
static int16_T A[2];
static int16_T B;
static int16_T C[2];
RT_MODEL_caching_example_T caching_example_M_;
RT_MODEL_caching_example_T *const caching_example_M = &caching_example_M_;
void caching_example_step(void)
{
  int32_T i;
  int16_T A_0;
  for (i = 0; i < 2; i++) {
    A_0 = A[i];
    C[i] = (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];
  }
```

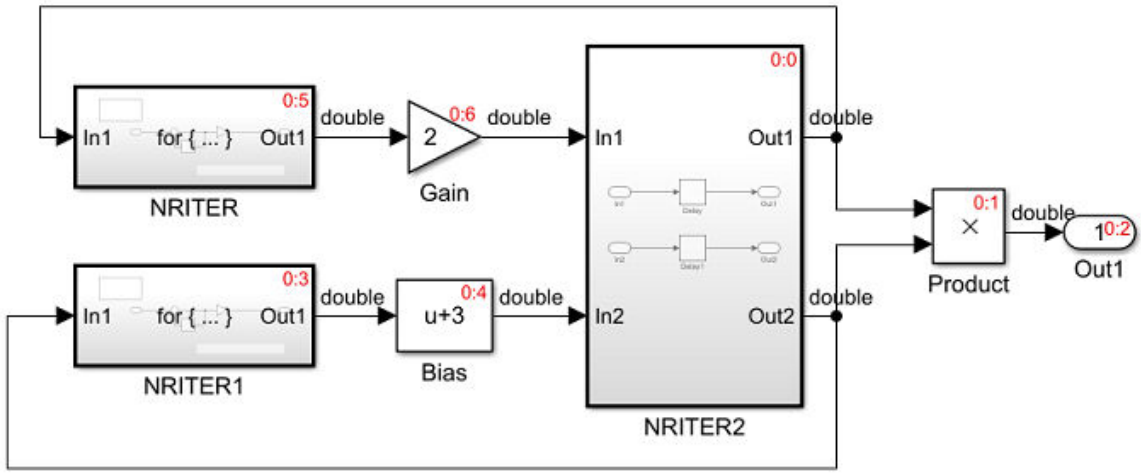
```
  rtY.sumOut = tmp;
}
```

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.

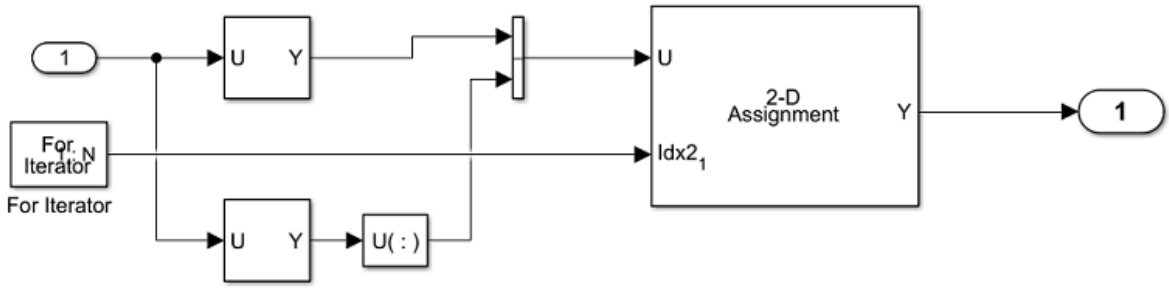
- New `Inlined ARM NEON Intrinsics` code replacement library for floating-point arithmetic operations. Use the library to inline code that you generate for ARM® Cortex-A processors.
- For the Simulink environment, there is now support for row-major order:
  - **Array layout supported by entry** menu 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.In1[8 + tmp[i]];
      rtb_VectorConcatenate[3 + i] = rtU.In1[((i << 1) + 1) << 2];
    }

    for (i = 0; i < 6; i++) {
      rtY.Out1[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;
```

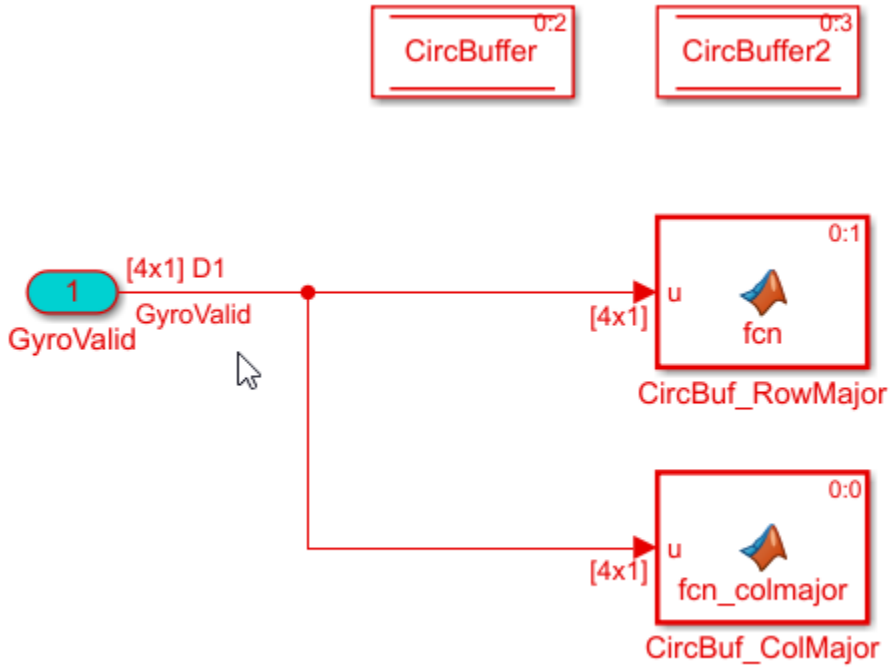
```
  int32_T i;
  for (s1_iter = 0; s1_iter < 6; s1_iter++) {
    s1_iter_0 = s1_iter * 6;
    for (i = 0; i < 3; i++) {
      rtY.Out1[i + s1_iter_0] = rtU.In1[8 + tmp[i]];
    }

    s1_iter_0 = s1_iter * 6 + 3;
    for (i = 0; i < 3; i++) {
      rtY.Out1[i + s1_iter_0] = rtU.In1[((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];
  }
}
```

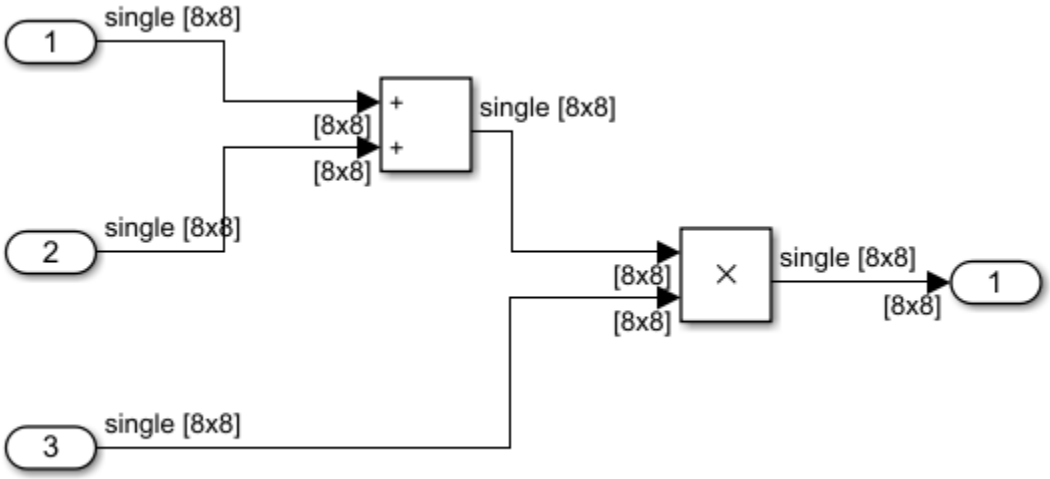
The generated code contains the temporary variables `CircBuffer2_tmp` and `CircBuffer2_tmp_0` for holding the result of the left shift operation. For more information, see “Eliminate superfluous local variables (Expression folding)” (Simulink Coder).

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” on page 2-36.

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

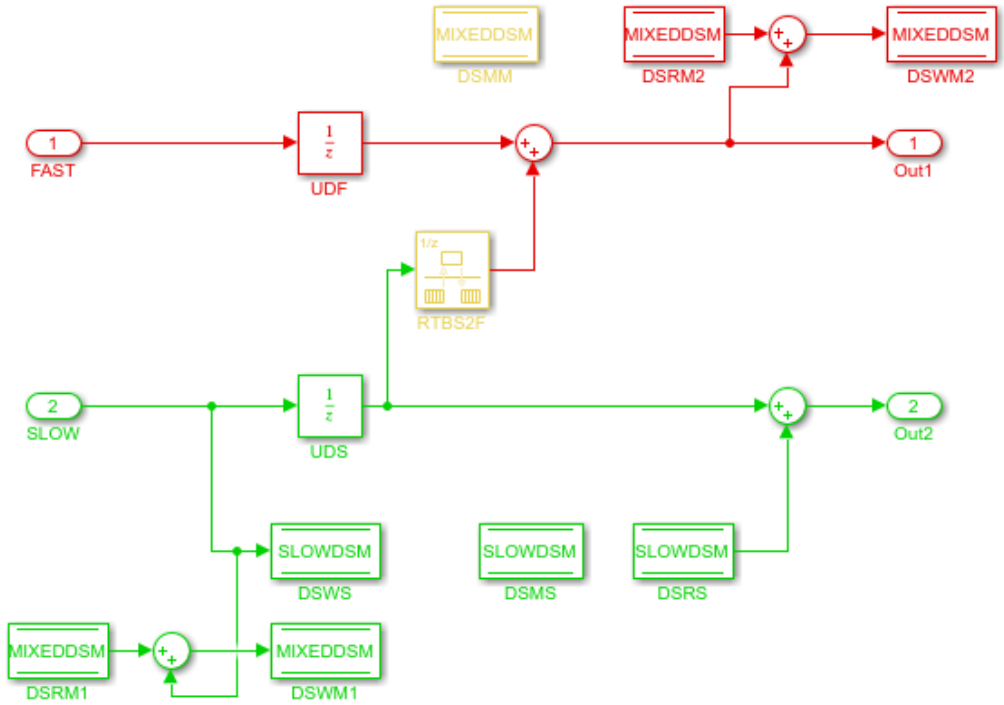
In R2018b, the configuration parameter **Parameter structure** is being removed. **Parameter structure** or `InlinedParameterPlacement` controlled how parameter data was generated for reusable subsystems.

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 hyrbrid 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2018a

Version: 7.0

New Features

Bug Fixes

Compatibility Considerations

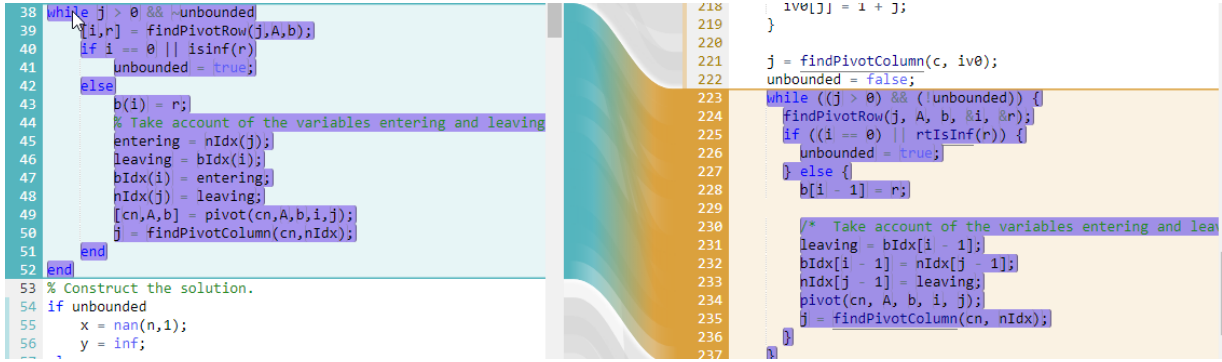
Code Generation from MATLAB Code

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.



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 2-18.

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.	• 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 for blocks not recommended for MISRA C:2012	Check now analyzes content in library linked blocks and masked subsystems.

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.

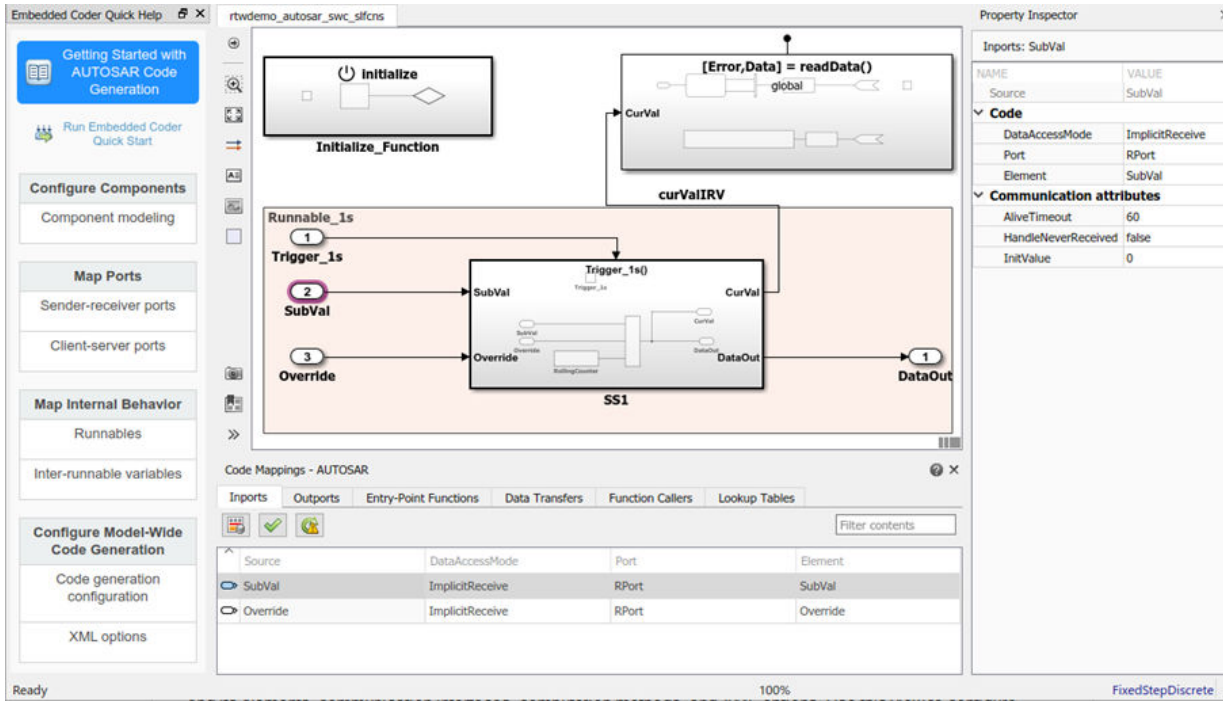
`4.3` is now the default value for the model configuration parameter **Generate XML file for schema version**.

For more information, see Select an AUTOSAR Schema.

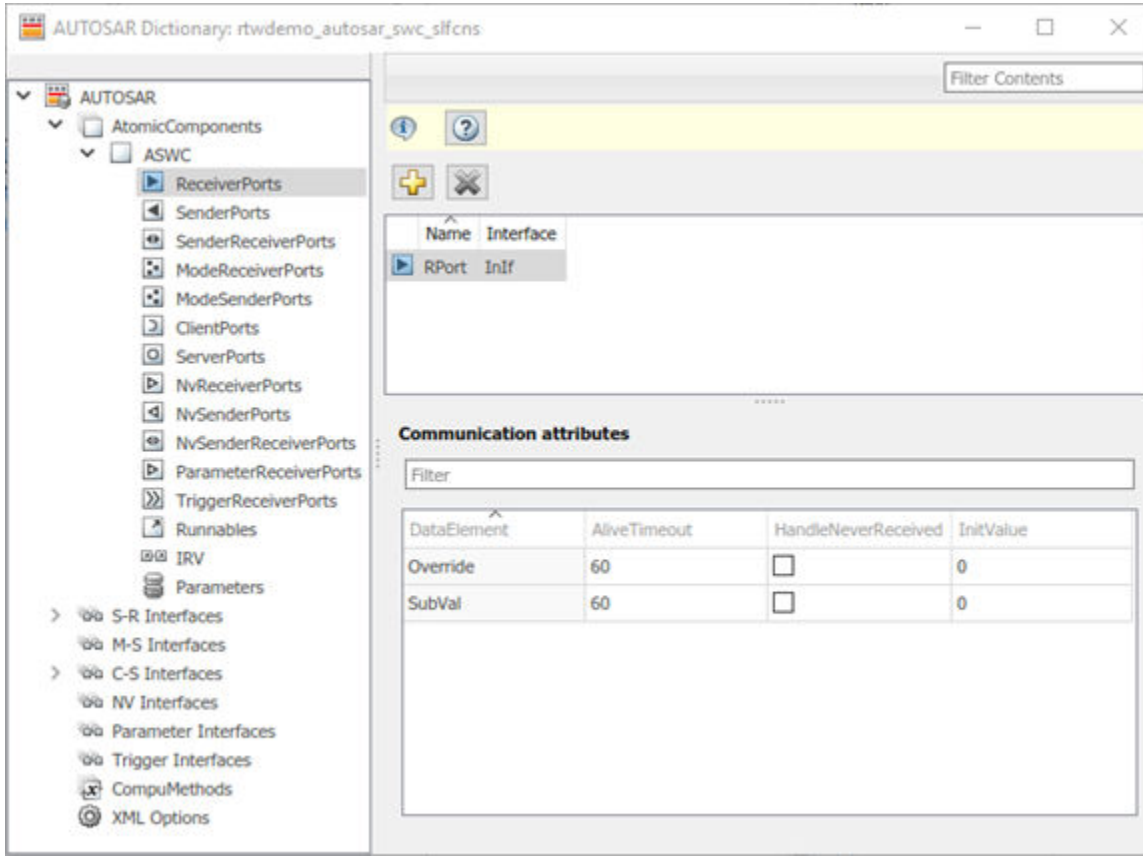
AUTOSAR Perspective: Map and configure software components by using Code Mapping Editor and AUTOSAR Dictionary

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 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 outport if the outport 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.

`getApplicationComponentNames`	Get AUTOSAR application software component names from `arxml` files
`getCalibrationComponentNames`	Get AUTOSAR calibration component names from `arxml` files
`getClientServerInterfaceNames`	Get AUTOSAR client-server interface names from `arxml` files
`getDependencies`	Get AUTOSAR `arxml` dependency files
`getFile`	Get AUTOSAR `arxml` software component file
`getSensorActuatorComponentNames`	Get AUTOSAR sensor/actuator software component names from `arxml` files
`setDependencies`	Set AUTOSAR `arxml` dependency files
`setFile`	Set AUTOSAR `arxml` 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 `get`, 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 Configure Default Code Generation for Data and Configure Default Code Generation for Functions)

To Configure	Instead of Setting	Map
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 Configure Default Code Generation for Functions)

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 busses, 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 (Simulink).

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
Memory Sections pane in configuration parameters	The pane has been removed. The parameters have been moved to **Code Generation > Advanced parameters**.	Before you open the new Code Perspective for the first time in the model, you can continue to use these configuration parameters. 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 2-18.	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 2-18).

Functionality	Result	Use Instead	Compatibility Considerations
Configuration Parameters > Code Generation > Symbols > Identifier format control > Shared utilities	The parameter has been moved and renamed to **Configuration Parameters > Code Generation > Symbols > Advanced parameters > Shared utilities identifier format**.	Before you open the new Code Perspective for the first time in the model, you can continue to use this configuration parameter. 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 2-18.	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 2-18).

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

Device vendor	Device type	Device detail	R2018a value	Previous value
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).

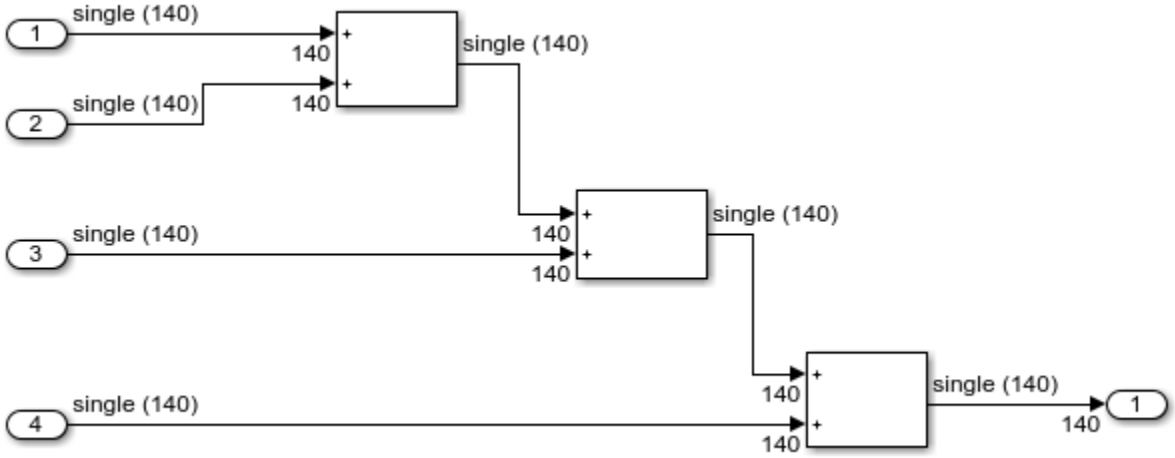
Performance

Single Instruction, Multiple Data (SIMD) Intrinsics: 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: • `Minimum (debugging)` • `Balanced with Readability` • `Maximum` (default setting)
Priority	If you set the **Level** parameter to `Maximum`, choose one of these priorities: • `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

Parameter	R2018a Pane
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

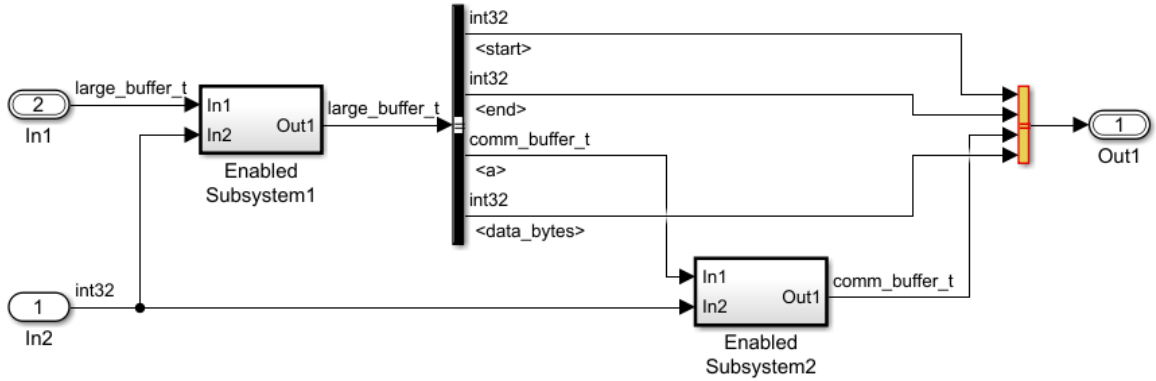
For more information, see Performance.

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:

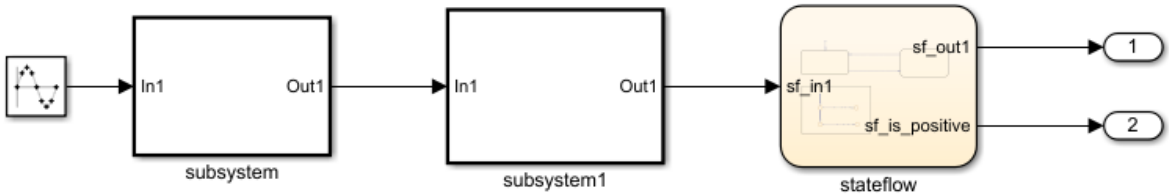
```
void ex_bus_fold_step(void)
{
  const comm_buffer_t *rtb_MultiportSwitch_a;
  if (ex_bus_fold_U.In2 == 1) {
    ex_bus_fold_Y.Out1.start = ex_bus_fold_U.In2;
    ex_bus_fold_Y.Out1.end = ex_bus_fold_U.In1.end;
    rtb_MultiportSwitch_a = &ex_bus_fold_U.In1.a;
    ex_bus_fold_Y.Out1.data_bytes = ex_bus_fold_U.In1.data_bytes;
  } else {
    ex_bus_fold_Y.Out1.start = ex_bus_fold_U.In1.start;
```

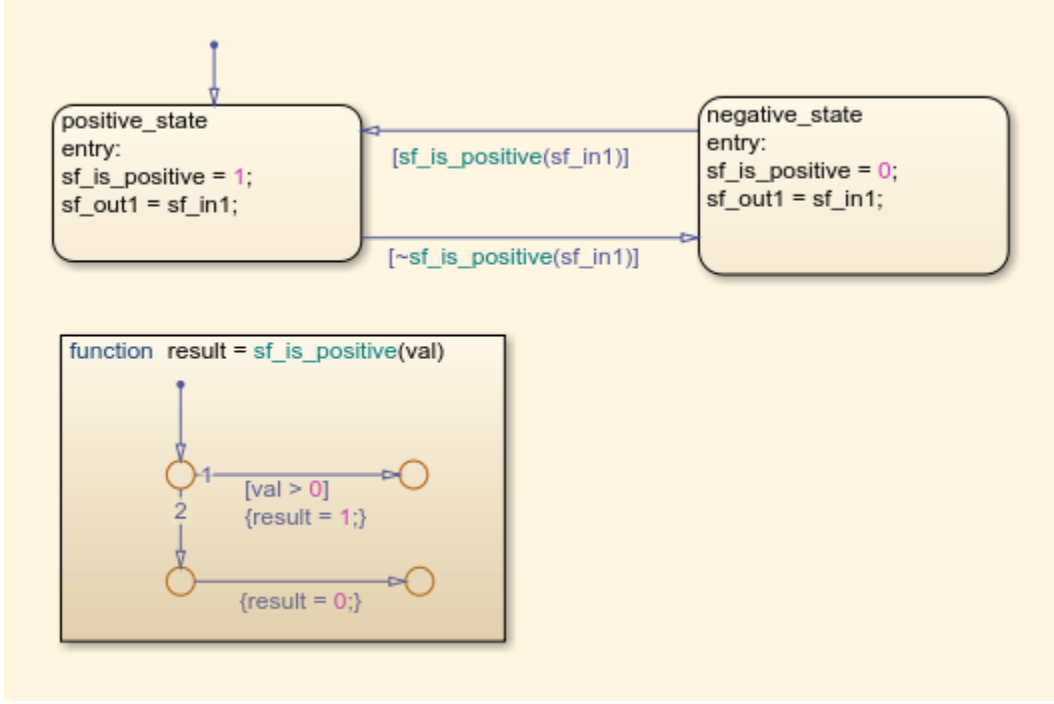
```
    ex_bus_fold_Y.Out1.end = ex_bus_fold_U.In1.end;
    rtb_MultiportSwitch_a = &ex_bus_fold_U.In1.a;
    ex_bus_fold_Y.Out1.data_bytes = ex_bus_fold_U.In1.data_bytes;
  }

  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_c1_ex_control_flow == 0U) {
    ex_control_flow_DW.is_active_c1_ex_control_flow = 1U;
    ex_control_flow_DW.is_c1_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_c1_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_c1_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_c1_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_c1_ex_control_flow == 0U) {
    ex_control_flow_DW.is_active_c1_ex_control_flow = 1U;
    ex_control_flow_DW.is_c1_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_c1_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_c1_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;
  }
}
```

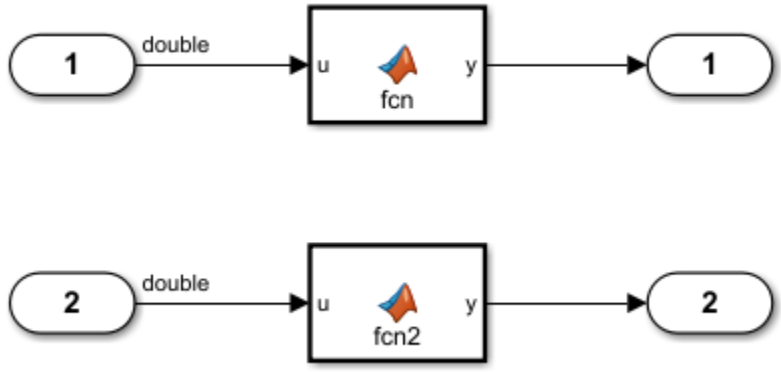
In R2018a, the variable `rtb_sf_is_positive` and its corresponding data copy are not in the generated code. In the control flow code, the writes occur directly to `ex_control_flow_Y.Out2`.

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 `sfn_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 2-36, 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:

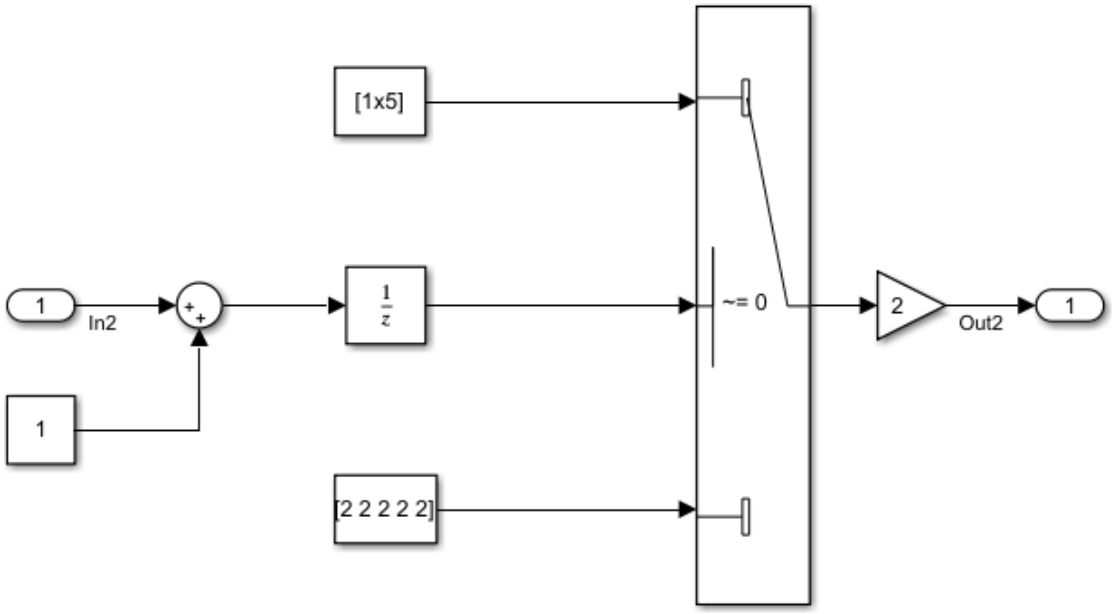
  - **Array layout supported by entry** menu in the Code Replacement Tool for creating row-major code replacement table entries. The menu appears when you set **Argument type** to `Matrix`. You can set **Array layout supported by entry** to `Column-major` (default), `Row-major`, or `Column-and-Row`.
  - 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

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` function `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	Description	Addresses MISRA C: 2012
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	Flags the following parameter settings: • 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 `ERTFilePackagingFormat` is set to `Modular`. • Parameter `PreserveStaticInFcnDecls` is set to `off`. hisl_0060: Configuration parameters that improve MISRA C:2012 compliance reflects these parameter settings.
Check for switch case expressions without a default case	Check can be executed on library models. Check can exclude blocks when you have Simulink Check.

Check	Description of Modification to the Check
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	Description	Addresses 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**	• 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.	• 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.	• 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.	• CERT C, FLP00-C • CWE, CWE-697
Check integer word length	Identifies integer word lengths that do not comply with hardware implementation settings.	• 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.	• CERT C, MSC07-C • CWE, CWE-561

Check	Description	Addresses Secure Coding Standards
Detect Integer Overflow	Identifies operations that exceed the data type range for integer or fixed-point operations.	• 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.	• 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	• 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.	• 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” (Simulink Coder).

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
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:

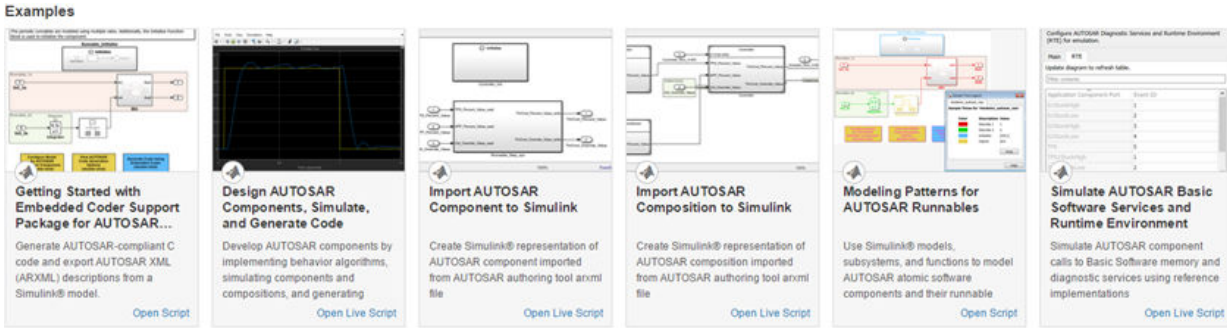
```
modelex_DW.Delay_DSTATE[0] = (modelex_P.Gain_Gain_o *
     modelex_DW.Delay_DSTATE[0]) >> 6;
```

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.



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.

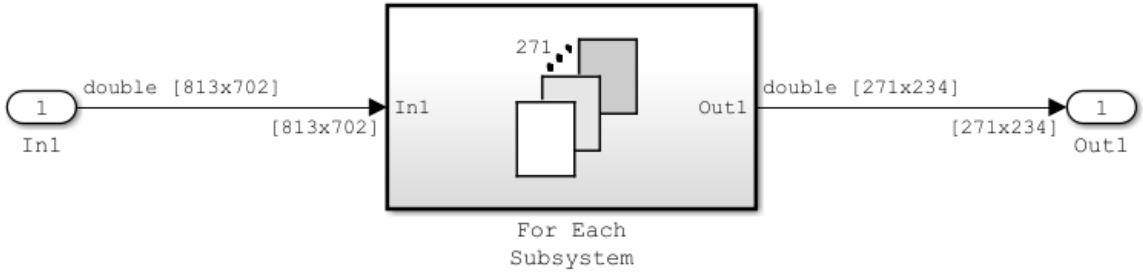
For For Each Subsystem blocks, the code generator can perform these optimizations:

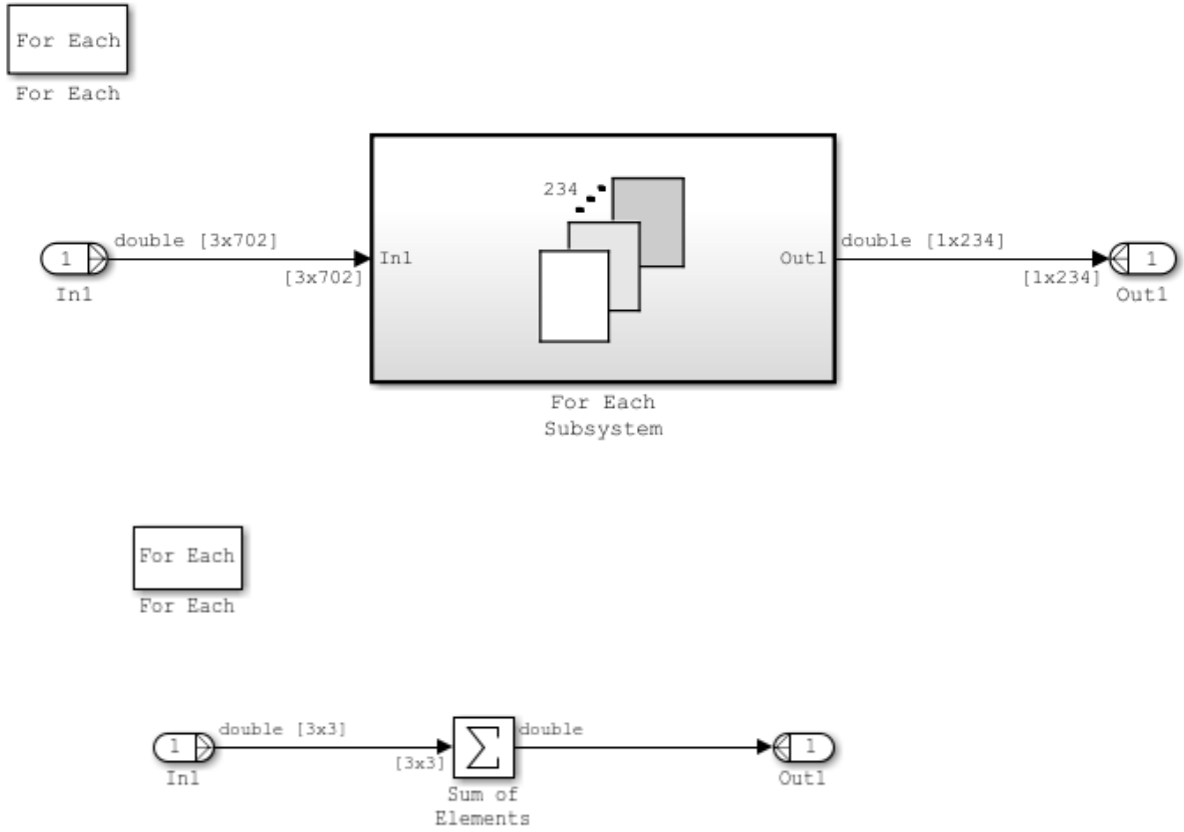
- Combine more `for` loops.
- Unroll more `for` loops whose size is under the **Loop Unrolling Threshold** parameter value.
- Generate fewer data copies for For Each Subsystem block input and output ports.

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` .

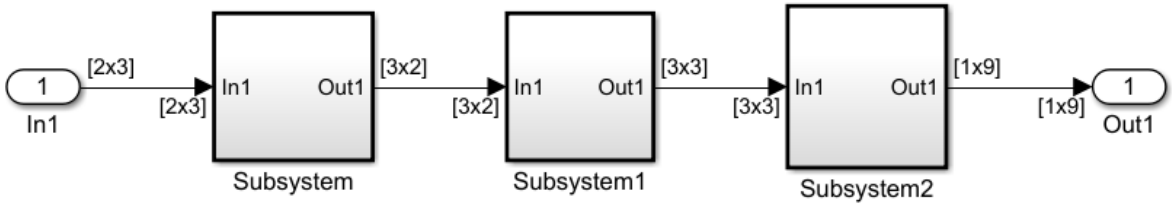
In R2017b, the code generator produces this code:

```
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_13` 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 `difSizeAutoReuse_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.

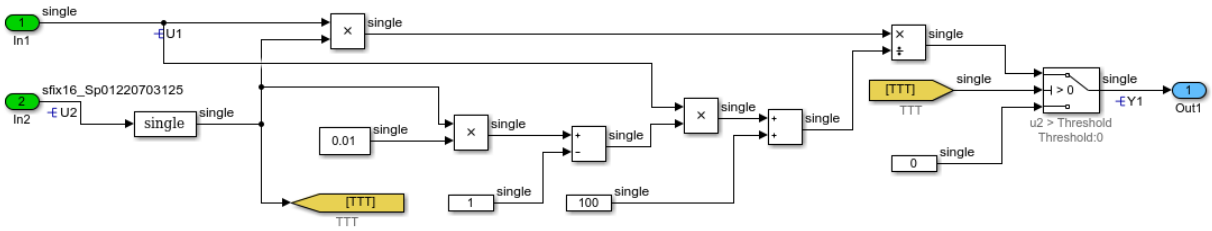
- Control how signal and state data interface with externally-written code.

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;
  }
}
```

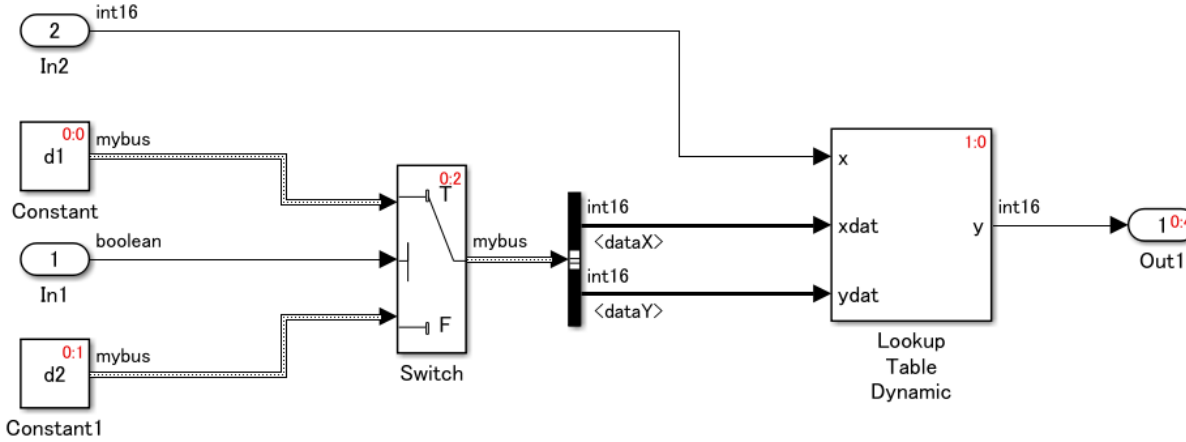
The `M01_step` function contains the variable `tmp` to hold the value of the subexpression `U2*0.0122070313F`.

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'
   *  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);
}
```

The `structure_pointer_conversion.h` file contains the same variable declaration as was in R2017a.

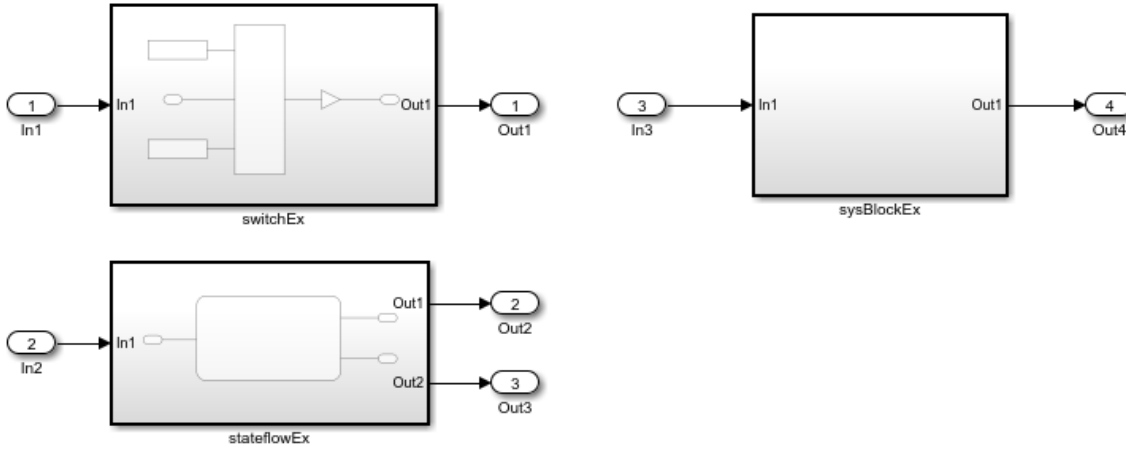
```
typedef struct {
  int16_T dataX[100];
  int16_T dataY[100];
} mybus;

#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:

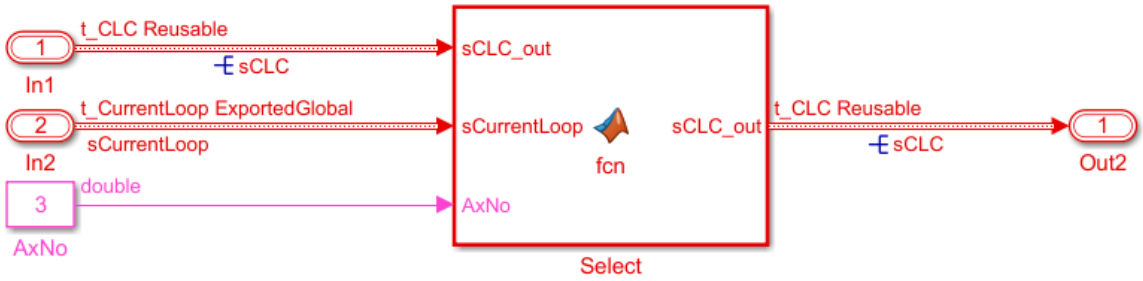
- Reuse the same variable for the input, output, and state of a Unit Delay block by executing the Unit Delay block before upstream blocks.
- Combine more `for` loops by executing blocks together that have the same size.
- Eliminate data copies by executing blocks together that meet these conditions:
  - 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.

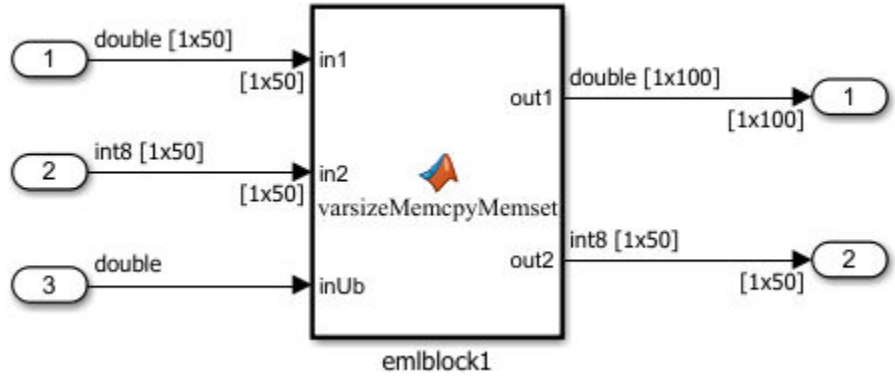
Execution-Time Profiling: Display time units in code execution profiling report and Simulation Data Inspector

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);

  /* Outport: '<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`.

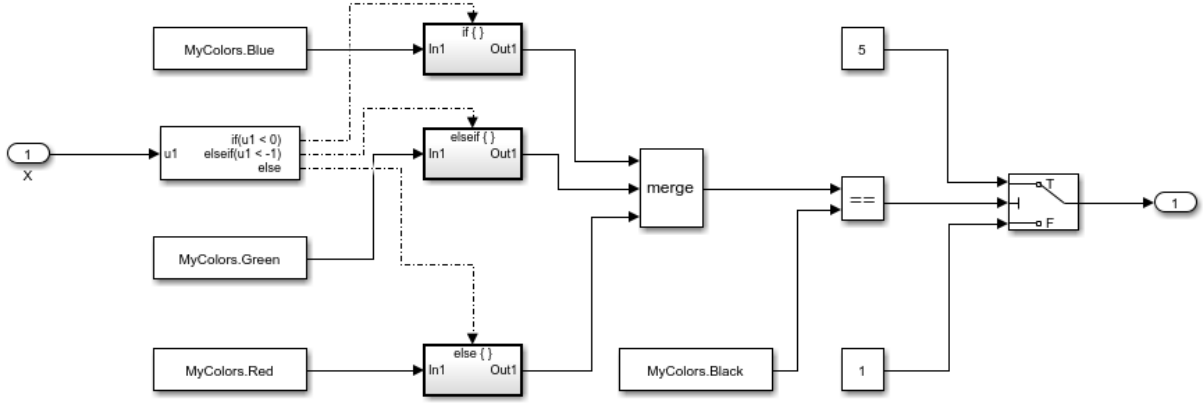
Code Replacement: Apply MustHaveZeroNetBias and SlopesMustBeTheSame properties for fixed-point operator code replacement

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 setTflCOperationEntryParameters.

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” (Simulink Coder)
- 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 endianess, 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2017a

Version: 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'`	For a given set of input values, the function: • Runs a MATLAB function on your development computer. • Performs PIL executions of generated MATLAB code on your target hardware with `config.TargetLang` settings `'C'` and `'C++'`. The function compares results from the MATLAB function run and the PIL executions. If the function detects differences, it produces an error.

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.

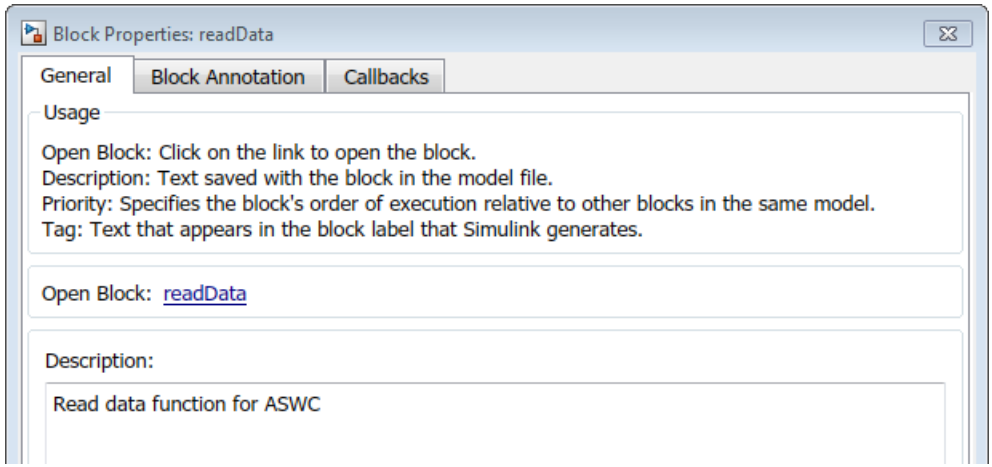
```
obj = arxml.importer('mySWC.arxml')
createComponentAsModel(obj,'/pkg/swc/ASWC','ModelPeriodicRunnablesAs','AtomicSubsystem')
```

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 `arxml` 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.

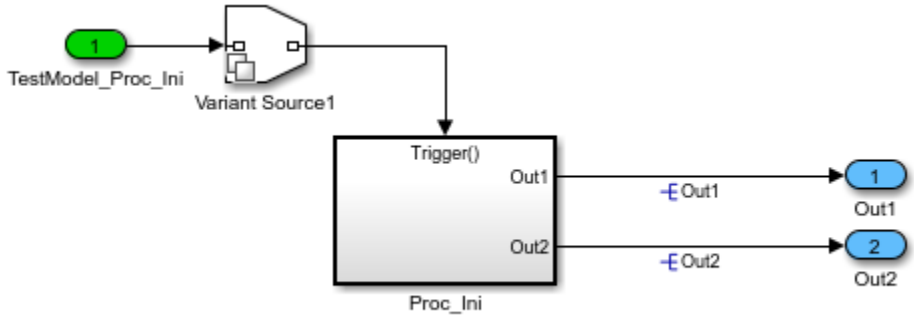
Code generation support for Variant Subsystems containing global signals

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>/OutportBufferForOut1' */
  Out1 = 1.0F;

  /* SignalConversion: '<S1>/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' */
}
```

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`.

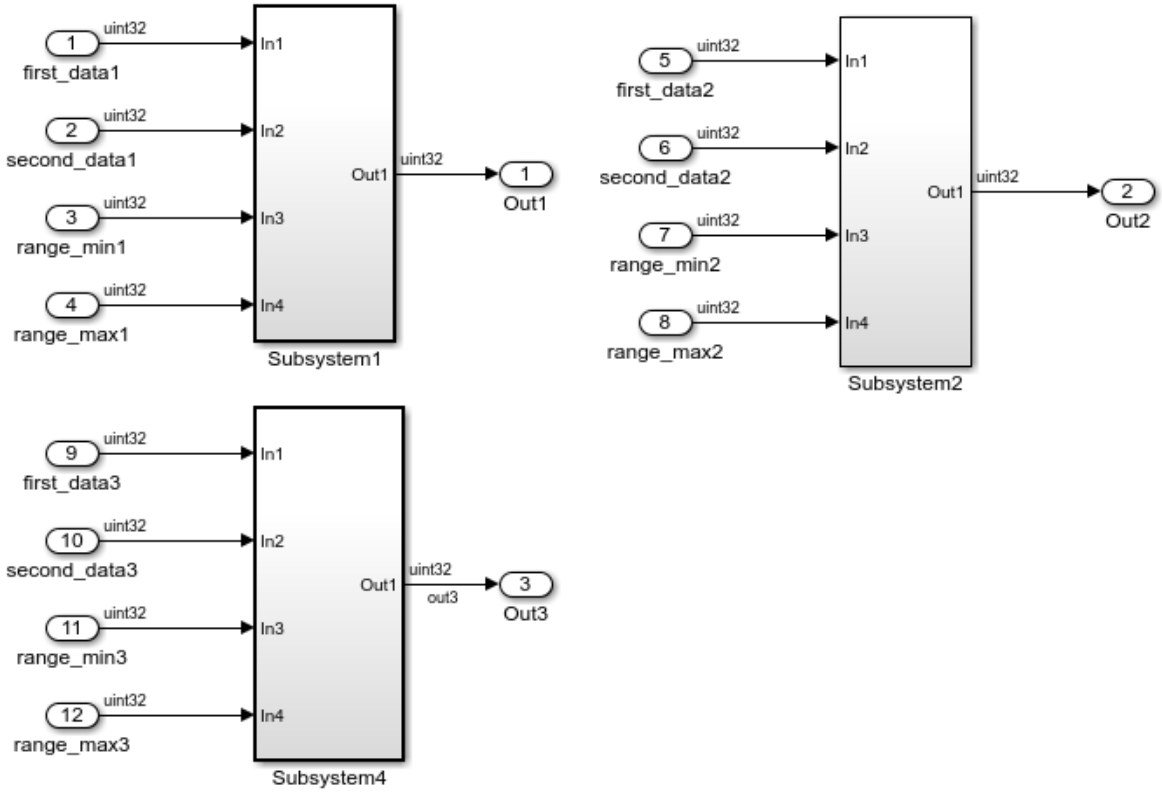
Data, Function, and File Definition

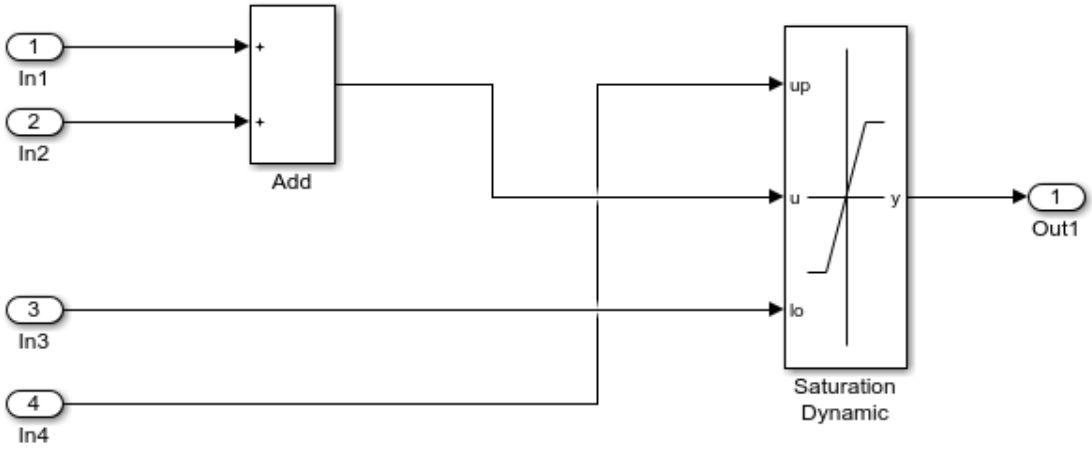
Function Interface: Return nonvoid type for scalar output of reusable functions

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.


1
uint32
first_data1
2
uint32
second_data1
3
uint32
range_min1
4
uint32
range_max1
In1
In2
In3
In4
Out1
uint32
1
Out1
Subsystem1
5
uint32
first_data2
6
uint32
second_data2
7
uint32
range_min2
8
uint32
range_max2
In1
In2
In3
In4
Out1
uint32
2
Out2
Subsystem2
9
uint32
first_data3
10
uint32
second_data3
11
uint32
range_min3
12
uint32
range_max3
In1
In2
In3
In4
Out1
uint32
out3
3
Out3
Subsystem4



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	*codeGenFolder*`/`*modelName*`_ert_rtw`	*anchorFolder*`/`*modelName*`_ert_rtw`
Referenced model	Model reference	*codeGenFolder*`/slprj/ert/`*refModelName*	*anchorFolder*`/slprj/ert/`*refModelName*
Subsystem	Standalone	*codeGenFolder*`/`*subSysName*`_ert_rtw`	*anchorFolder*`/`*subSysName*`_ert_rtw`

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.
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

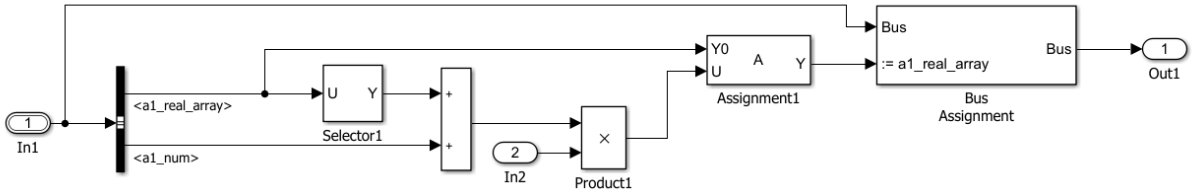
In R2017a, the generated code contains less temporary variables and associated data copies for modeling patterns involving Bus Assignment, Data Store Read, Data Store Write, and Model blocks. These optimizations conserve RAM usage and improve code execution speed. The following examples highlight these improvements:

- “Data copy reduction for Bus Assignment block” on page 4-25
- “Data copy reduction for Data Store Read and Data Store Write blocks” on page 4-27
- “More efficient code for Model blocks” on page 4-30

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' */
}
```

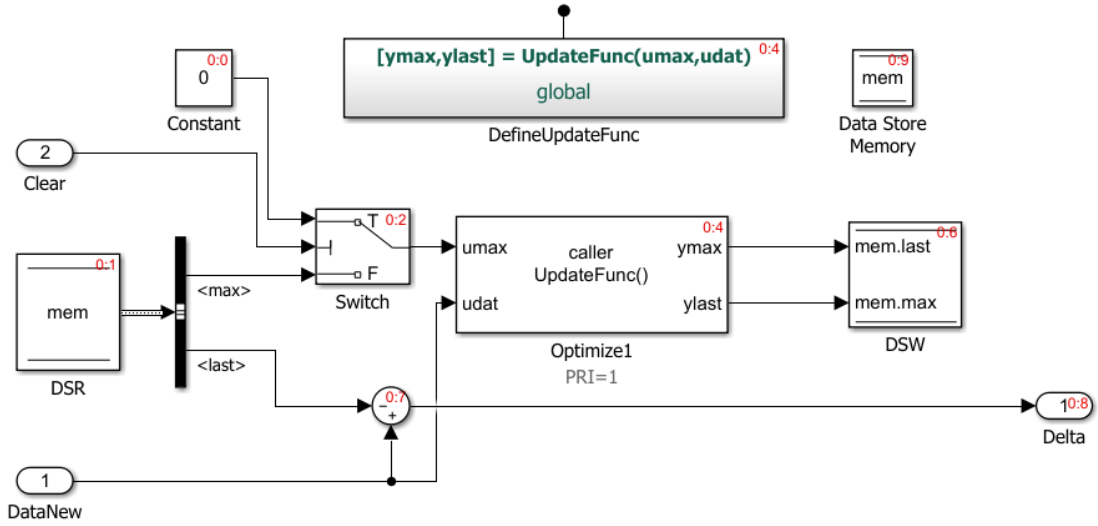
The generated code does not contain the temporary array `rtb_Assignment1` for holding data. The generated code directly assigns the data to `bus_assignoptim_Y.Out2.dbl_real_array`.

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:

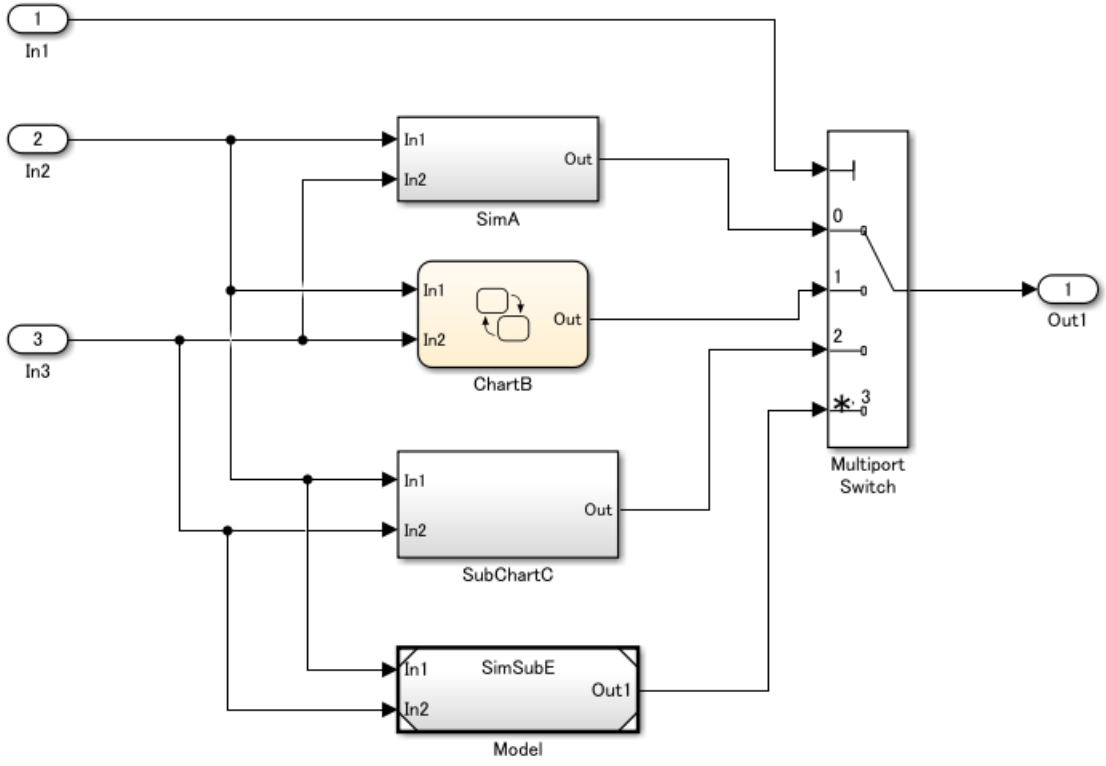
- A Simulink Function internally writes to the Data Store Memory block.
- The Data Store Read or Data Store Write blocks select elements of an array from the Data Store Memory block.
- The Data Store Memory block has a custom storage class.
- The Data Store Read and Data Store Write blocks occur on the 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:OvSatOk*/ 5U;
    if (u16_Out_m < model_ref_U.In2) {
      u16_Out_m = MAX_uint16_T;
```

```
    }
  } else {
    u16_Out_m = model_ref_U.In2 + /*MW:OvSatOk*/ 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:OvSatOk*/ 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:OvSatOk*/ 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:OvSatOk*/ 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:OvSatOk*/ 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;
```

```
  }
}
```

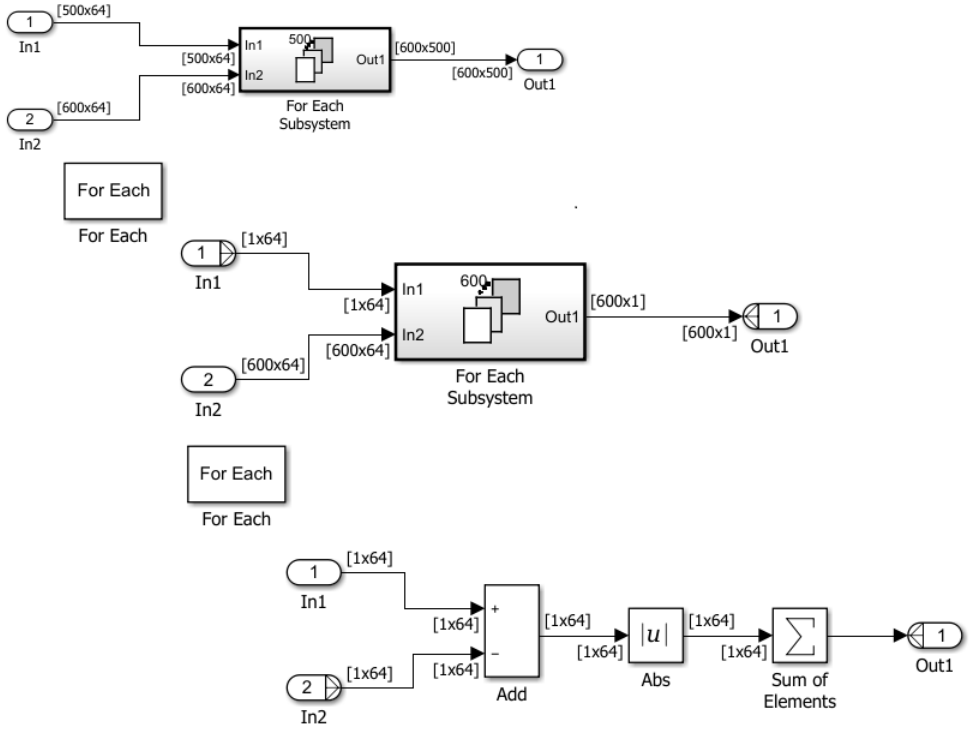
In the `model_ref_step` function, there are two local variables instead of three local variables which conserves stack space. Each `switch-case` statement includes the corresponding `if-else` statement. Including the `if-else` statements in the `switch-case` statements increases code execution speed 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:

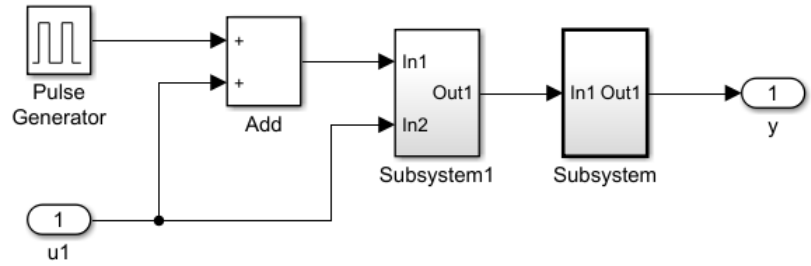
```
(counter >> 1) >= DELAY
```

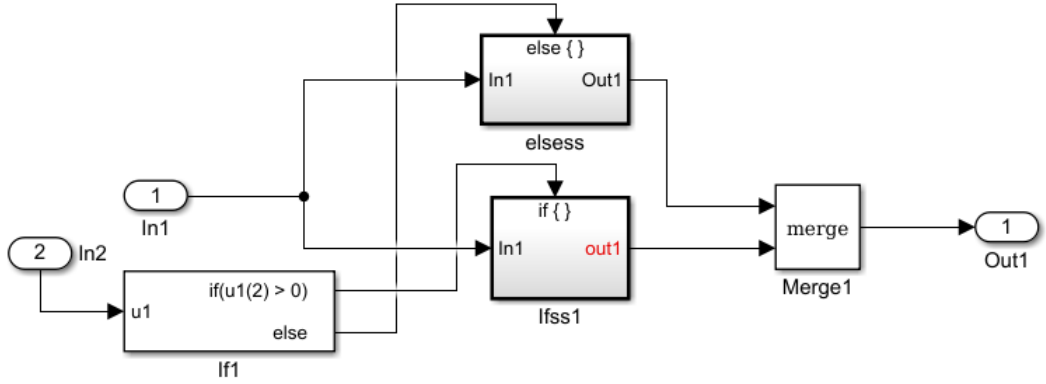
This code contains fewer operations and does not include floating-point operations.

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.Merge1[0], &rtb_Add[0], sizeof(real_T) << 6U);
  } else {
    for (i = 0; i < 64; i++) {
      cond_reuse_B.Merge1[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.Merge1`.

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 Outport 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 Outport 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`.

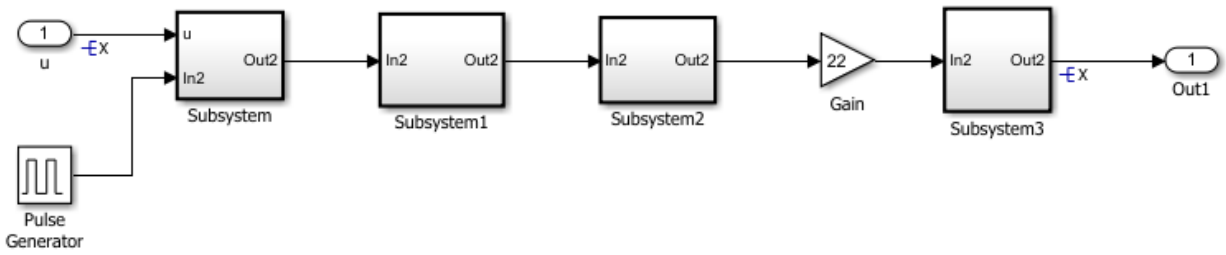
These optimizations reduce data copies in the generated code thereby conserving RAM and ROM consumption and improving code execution speed.

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 Outport 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 Outport 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 {
```

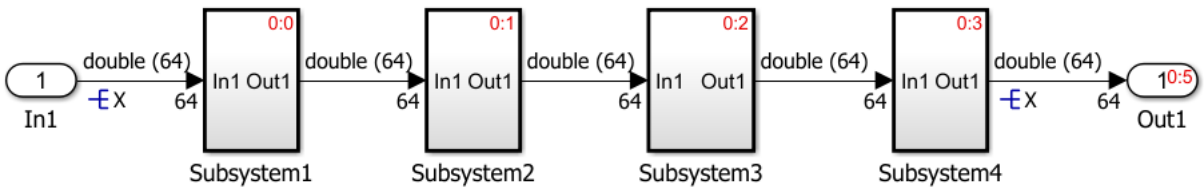
```
    Chainbuffer_DW.clockTickCounter++;
  }

  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_Gain_n[64];
  real_T rtb_Gain_m[64];
  subsreuse_Subsystem1((&(X[0])), rtb_Gain_n, (P_Subsystem1_subsreuse_T *)
                       &subsreuse_P.Subsystem1);
  subsreuse_Subsystem2(rtb_Gain_n, rtb_Gain_m, (P_Subsystem2_subsreuse_T *)
                       &subsreuse_P.Subsystem2);
```

```
  subsreuse_Subsystem2(rtb_Gain_m, rtb_Gain_n, (P_Subsystem2_subsreuse_T *)
                       &subsreuse_P.Subsystem4);
  subsreuse_Subsystem1(rtb_Gain_n, (&(X[0])), (P_Subsystem1_subsreuse_T *)
                       &subsreuse_P.Subsystem3);
}
```

The code contained two temporary variables, `rtb_Gain_n` and `rtb_Gain_m`, 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])), (P_Subsystem1_subsreuse_T *)
                       &subsreuse_P.Subsystem1);
  subsreuse_Subsystem2((&(X[0])), (&(X[0])), (P_Subsystem2_subsreuse_T *)
                       &subsreuse_P.Subsystem2);
  subsreuse_Subsystem2((&(X[0])), (&(X[0])), (P_Subsystem2_subsreuse_T *)
                       &subsreuse_P.Subsystem3);
  subsreuse_Subsystem1((&(X[0])), (&(X[0])), (P_Subsystem1_subsreuse_T *)
                       &subsreuse_P.Subsystem4);
}
```

The generated code uses one global variable X for the input and output of each function.

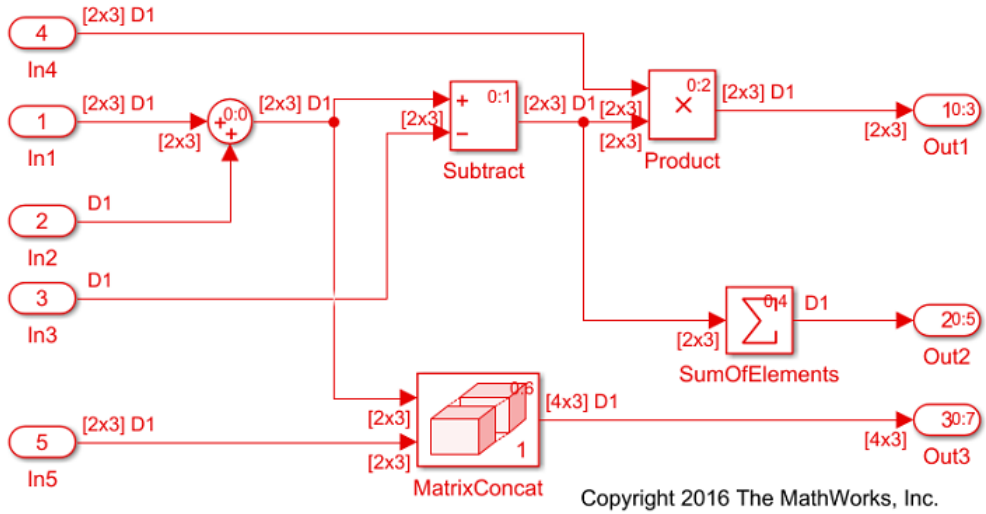
Improved buffer reuse due to changes in block execution order

In R2016b, if you specified a signal for reuse, the code generator changed the block operation order so that buffer reuse occurred.

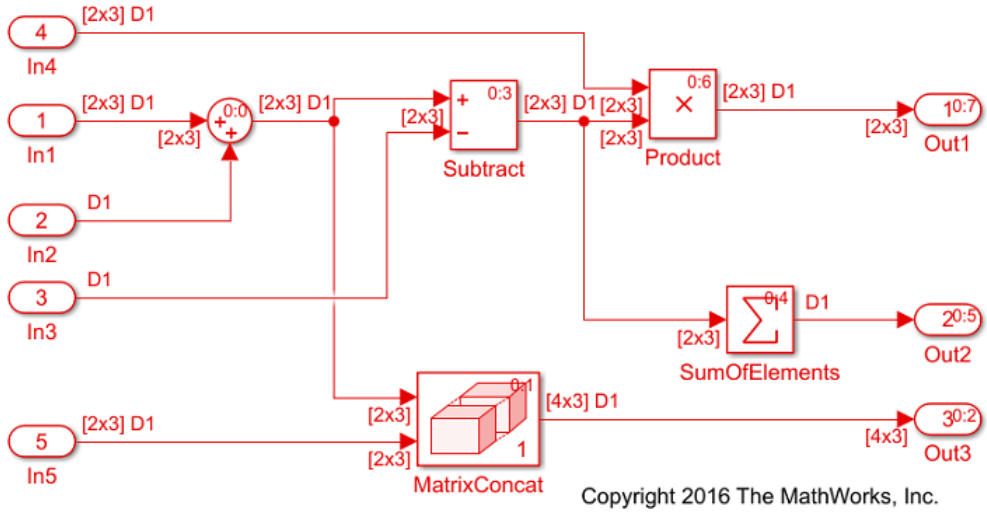
In R2017a, even if you do not specify a signal for reuse, the code generator can change the block operation order so that buffer reuse can occur. If the generated code contains extra buffers, you can try to eliminate them by setting the **Optimize block operation order in the generated code** parameter 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.



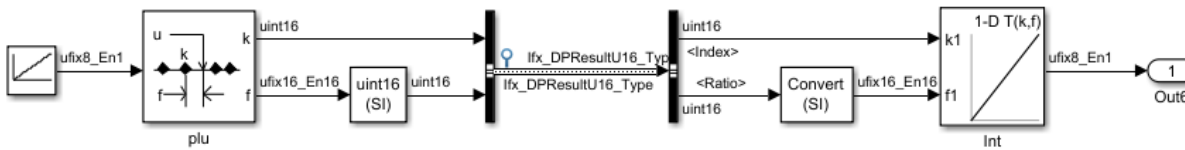
Note To implement buffer reuse, the code generator does not violate user-specified block priorities.

For more information, seeRemove 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.Ifx_DPResultU16_Type_h.Index = dpResult.Index;
  bus_creator_ex_B.Ifx_DPResultU16_Type_h.Ratio = dpResult.Ratio;
  dpResult_0.Index = bus_creator_ex_B.Ifx_DPResultU16_Type_h.Index;
  dpResult_0.Ratio = bus_creator_ex_B.Ifx_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:

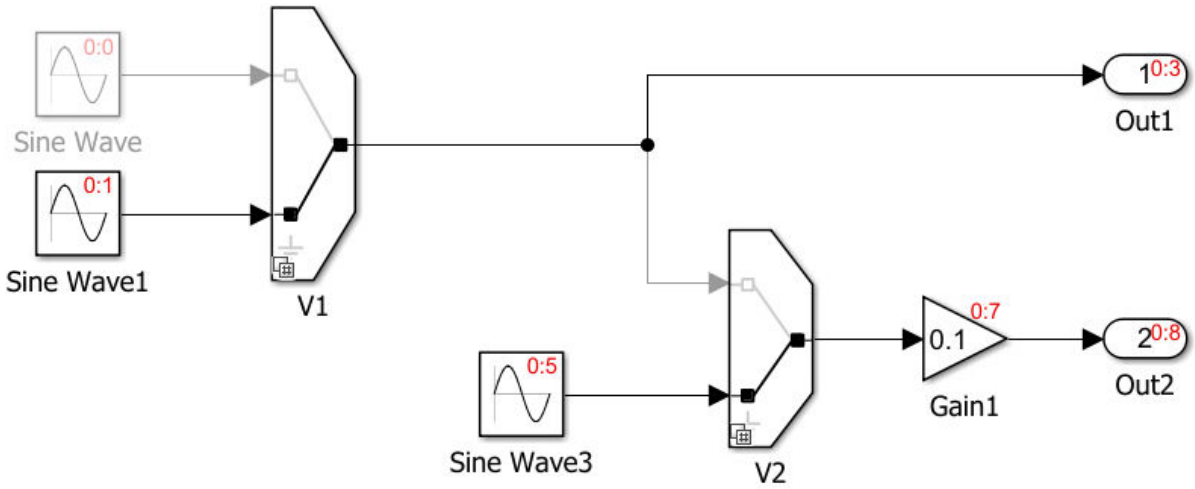
```
void bus_creator_ex_step(void)
{
  Ifx_DPResultU16_Type dpResult;
  Ifx_DPSearch_u8(&dpResult,
                  bus_creator_ex_ConstP.Vector_Value[bus_creator_ex_DW.Output_DSTATE],
                  2U, (*Rte_CData_BP1_SlopeBiasScaling_0_8_0p5_0()));
  bus_creator_ex_B.Ifx_DPResultU16_Type_h.Index = 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'`	New in R2017a. For a Simulink top model, the function: • 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'	New in R2017a. For a Simulink Model block, the function: • 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`. The function compares results from SIL and PIL mode simulations to results from a normal mode simulation.
'verifyModelBlock'	Updated in R2017a. 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`. Previously, **Language** was set to `C` .

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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2016b

Version: 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 5-5
- “AUTOSAR port-based and internal calibration parameters” on page 5-5

- “AUTOSAR SwRecordLayouts for lookup tables” on page 5-6
- “AUTOSAR SwAddrMethods for measurement and calibration tools” on page 5-6

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 5-5.
- 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 (`ParameterDataPrototype`s) 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.

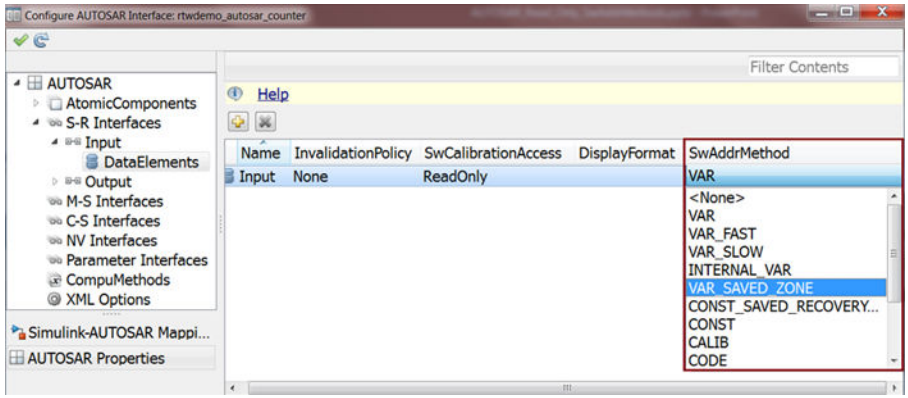
For more information, see Configure AUTOSAR Data for Measurement and Calibration.

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

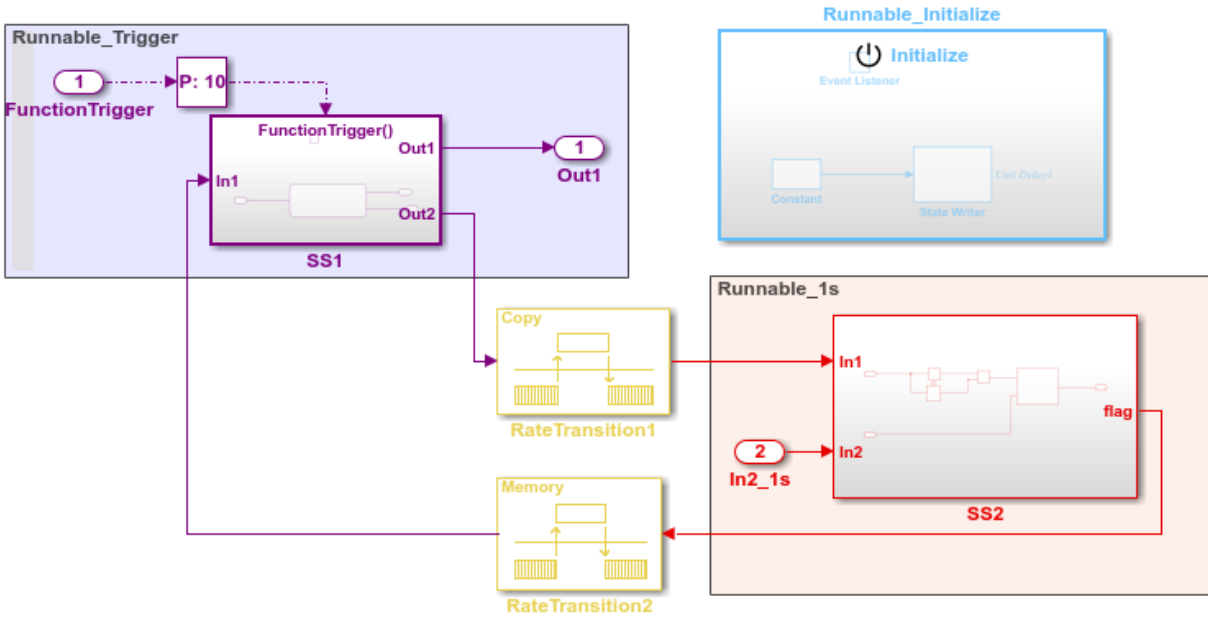
Embedded Coder supports AUTOSAR code generation for the model architectures described in the Japan MBD Automotive Advisory Board (JMAAB) document *Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow - Version 4.01*. The document is available from the MAAB Web page at `https://www.mathworks.com/solutions/automotive/standards/maab.html`.

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/Input</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.

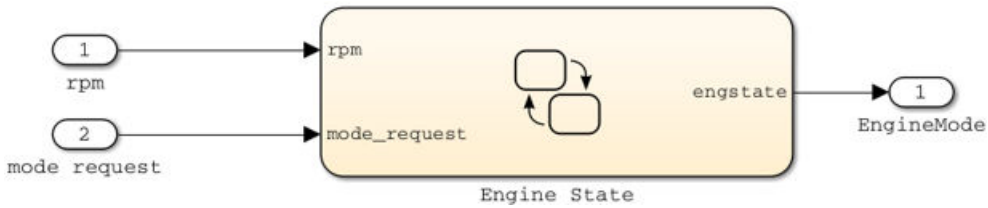
```
void Runnable_Step(void)
{
 ...
    /* Gain: '<S1>/Gain' incorporates:
     *  Inport: '<Root>/Input'
     *
     * Block description for '<S1>/Gain':
     *  This block references an AUTOSAR calibration parameter, which is
     *  accessed using the AUTOSAR Rte_Calprm function signature.
     */
    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.

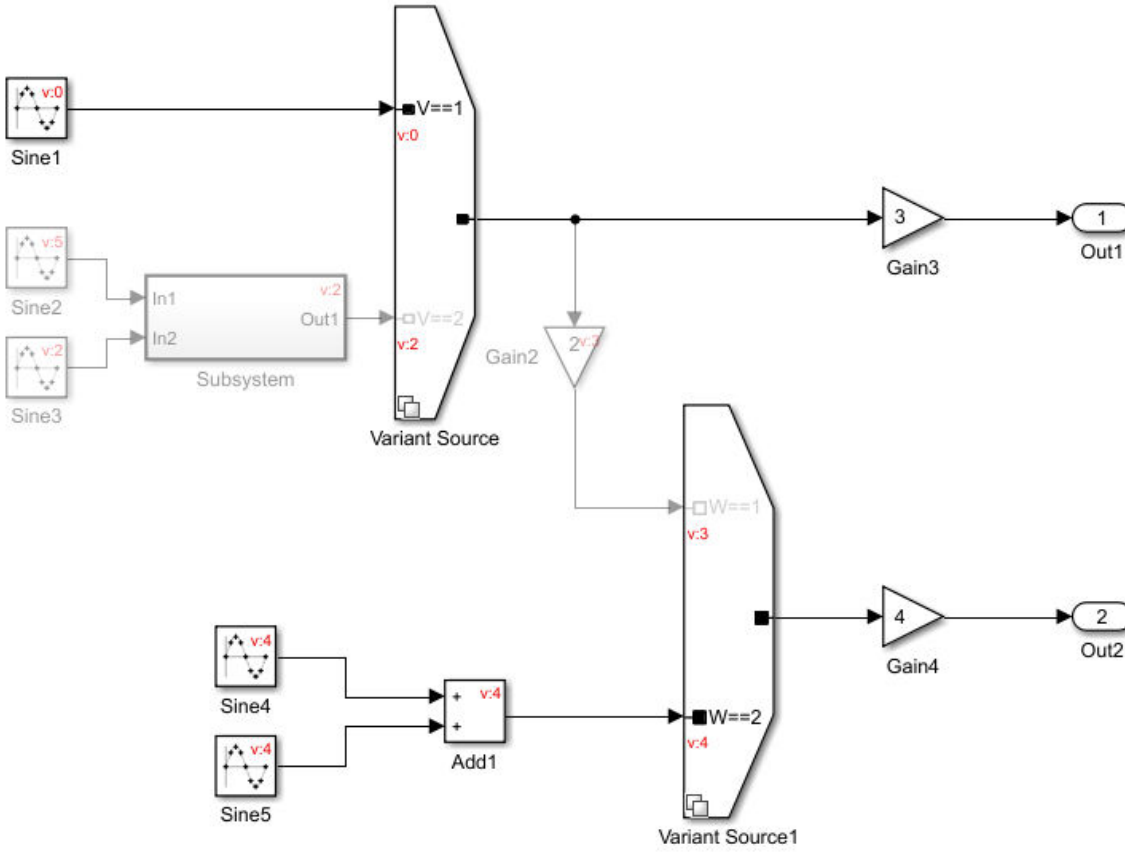
Additional guarding of global data for variant systems

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.

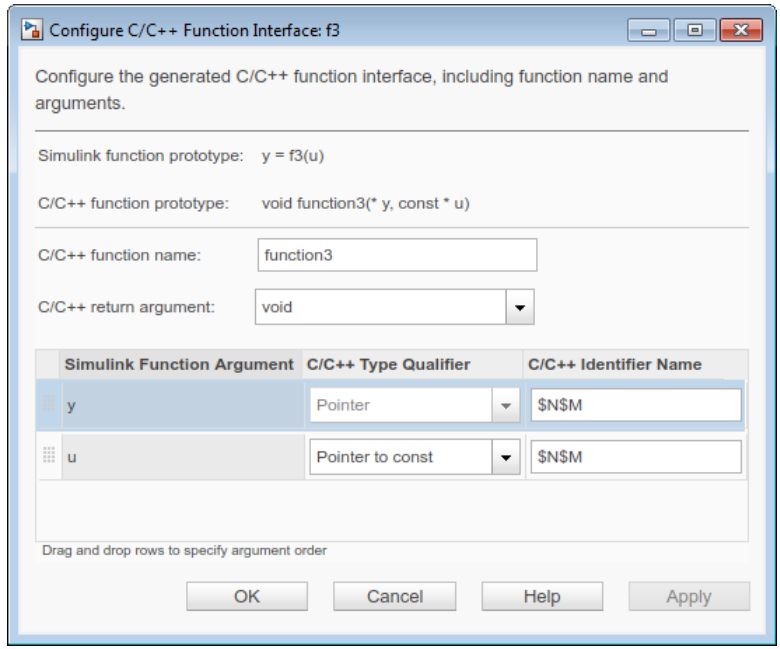
Data, Function, and File Definition

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

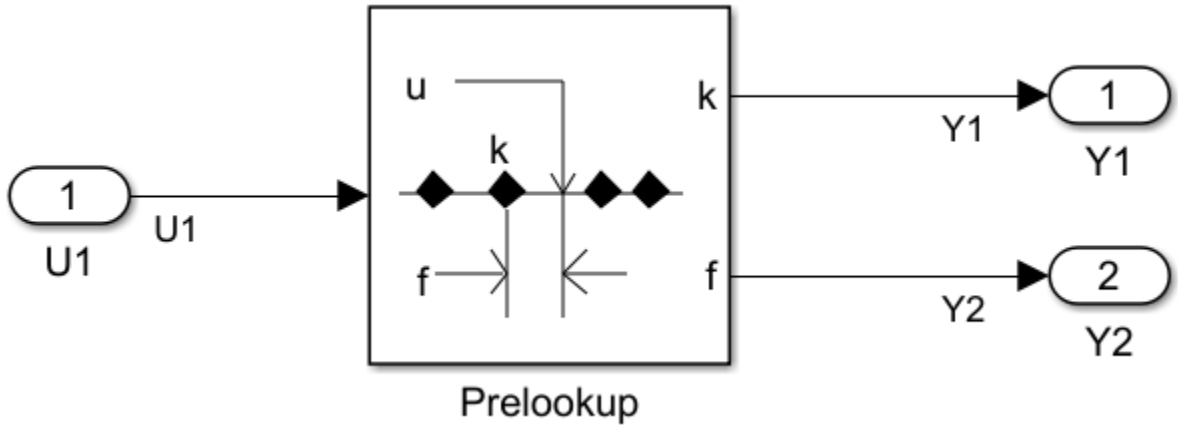
Previously, when you specified custom function argument names for a Simulink function by using the Subsystem method arguments parameter, for arguments that were input and output, the generated code inserted a `y` for the `$I` macro. In R2016b, the generated code inserts a `uy`.

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.

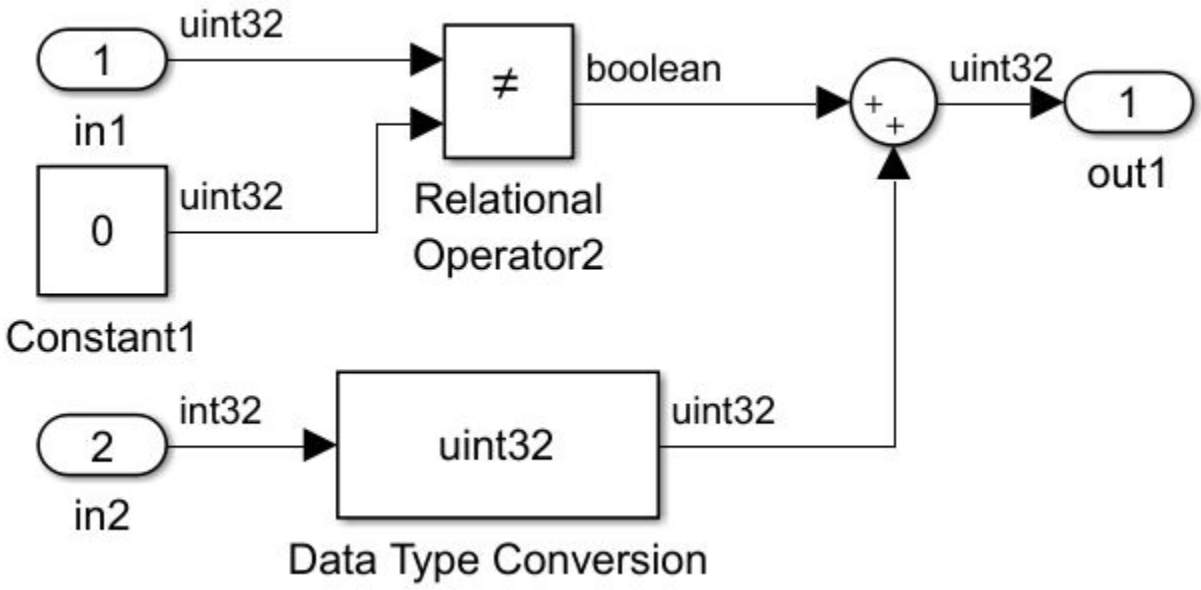
In R2016b, in the `misra1.c` file, for the Prelookup block, the code generator produces this code:

```
if (u < ((int32_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);
```

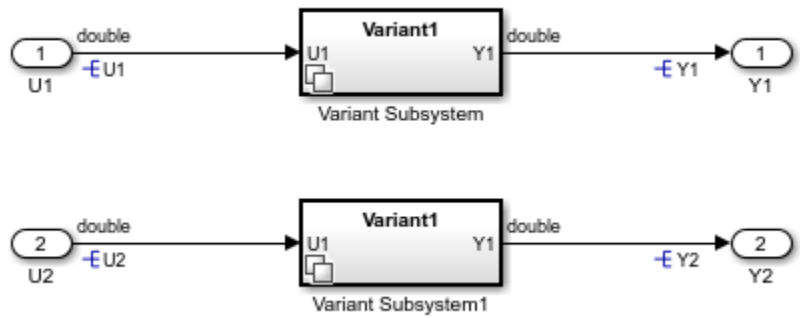
This code is compliant with MISRA C:2012 Rules 10.5 and 10.8 because the ternary operator prevents a cast from a Boolean to an unsigned integer.

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.

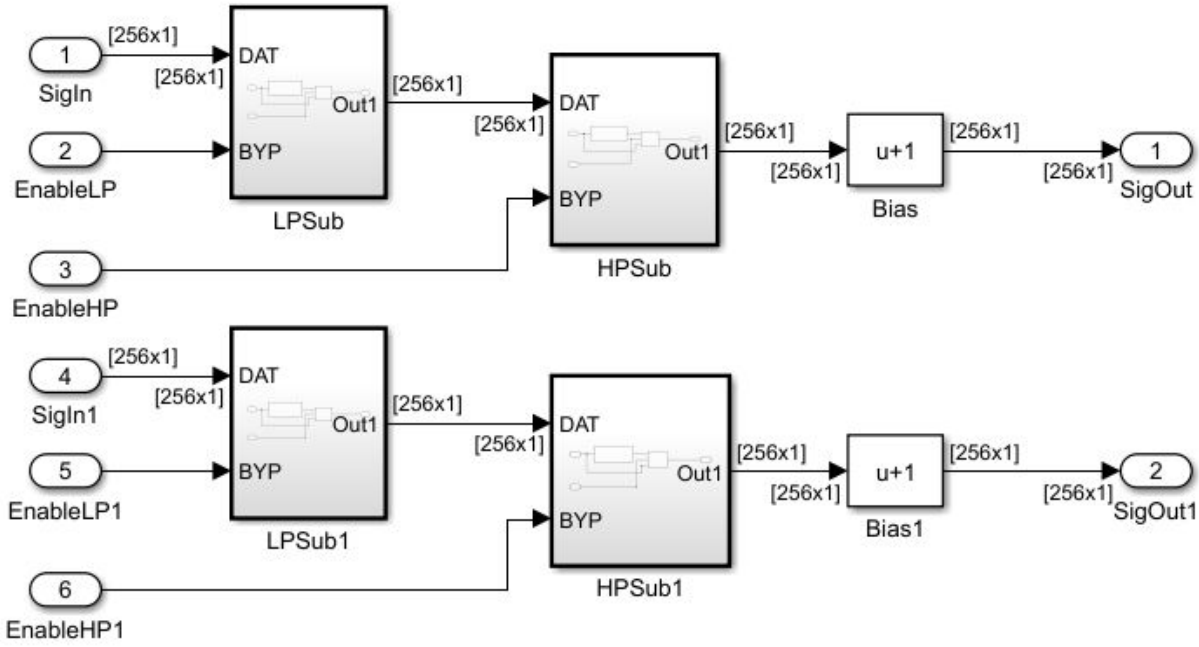
Performance

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.

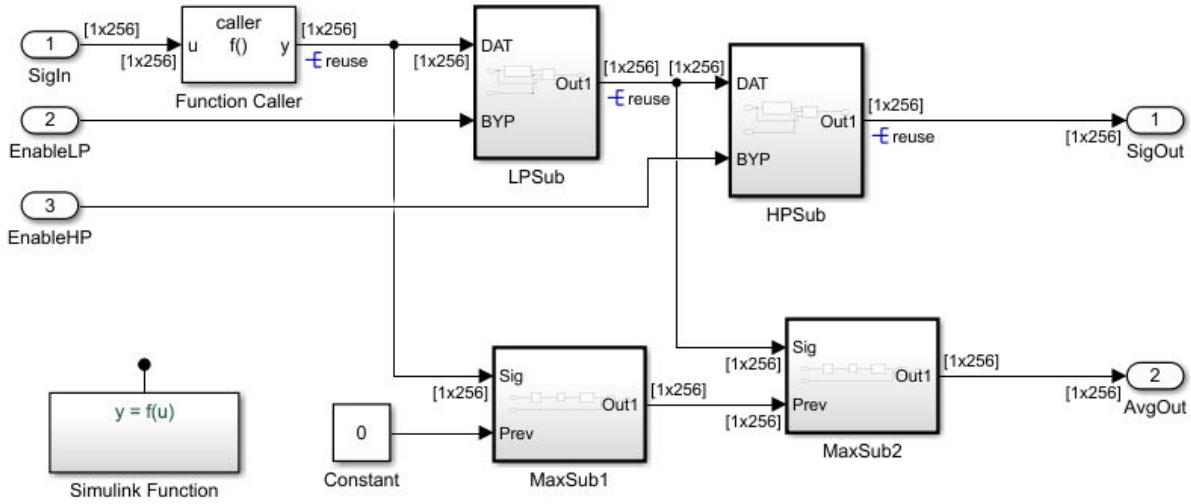
Buffer reuse for multiple signals in a path

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

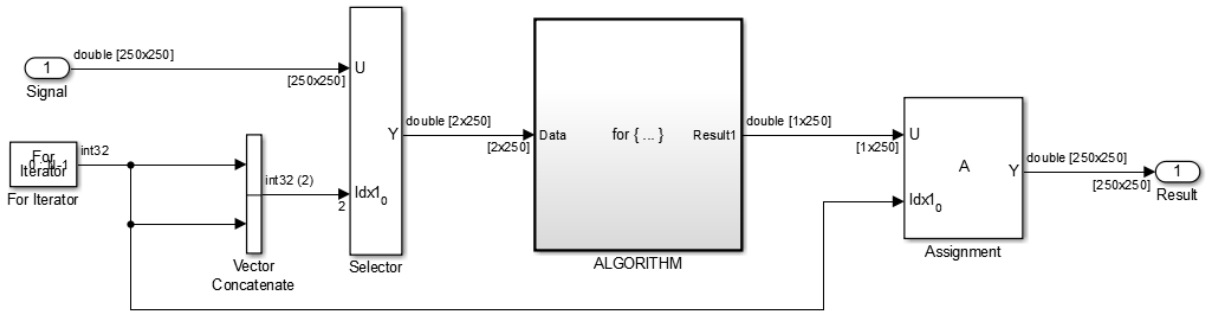
In R2016b, for iterative select assignment modeling patterns and matrix padding operations, there are buffer reductions in the generated code. This optimization reduces memory usage and increases code execution speed. Iterative select-assignment modeling patterns and matrix padding operations are useful in image processing.

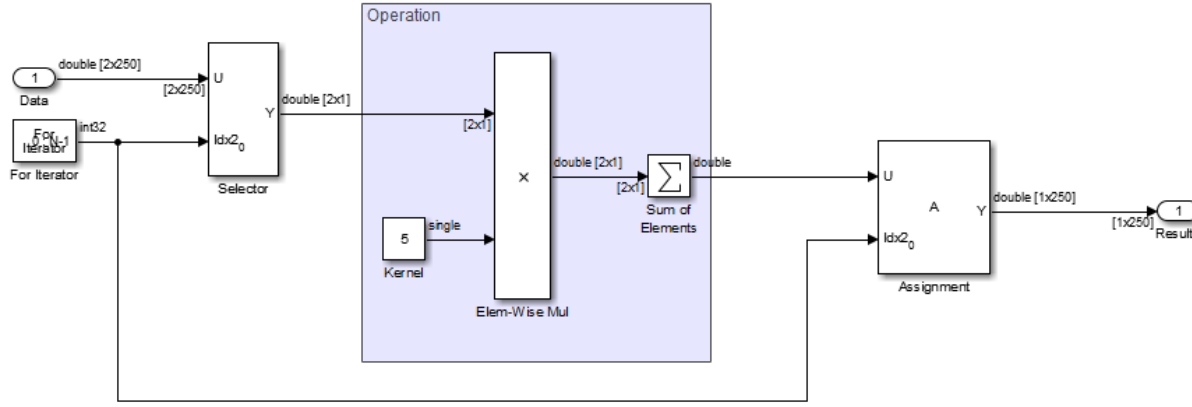
Data copy reduction for select-assign-iterator modeling pattern

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.

1
In1
double [250x250]
[250x250]
Signal
for { ... }
Result
double [250x250]
[250x250]
1
Out1
MyFilter


1
Signal
double [250x250]
[250x250]
U
Y
double [2x250]
[2x250]
Data
for { ... }
Result1
double [1x250]
[1x250]
U
A
Y
Idx1 0
double [250x250]
[250x250]
1
Result
For Iterator
int32
int32 (2)
2
Idx1 0
Vector Concatenate
Selector
ALGORITHM
Assignment


Operation
1
Data
double [2x250]
[2x250]
U
Y
double [2x1]
[2x1]
For Iterator
int32
Idx2 0
Selector
5
single
Kernel
×
double [2x1]
[2x1]
Elem-Wise Mul
Sum of Elements
double
U
A
Y
Idx2 0
double [1x250]
[1x250]
1
Result1
Assignment

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.

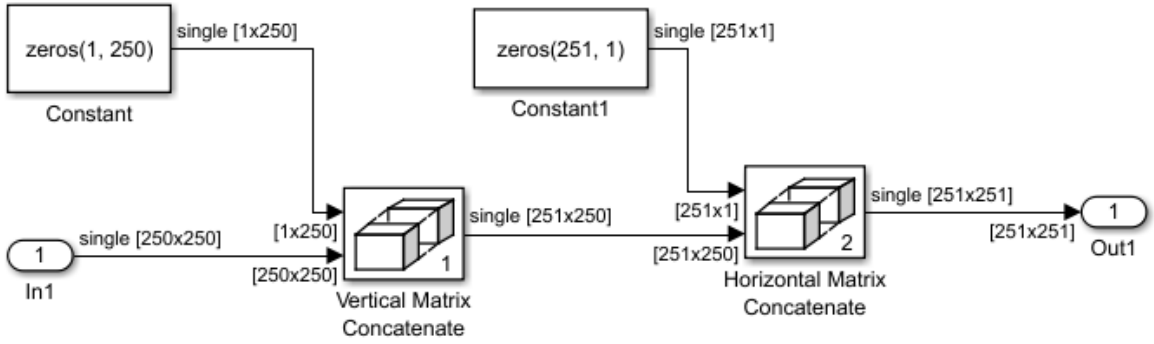
```
void rtwdemo_optimize_nestedloops_step(void)
{
  int32_T s1_iter;
  int32_T s2_iter;
  for (s1_iter = 0; s1_iter < 250; s1_iter++) {
    for (s2_iter = 0; s2_iter < 250; s2_iter++) {
      rtwdemo_optimize_nestedloops_Y.Out1[s1_iter + 250 * s2_iter] =
        rtwdemo_optimize_nestedloops_U.In1[250 * 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.

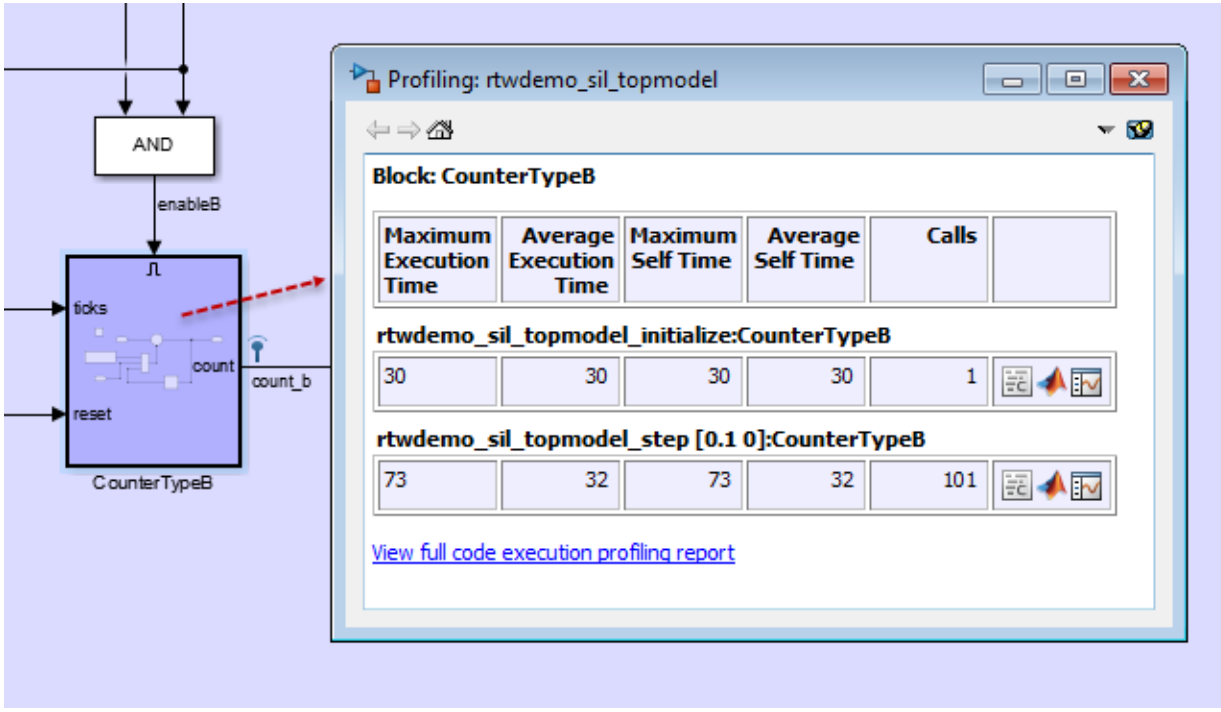
```
/* Model step function */
void pattern_grow_matrix_step(void)
{
  int32_T i;

  /* 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.

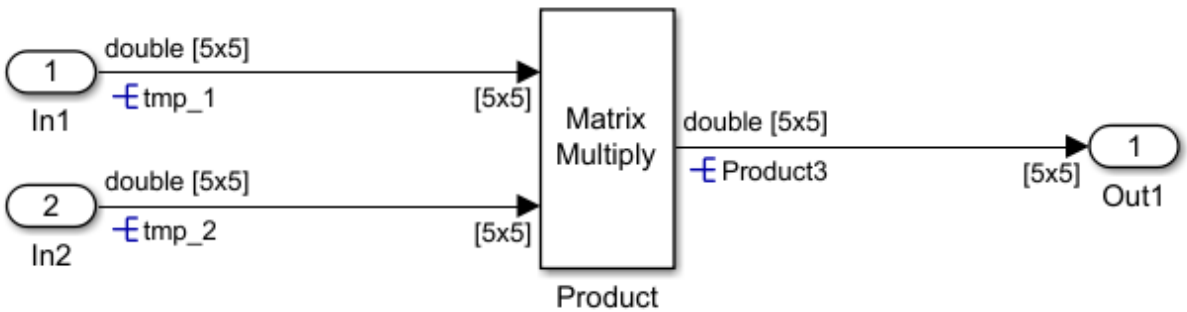
More efficient code for array element assignments

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 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.

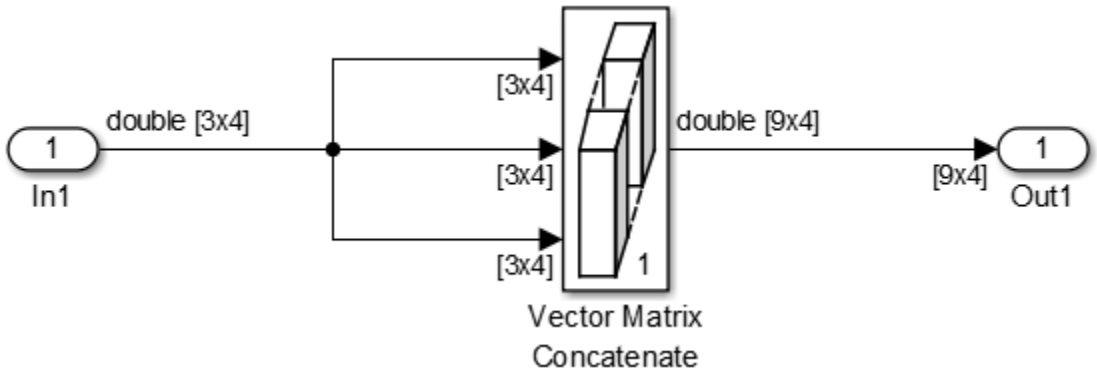
```
/* Product: '<Root>/Product' incorporates:
 *  Inport: '<Root>/In1'
 *  Inport: '<Root>/In2'
 */
for (i_0 = 0; i_0 < 5; i_0++) {
  for (i = 0; i < 5; i++) {
    matrix_multiply_B.Product3[i + 5 * i_0] = 0.0;
    for (i_1 = 0; i_1 < 5; i_1++) {
      matrix_multiply_B.Product3[i + 5 * i_0] += tmp_1[5 * 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.

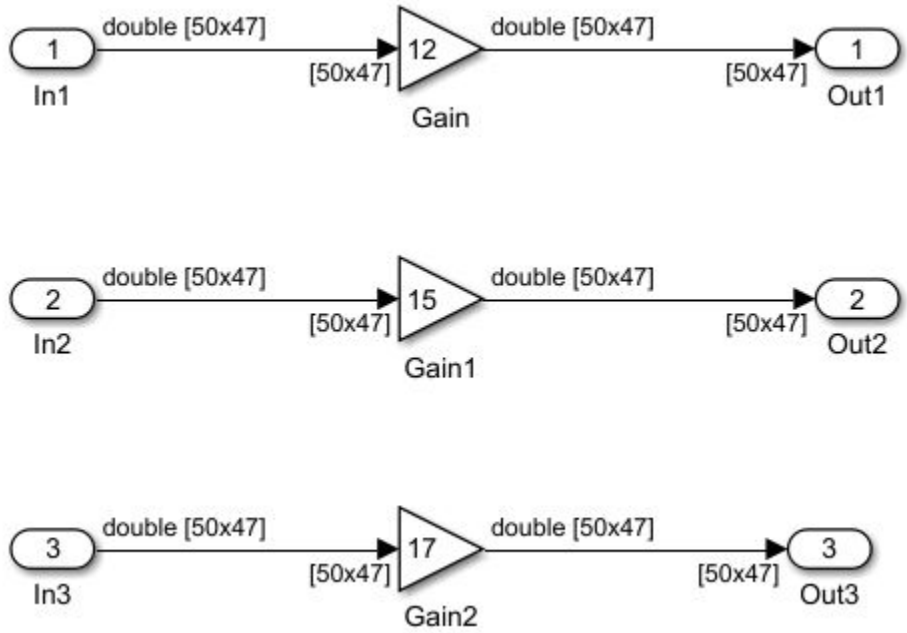
More efficient initialization code for root-level inports

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;
   }
 }
```

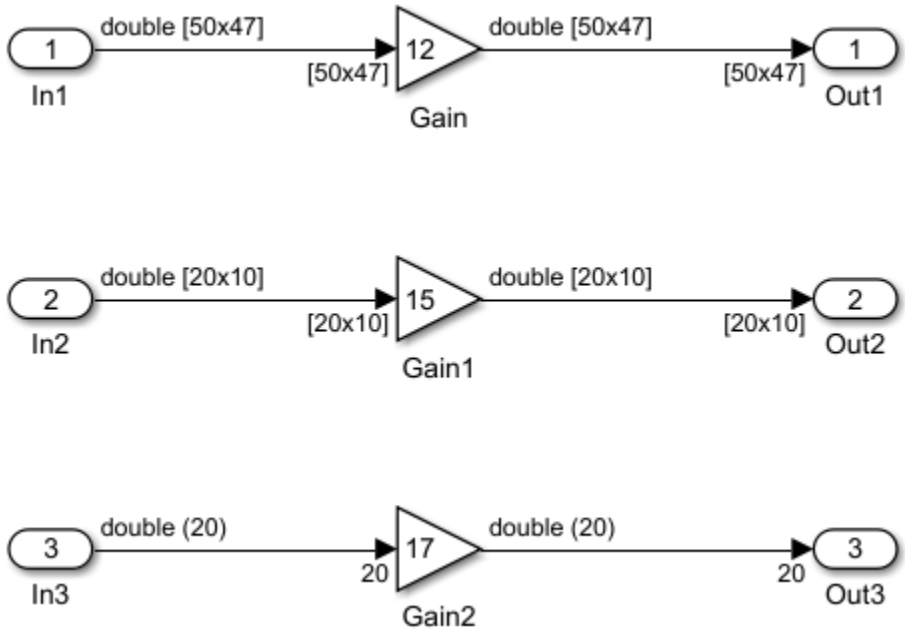
The generated code contains one `for` loop to initialize data for the three root-level inports.

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`.

AUTOSAR-typed per-instance memory import

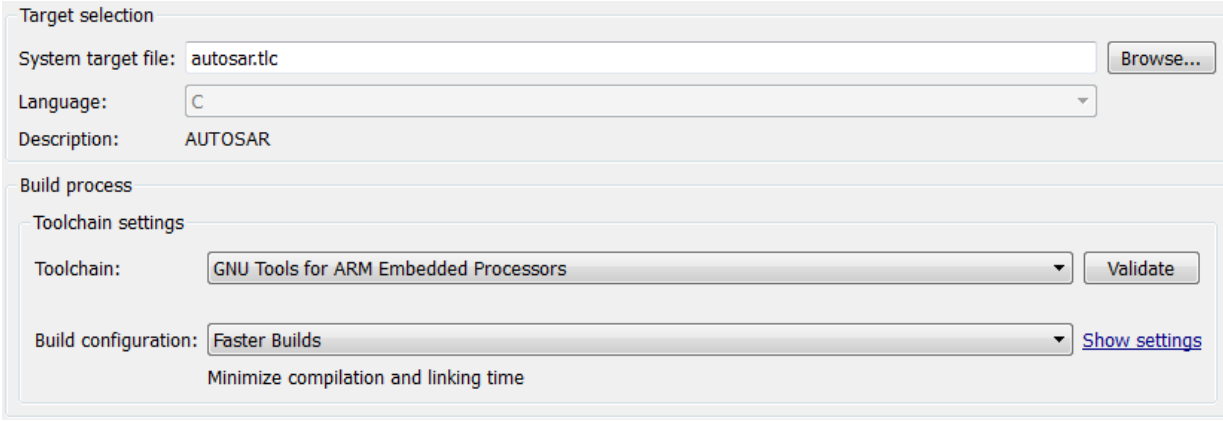
R2014a introduced modeling and code generation support for AUTOSAR-typed per-instance memory (`arTypedPerInstanceMemory`) in Simulink models. With R2015b, you can import `arTypedPerInstanceMemory` definitions from `arxml` code into a model. When you import an `arTypedPerInstanceMemory` definition, the `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.



Other targets that support toolchain controls include the ERT targets `ert.tlc` and `ert_shrlib.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)

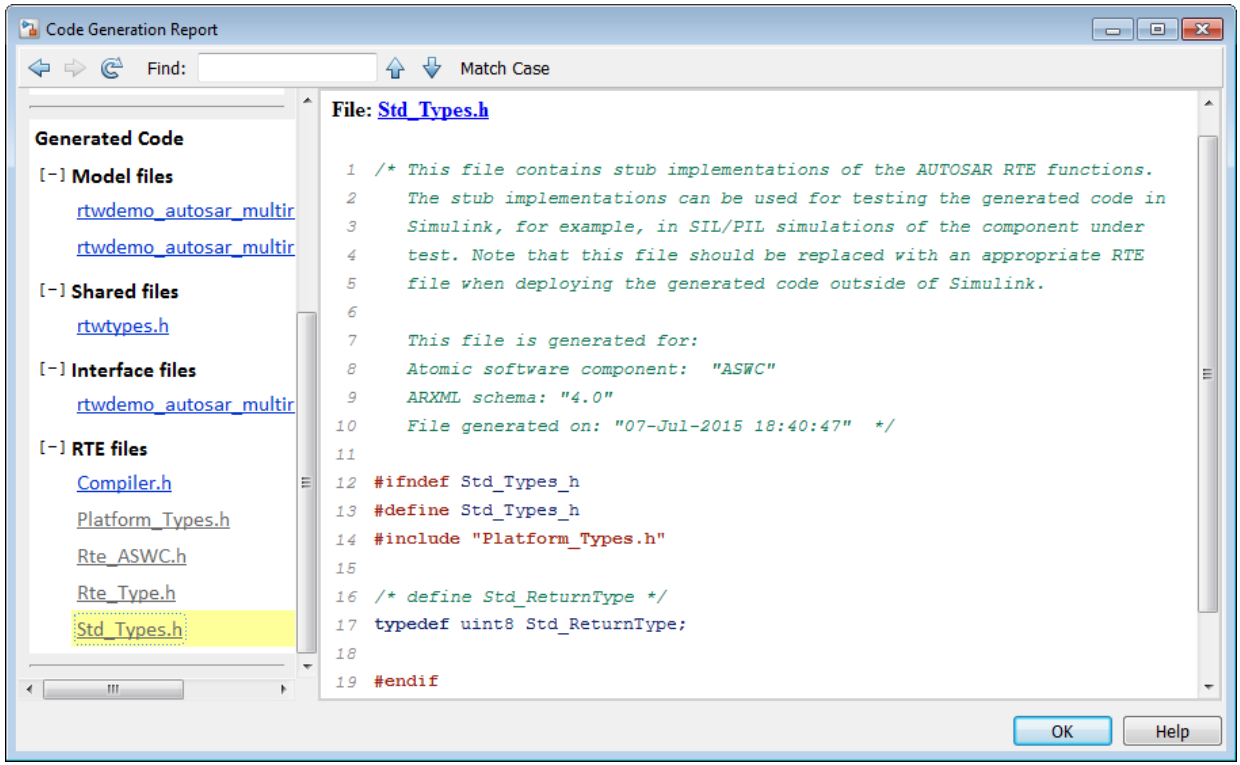
or processor-in-the-loop (PIL) simulations of the component under test. When the generated code ultimately is deployed in the AUTOSAR RTE, you replace the RTE stub files with externally-generated RTE files.

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

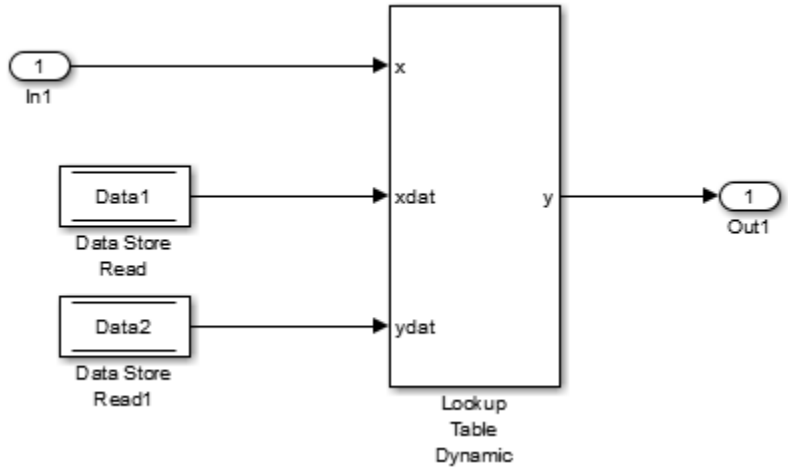
Previously, if a pair of model block I/O signals shared the same `Reusable` storage class specification, the code generator reused the root I/O signals in the generated code. In R2015b, this optimization extends to the input and output signals at a block or subsystem boundary if the input and output 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

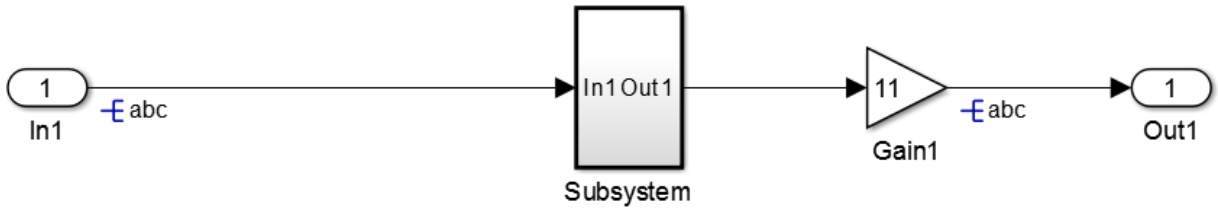
During a processor-in-the-loop (PIL) simulation, you can use the Simulation Data Inspector to view streamed task execution times. Previously, this data was available only at the end of the PIL simulation. For more information, see View and Compare Code Execution Times.

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];
  }
```

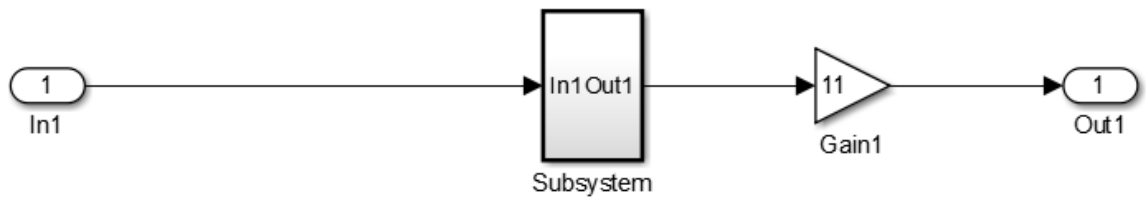
```
}

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.

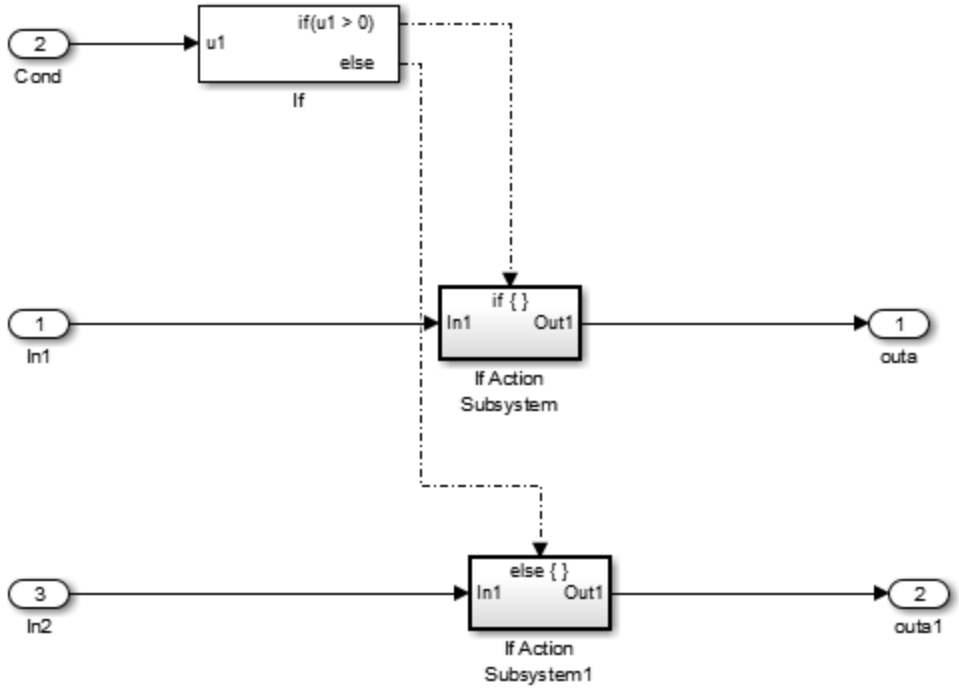
To configure model step function I/O arguments to allow buffer reuse, use either C function prototype control or C++ class interface control. For more information, see Combine Input and Output Arguments in Model Step Interface.

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;
  }
}
```

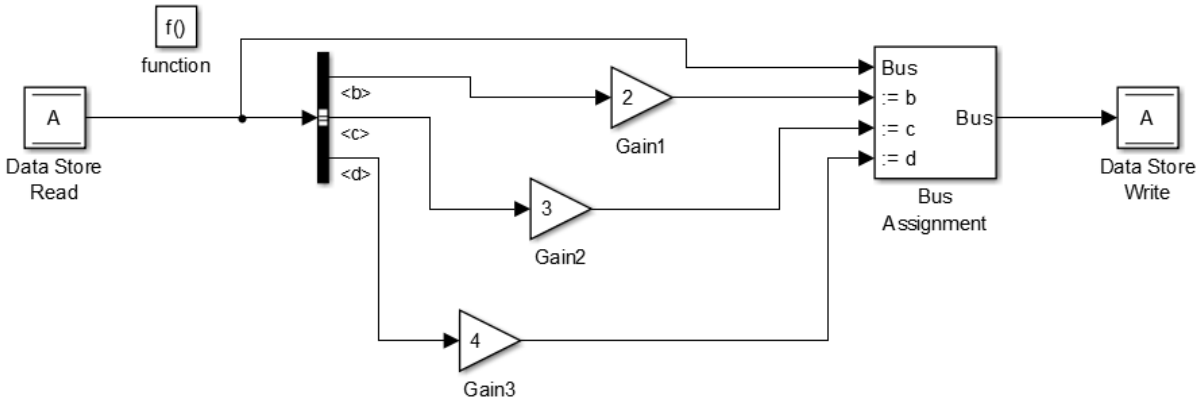
In R2015a, the code generator produced the reusable function named `auto_funcpack_IfActionSubsystem`, which is called twice in the generated code. In R2015b, because the subsystem consists of simple signal paths, the code generator inlines the code for each subsystem. For more information, see Generate 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:

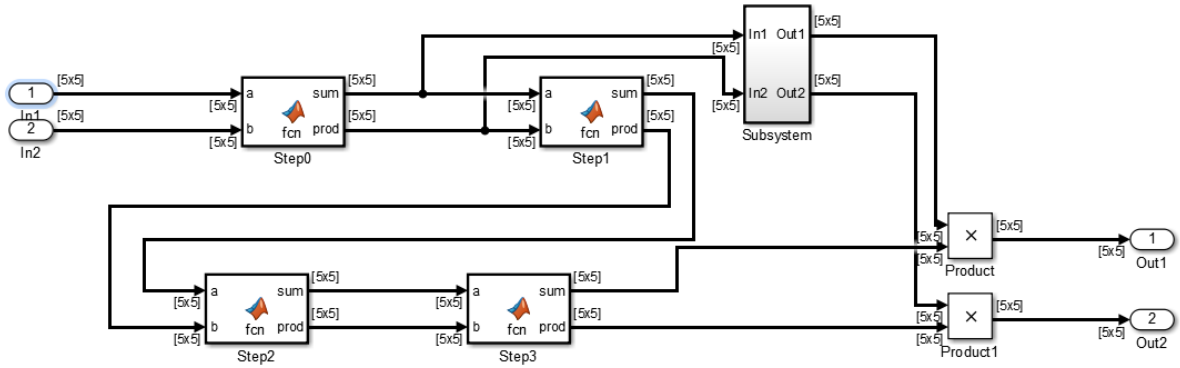
```
void f(void)
{
  int32_T i;
  for (i = 0; i < 150; i++) {
    rtDW.A.b[i] *= 2.0;
  }

  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:

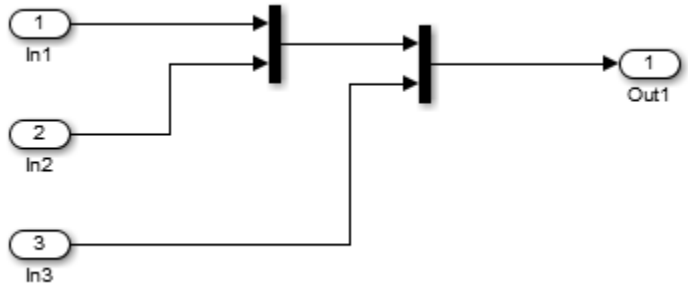
```
void local_reuse_step(void)
{
  real_T rtb_sum_g[25];
  real_T rtb_prod_o[25];
  real_T rtb_sum_a[25];
  real_T rtb_prod_h[25];
  int32_T i;
  local_reuse_Step0(local_reuse_U.In1, local_reuse_U.In2, rtb_sum_g, rtb_prod_o);
  local_reuse_Subsystem(rtb_sum_g, rtb_prod_o, local_reuse_B.Gain,
                        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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2015a

Version: 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 9-7
  - "Enhanced modeling with AUTOSAR system constants" on page 9-8
  - "AUTOSAR CompuMethod enhancements" on page 9-8
- Under Code Generation:
  - "Improved traceability for AUTOSAR RTE implicit read" on page 9-15
  - "Configurable aliveTimeout value for AUTOSAR ports" on page 9-16
  - "AUTOSAR calibration parameter export for COM_AXIS lookup tables" on page 9-17

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

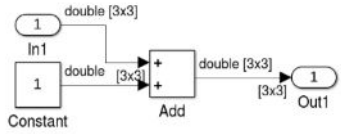
To improve MISRA-C compliance, you can run the following Model Advisor checks:

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 ConversionAtIn1Outport1' 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, `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 `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 **Outports** table.

Outports

Block Name	Code Identifier	Data Type	Scaling	Dimension
<Root>/Out1	*Defined externally*	uint32_T	ufix32_En14	1
<Root>/Out2	*Defined externally*	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.

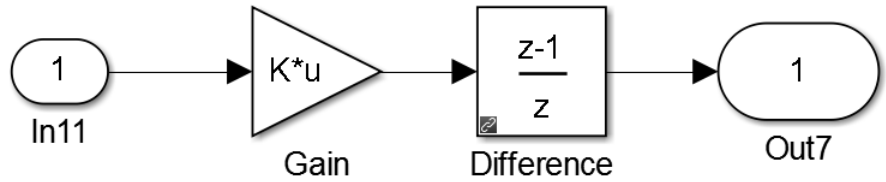
Compatibility Considerations

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.viewTfl` 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 9-13.

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.

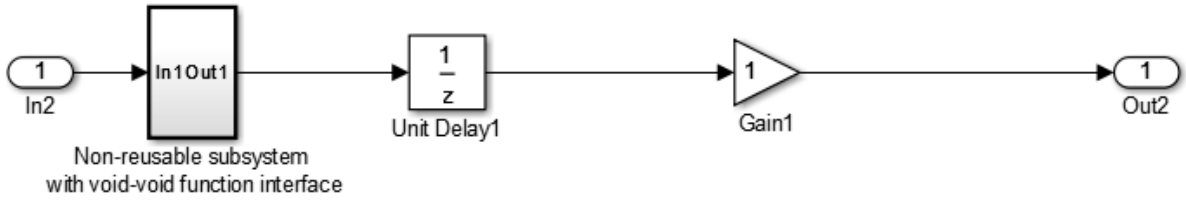
For more information on how to configure your model to take advantage of this optimization, see Buffer Reuse for Model Block Boundary and Unit Delay.

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();
}
```

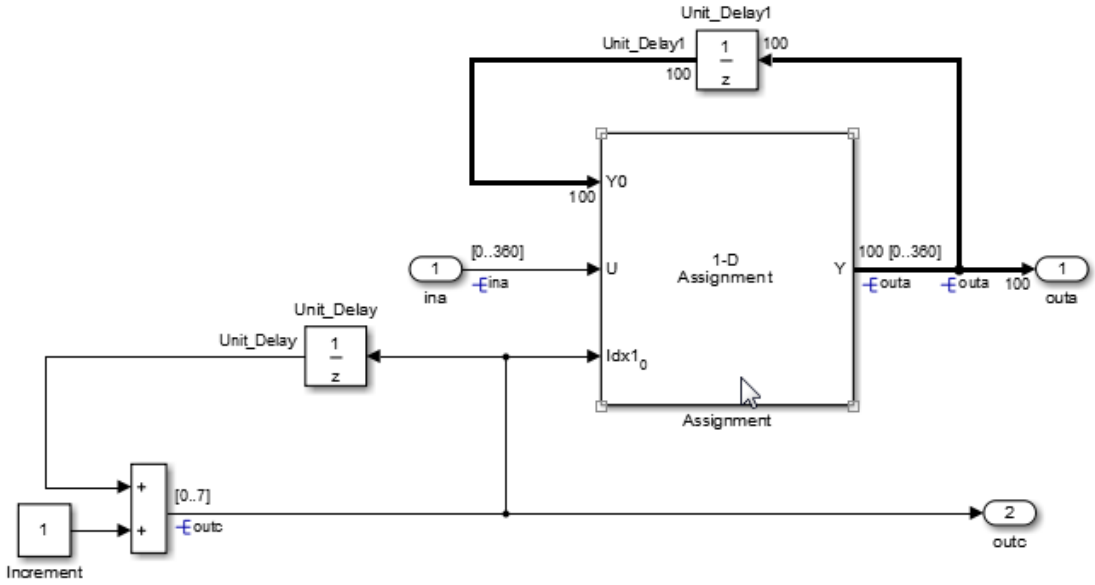
For more information on how to configure your model to use this optimization, see Buffer Reuse for Model Block Boundary and Unit Delay.

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:

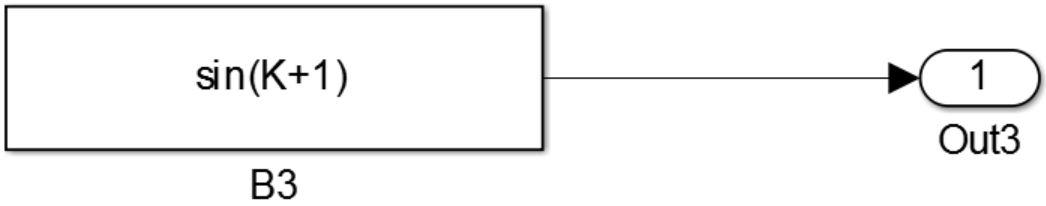
```
outc = (uint8_T)(outc + 1);

/* 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:

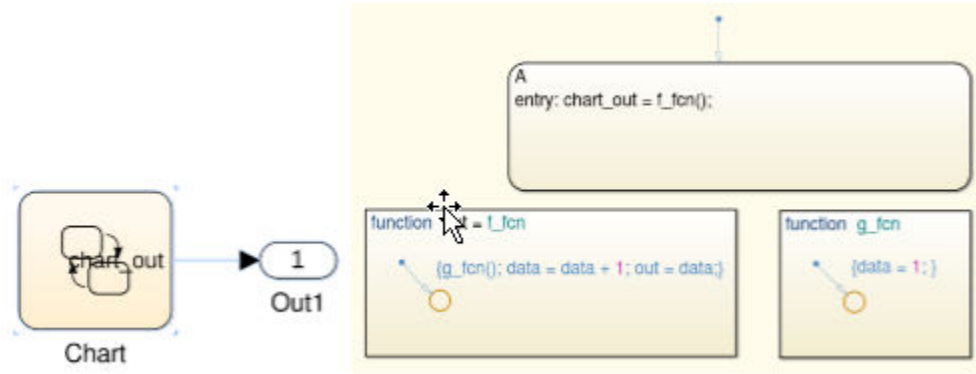
```
/* Outport: '<Root>/Out3' incorporates:
*  Constant: '<Root>/B3'
*/
mg1003222_Y.Out3 = sin(K + 1.0);
```

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;
}
```

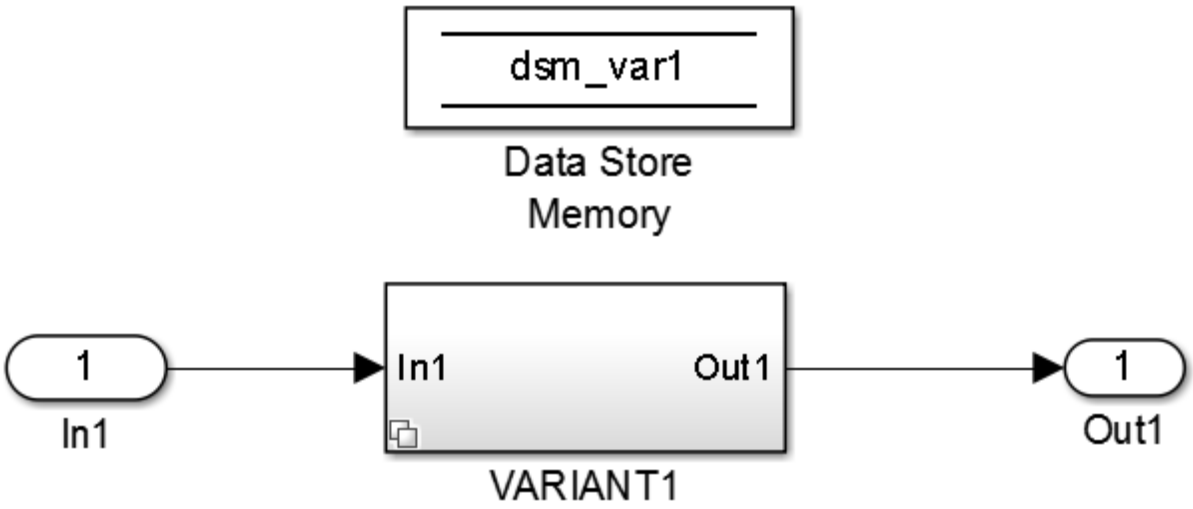
The code generator produces this code in R2015a:

```
/* 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

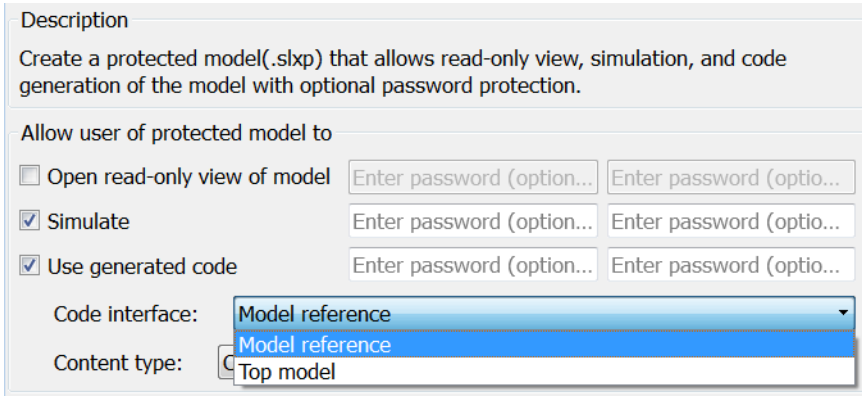
- “SIL/PIL for protected models” on page 9-31
- “SIL source code debugging using Microsoft Visual Studio Express” on page 9-32

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.



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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2014b

Version: 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

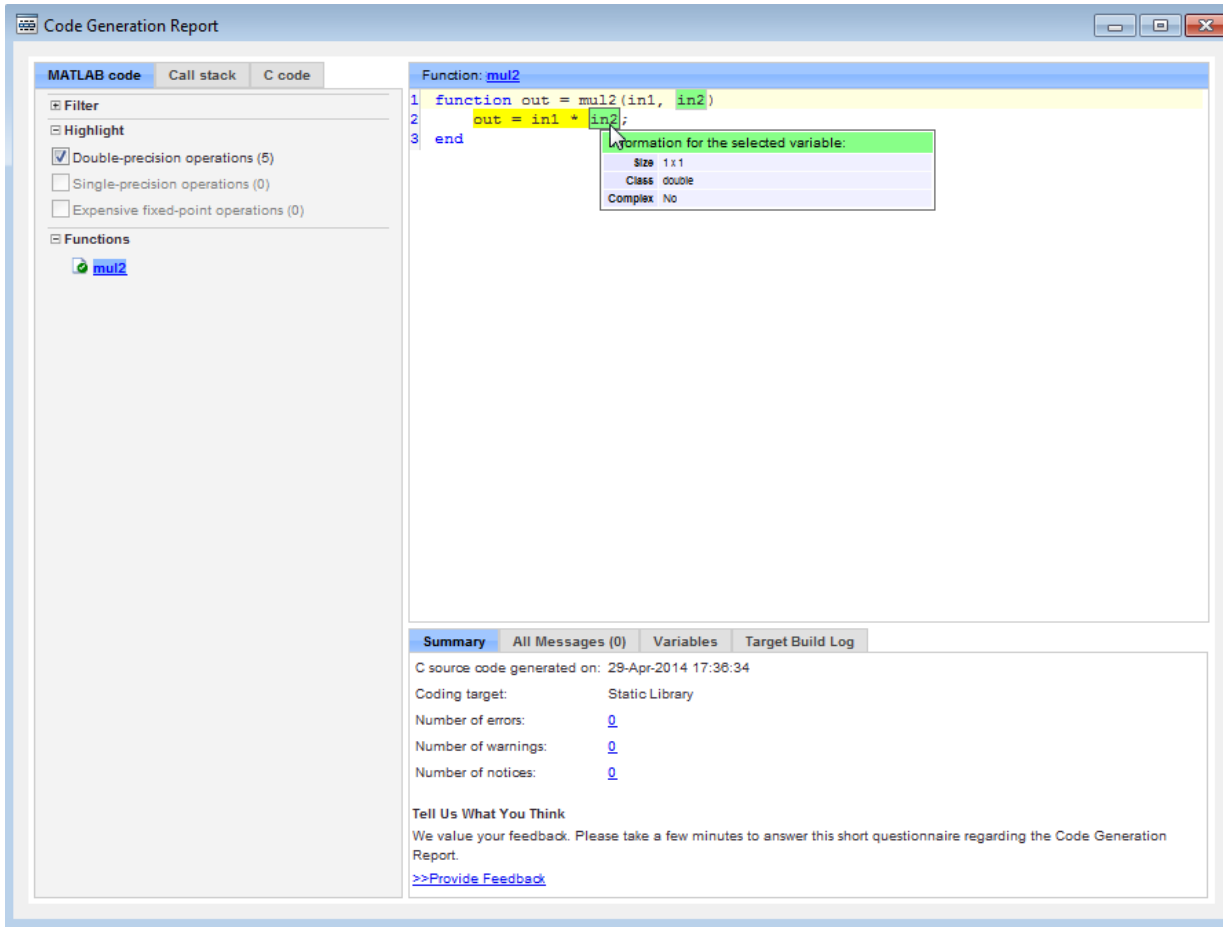
```
//!
//! multiple line comments
//! second line
//!
```

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);
   ```

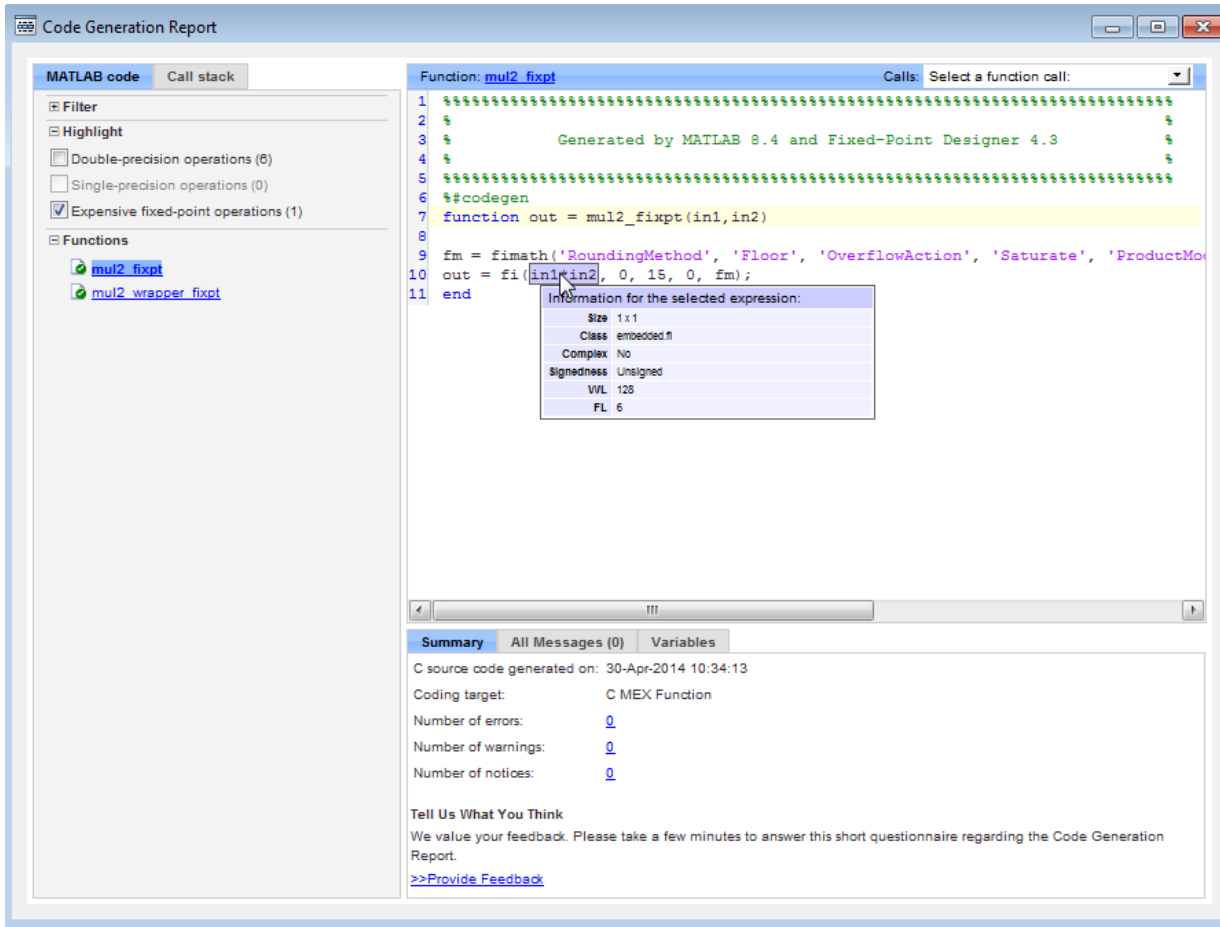
2. Set the `HighlightPotentialDataTypeIssues` property to `true`:

```
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.


Code Generation Report
MATLAB code
Call stack
Filter
Highlight
Double-precision operations (6)
Single-precision operations (0)
Expensive fixed-point operations (1)
Functions
mul2_fixpt
mul2_wrapper_fixpt
Function: mul2_fixpt
Calls: Select a function call:
%%%

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 10-29.)

For more information about AUTOSAR-related enhancements in R2014b, see:

- “Support for AUTOSAR Release 4.1” on page 10-16
- “AUTOSAR client and server modeling” on page 10-10
- “Multi-instance AUTOSAR atomic software components” on page 10-17
- Code Replacement for AUTOSAR
- “Support Package for AUTOSAR Standard” on page 10-14

- “AUTOSAR help navigation enhancements” on page 10-15

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.

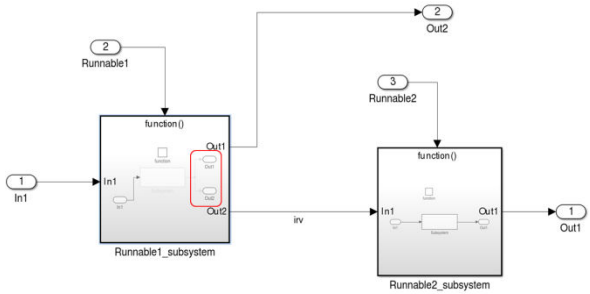
For more information, see Client-Server Interface and Configure AUTOSAR Client-Server Communication.

Global From and Goto blocks for AUTOSAR modeling

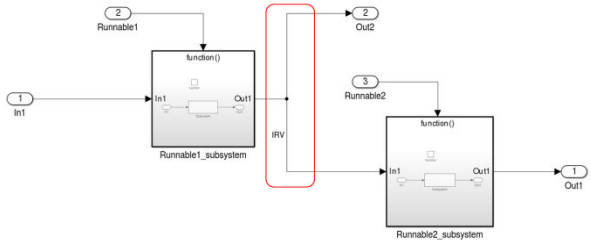
Beginning in R2014b, you can use global From and Goto blocks in a model configured for AUTOSAR. With From and Goto blocks, you can pass a signal from one block to another without actually connecting them. You can model AUTOSAR runnables with more flexibility and cleaner separation of components and interfaces.

AUTOSAR IRV branch from outport 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 outport, 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 outport outside a runnable boundary.
- When a runnable output signal branches to an IRV and a root outport 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 outport.
- 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 Outport block to inherit a constant sample time from its sources. The compiler then set the sample time of the root-level Outport 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.

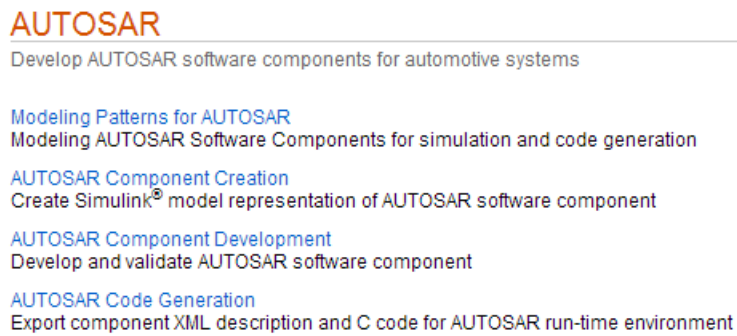
Compatibility Considerations

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.

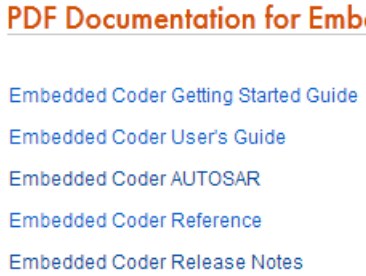
AUTOSAR help navigation enhancements

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.



- 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.



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 date 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.

DataTypeMappingSet 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 `DataTypeMappingSet` 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 `DataTypeMappingPackage` using the Configure AUTOSAR Interface dialog box or the AUTOSAR property `set` function. For example:

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 DataTypeMappingSet 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 (`ApplicationValueSpecification`s).

Beginning in R2014b, for AUTOSAR data initialization with `ApplicationValueSpecification`, Embedded Coder provides the following support:

- The `arxml` importer uses `ApplicationValueSpecification`s 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 `arxml` code that uses `ApplicationValueSpecification`s 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 (`CompuMethod`s) 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 `CompuMethod`s, but did not allow users control over `CompuMethod` attributes, including the direction of `CompuMethod` conversion between internal and physical representations of a value. For `CompuMethod`s 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 `CompuMethod`s. 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 (`CompuMethod`s) that you imported into Simulink, but for designs originated in Simulink, generated `CompuMethod`s only for fixed point application types.

Beginning in R2014b, the exporter generates `CompuMethod`s 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 `CompuMethod`s did not contain unit references. Beginning in R2014b:

- The `arxml` importer preserves unit and physical dimension information found in imported `CompuMethod`s. 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 `CompuMethod`s 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 10-18.

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_OBJS` 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_OBJS` is defined in your TMF. For example, you must replace:

```
SHARED_OBJS = $(SHARED_SRC:.c=.obj)
```

with:

```
SHARED_OBJS = $(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 10-28
- “Replacement of shift and cast operations involving vector and matrix operands” on page 10-28

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

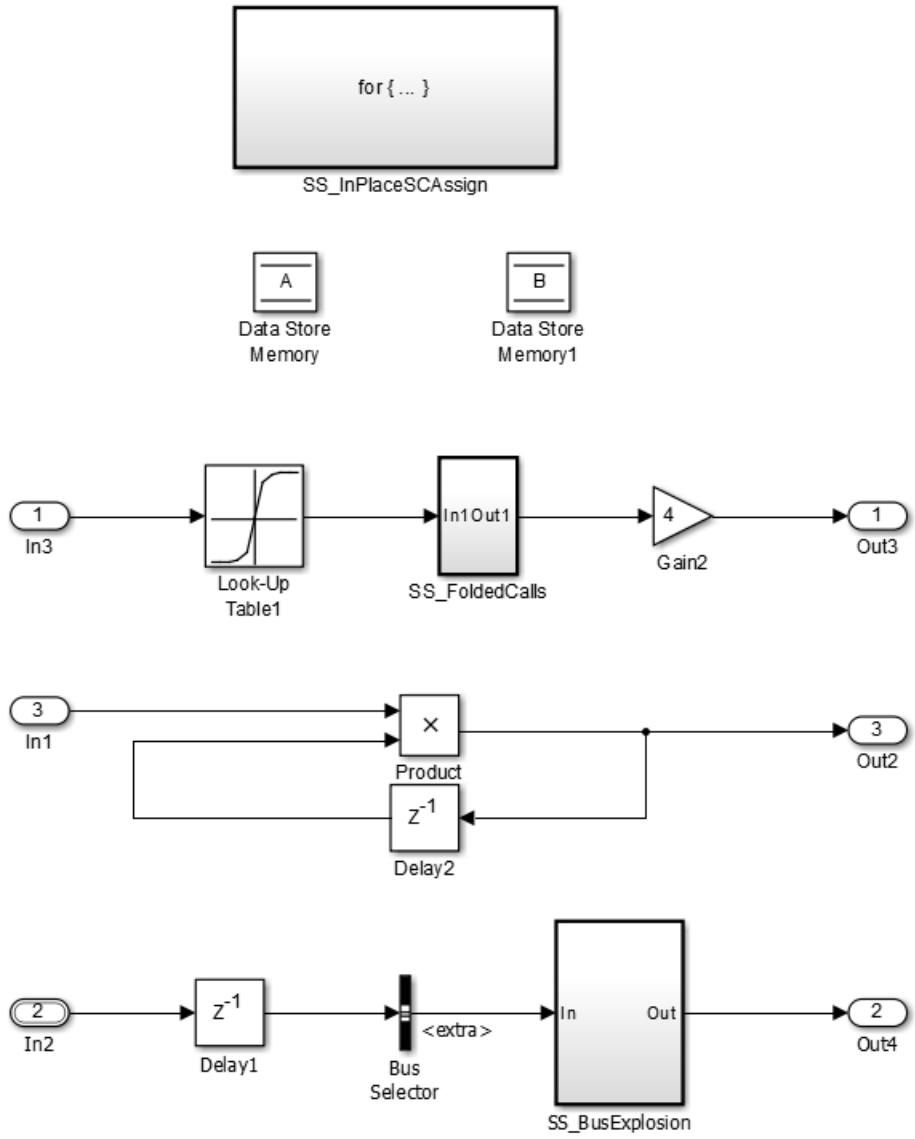
Reduced RAM and faster execution for modeling patterns including select-assign-iterate blocks, subsystem interfaces, and model references

- “Example Model” on page 10-33
- “In-place assignments for select-assign-iterate pattern” on page 10-35
- “Subsystem signal information” on page 10-36
- “Variable reuse around call site” on page 10-37

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

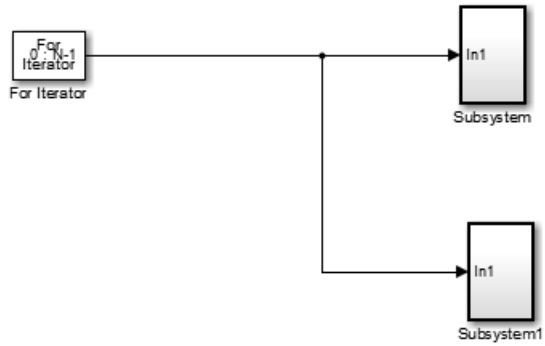
Consider the model `example_subsys1`, that contains the subsystem and models used for the examples for each optimization:


for { ... }
SS_InPlaceSCAssign
A
Data Store Memory
B
Data Store Memory1
1
In3
Look-Up Table1
In1 Out1
SS_FoldedCalls
4
Gain2
1
Out3
3
In1
Product
3
Out2
Z-1
Delay2
2
In2
Z-1
Delay1
<extra>
Bus Selector
In
Out
SS_BusExplosion
2
Out4

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:

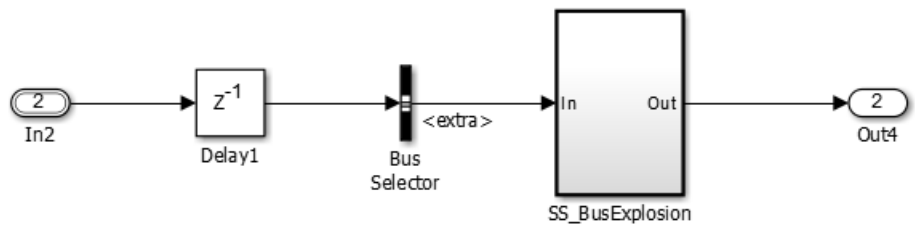
```
/* Output and update for atomic system: '<S3>/Subsystem1' */
void example_subsys1_Subsystem1(int32_T rtu_In1)
{
  /* Assignment: '<S5>/Assignment' */
  if (example_subsys1_DWork.ForIterator_IterationMarker < 2) {
    example_subsys1_DWork.ForIterator_IterationMarker = 2U;
  }

  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:

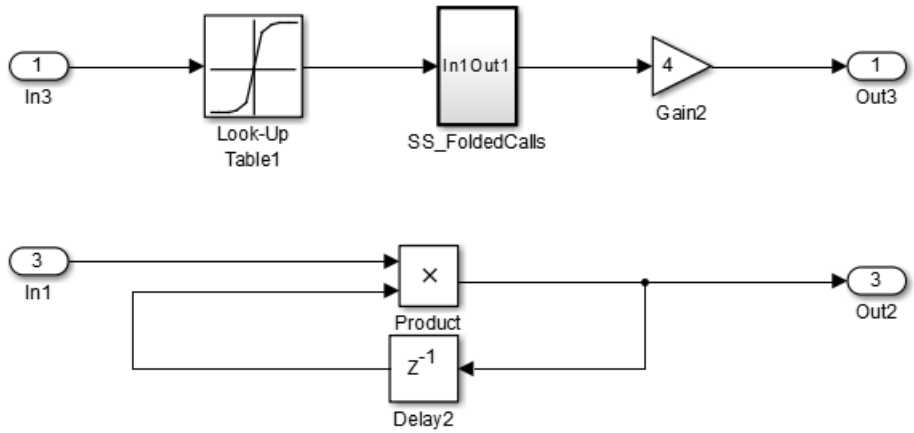
```
/* Delay: '<Root>/Delay1' */
  example_subsys1_SS_BusExplosion(example_subsys1_DWork.Delay1_DSTATE.extra);

  /* 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:

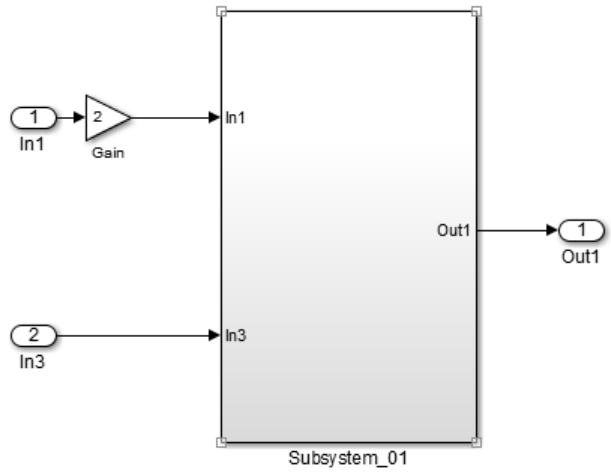
```
  /* Product: '<Root>/Product' incorporates:
   *  Delay: '<Root>/Delay2'
   *  Inport: '<Root>/In1'
   */
  for (rtb_DataTypeConversion = 0; rtb_DataTypeConversion < 12;
       rtb_DataTypeConversion++) {
    example_subsys1_Y.Out2[rtb_DataTypeConversion] *=
      example_subsys1_U.In1[rtb_DataTypeConversion];
  }

 /* 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];
```

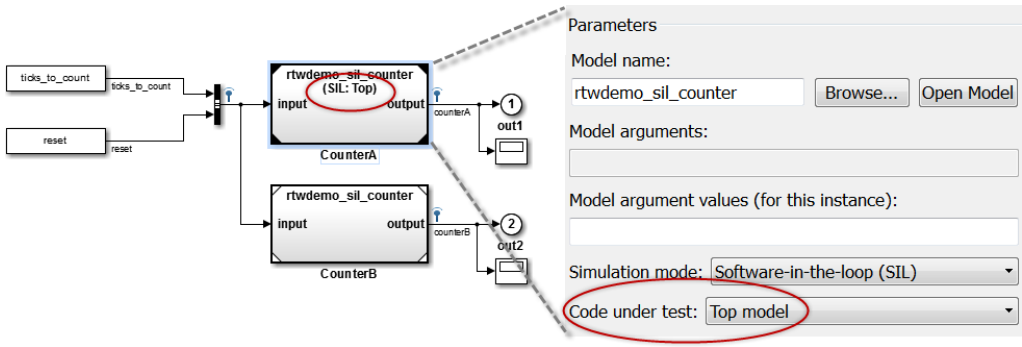
```
  Gain[2] = 2.0 * exlocal_U.In1[2];
  Gain[3] = 2.0 * exlocal_U.In1[3];

    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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2014a

Version: 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 `//`.
`Multi-line (Use C-style comments)`	Generate single or multi-line comments delimited by `/*` and `*/`.

- 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
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.

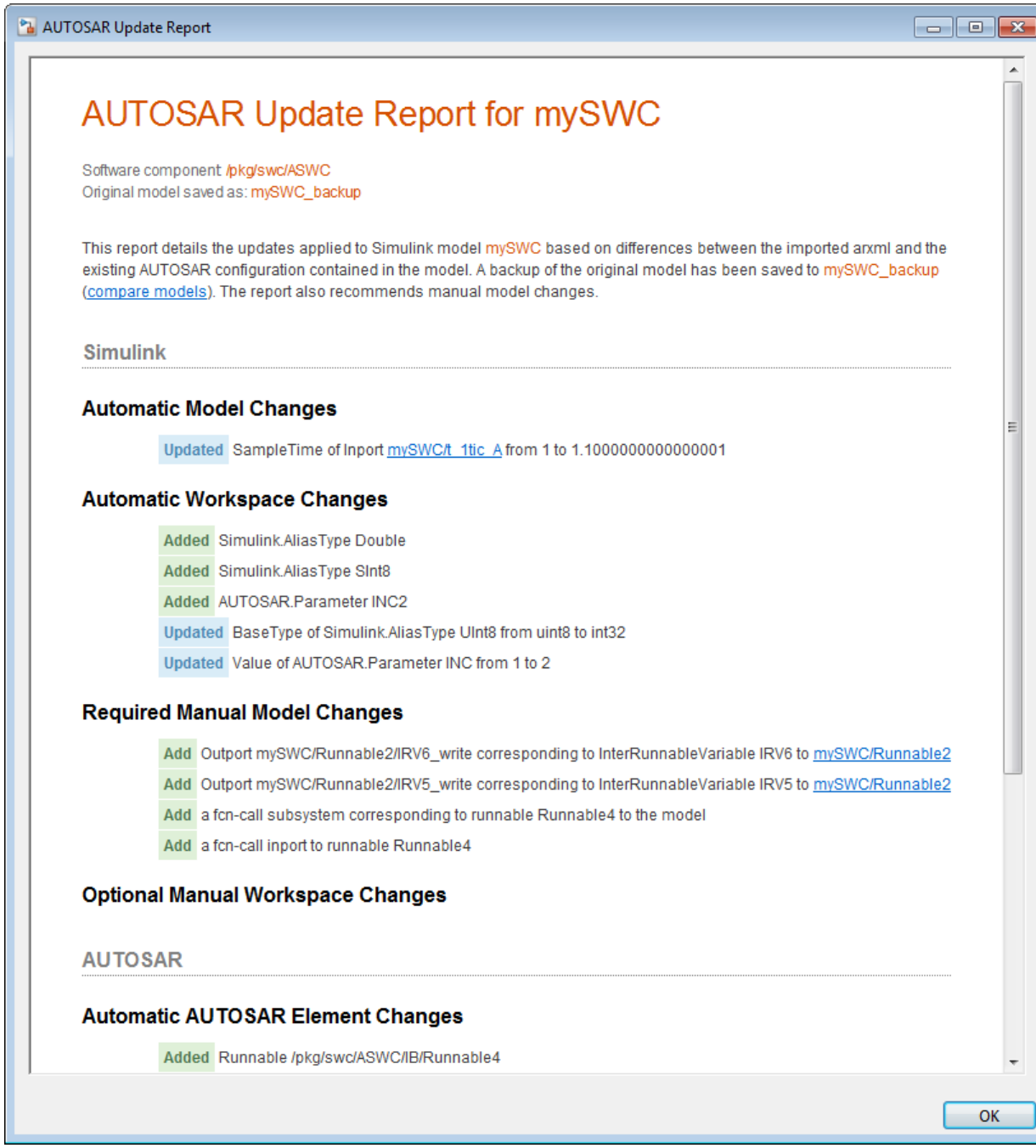
Model Architecture and Design

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` method `updateModel`, 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 Update Report
AUTOSAR Update Report for mySWC
Software component /pkg/swc/ASWC
Original model saved as: mySWC_backup
This report details the updates applied to Simulink model mySWC based on differences between the imported arxml and the existing AUTOSAR configuration contained in the model. A backup of the original model has been saved to mySWC_backup (compare models). The report also recommends manual model changes.
Simulink
Automatic Model Changes
Updated SampleTime of Inport mySWC/t_1tic_A from 1 to 1.1000000000000001
Automatic Workspace Changes
Added Simulink.AliasType Double
Added Simulink.AliasType SInt8
Added AUTOSAR.Parameter INC2
Updated BaseType of Simulink.AliasType UInt8 from uint8 to int32
Updated Value of AUTOSAR.Parameter INC from 1 to 2
Required Manual Model Changes
Add Outport mySWC/Runnable2/IRV6_write corresponding to InterRunnableVariable IRV6 to mySWC/Runnable2
Add Outport mySWC/Runnable2/IRV5_write corresponding to InterRunnableVariable IRV5 to mySWC/Runnable2
Add a fcn-call subsystem corresponding to runnable Runnable4 to the model
Add a fcn-call inport to runnable Runnable4
Optional Manual Workspace Changes
AUTOSAR
Automatic AUTOSAR Element Changes
Added Runnable /pkg/swc/ASWC/IB/Runnable4
OK

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 11.1. rtwdemo_autosar_multirunnables.arxml

```
<RUNNABLE-ENTITY UUID="65432c3e-34c7-5e82-4229-f6d04927eb78">
    <SHORT-NAME>Runnable1</SHORT-NAME>
...
    <SYMBOL>test_symbol</SYMBOL>
...
</RUNNABLE-ENTITY>
```

Example 11.2. 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 Outport blocks

In R2014a, the sample time of root-level Outport 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 Outport block has a constant sample time.

- For models using the AUTOSAR target, the compiler reports a warning if a root-level Outport block is configured to inherit a constant sample time from its sources. The compiler then sets the sample time of the root-level Outport 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 Outport blocks to the fundamental rate of the model.
- The Upgrade Advisor adds a check identifying root-level Outport 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 Outport 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.

`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.

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

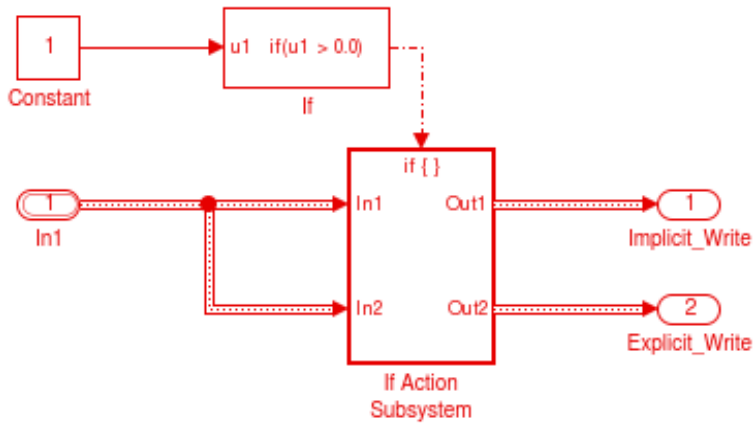
Previously, when specifying a namespace for a model class, the generated namespace code might be incorrectly indented if you selected K&R for the **Indent style** on the **Code Generation > Code Style** pane. In R2014a, the generated namespace code follows coding standards when you select the K&R style.

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_PPorts_eOut2(
    Rte_IRead_Runnable_Step_RPorts_iIn1());
  }
  /* Outport: '<Root>/Implicit_Write' */
  Rte_IWrite_Runnable_Step_PPorts_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 (targetcommlib)

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	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. For example, if your host computer is a 64-bit Linux machine, you cannot specify the following combination of settings: • **Device vendor**: `Generic` • **Device type**: `32-bit x86 compatible` To resolve the mismatch errors, do one of the following: • 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.
	The software generates an error if: • 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	The `GenerateErtSFunction` parameter has the following command-line behavior: • `set_param(`*`model`*`, 'GenerateErtSFunction', 'on')` generates a warning that the parameter will be removed in a future release. As a replacement, use the command `set_param(`*`model`*`, 'CreateSILPILBlock', 'SIL')`. • `set_param(`*`model`*`, 'GenerateErtSFunction', 'off')` does not change the parameter. As a replacement, use the command `set_param(`*`model`*`, 'CreateSILPILBlock', 'None')`. • `get_param(`*`model`*`, 'GenerateErtSFunction')` returns the value `off`. As a replacement, use the command `get_param(`*`model`*`, 'CreateSILPILBlock')`.
Parameter tuning	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
	The software does not support SIL block creation if all of the following apply: • **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	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.
	The software does not support automatic data definition and data transfer for local data stores in the SIL block.

Feature	R2014a Details
C++ class code (previously called C++ encapsulated code)	For C++ class code: • 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`. You can specify these settings through the **Code Generation** > **Interface** pane.
Code generation report	The code generation report does not display test harness files for your SIL block.
Multiword fixed-point data	At the SIL block interface, the software does not support multiword fixed-point data types. The software supports: • At the block interface, single word data types that are wider than 32 bits. • Within the SIL block, multiword fixed-point data types.
Boolean data type replacement	At the SIL block boundary, the software does not support the replacement of the `boolean` 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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2013b

Version: 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 `mxArray`) 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, `'myemx$M$N'`.
- 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 12-12.

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`.

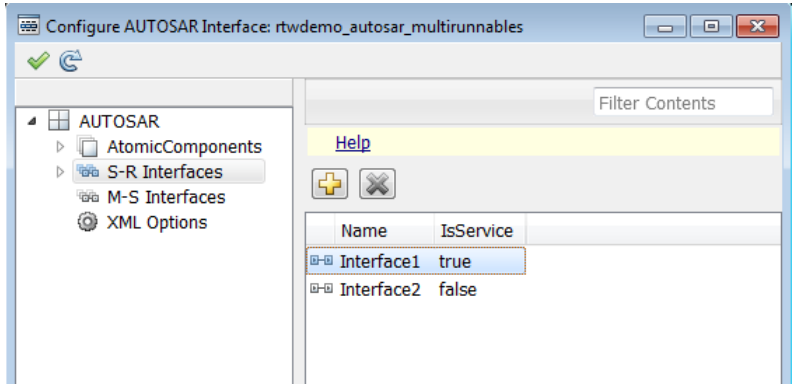
Programmatic interface for configuring AUTOSAR properties and Simulink-AUTOSAR mapping

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	`multiword_types.h`
Model reference target	`model_reference_types.h`
Model reference blocks	`model_reference_types.h`
MAT-file logging is selected	`builtin_typeid_types.h` `multiword_types.h`
C API	`builtin_typeid_types.h`
Interface is set to `External mode`	`multiword_types.h`

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 12-4.

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 outport, 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 (tiiqmathlib) 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.

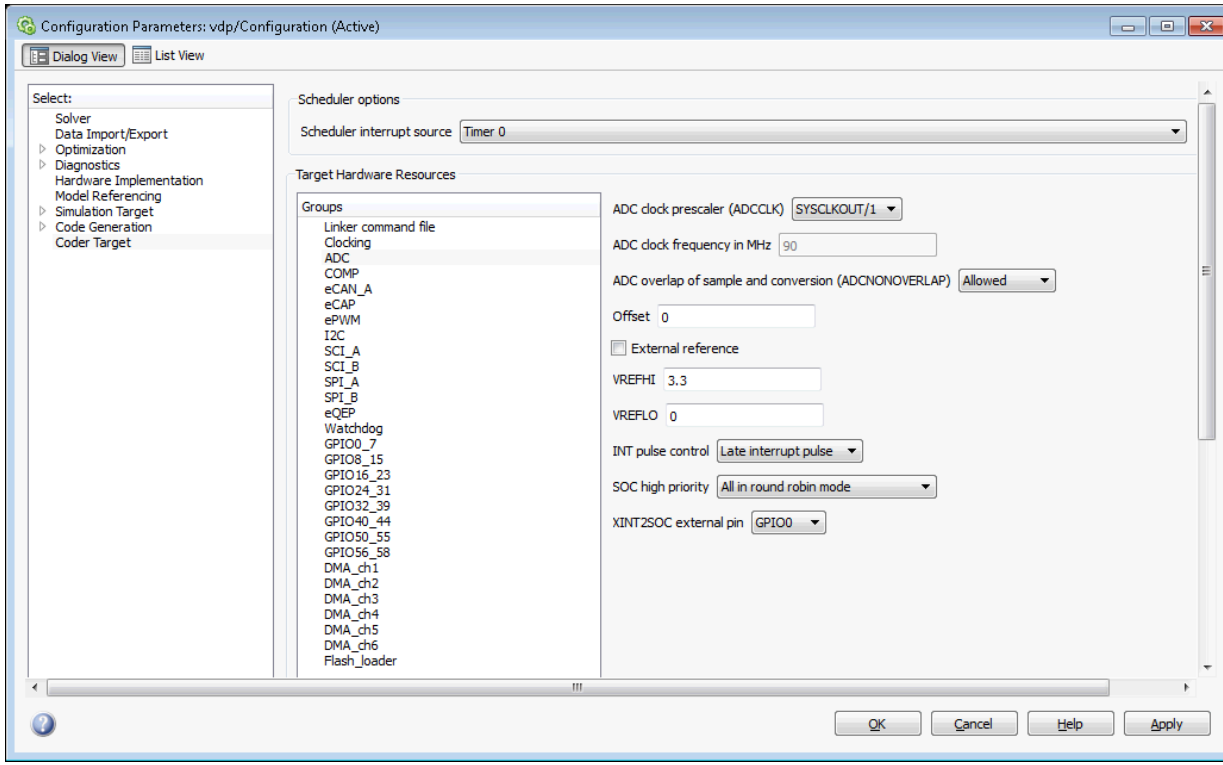
Compatibility Considerations

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 a 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 loccur.

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.
```

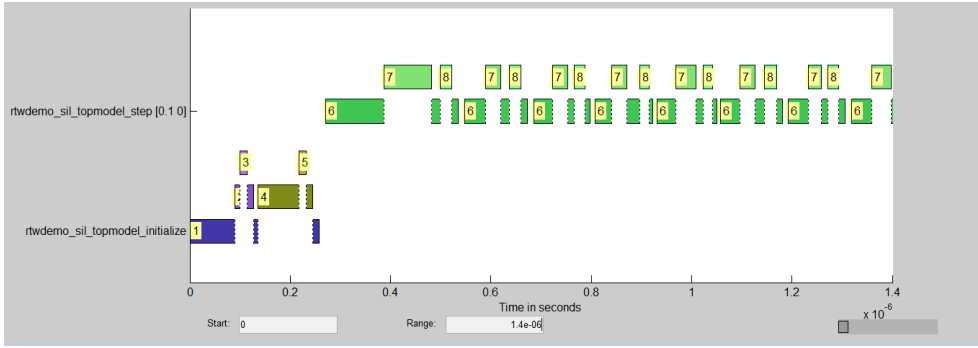
Previously, if function profiling was enabled for a SIL or PIL simulation, building your model produced an error message. For example:

```
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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2013a

Version: 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

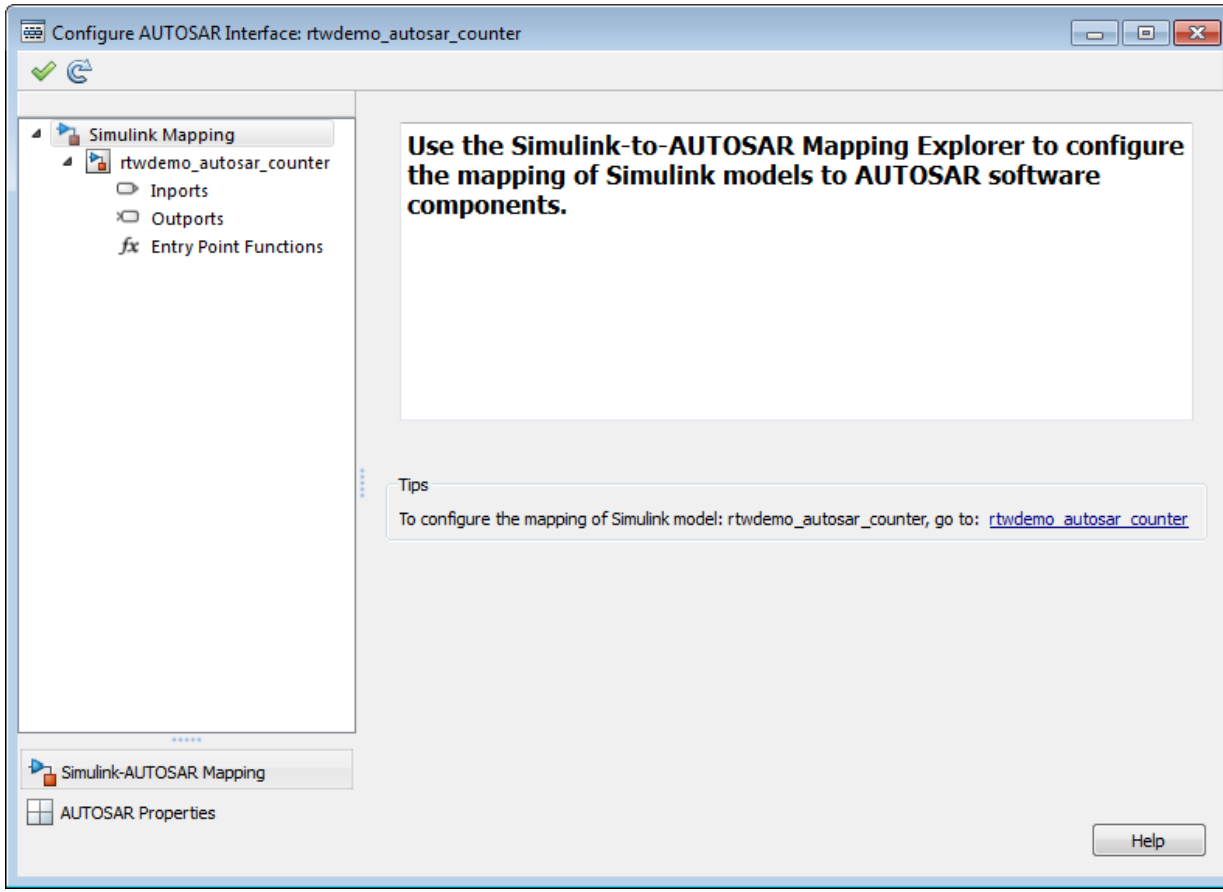
AUTOSAR user interface and round trip ARXML file import and export improvements

Improved graphical user interfaces for AUTOSAR configuration

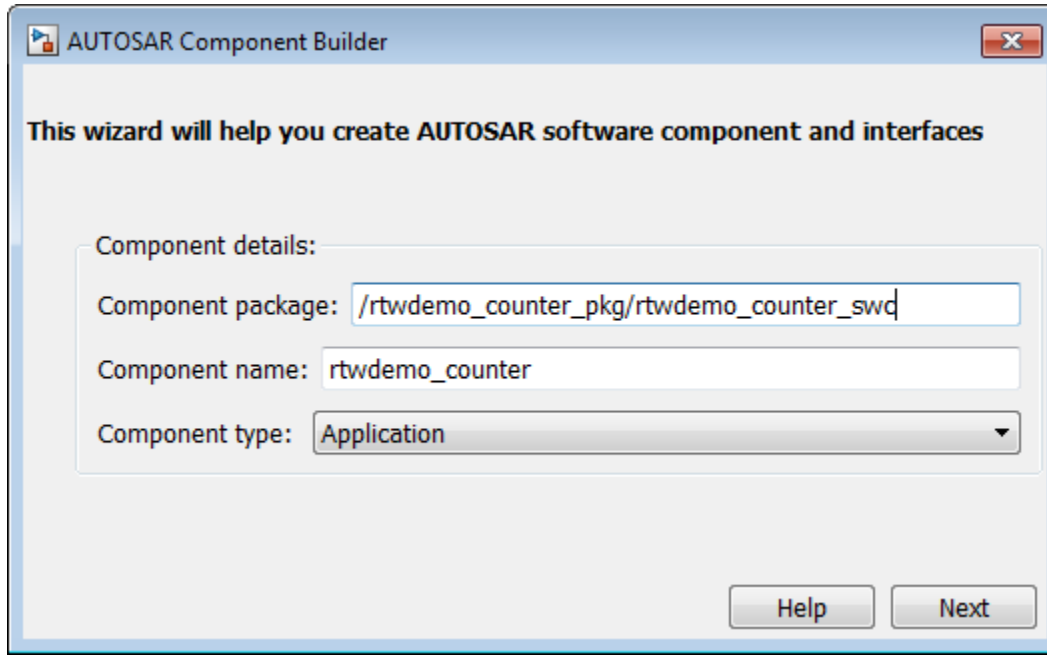
Embedded Coder software provides graphical user interfaces that allow you to add AUTOSAR elements to a Simulink model and map model components and interfaces to AUTOSAR components and interfaces. R2013a provides several improvements to the graphical user interfaces for AUTOSAR configuration:

- 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

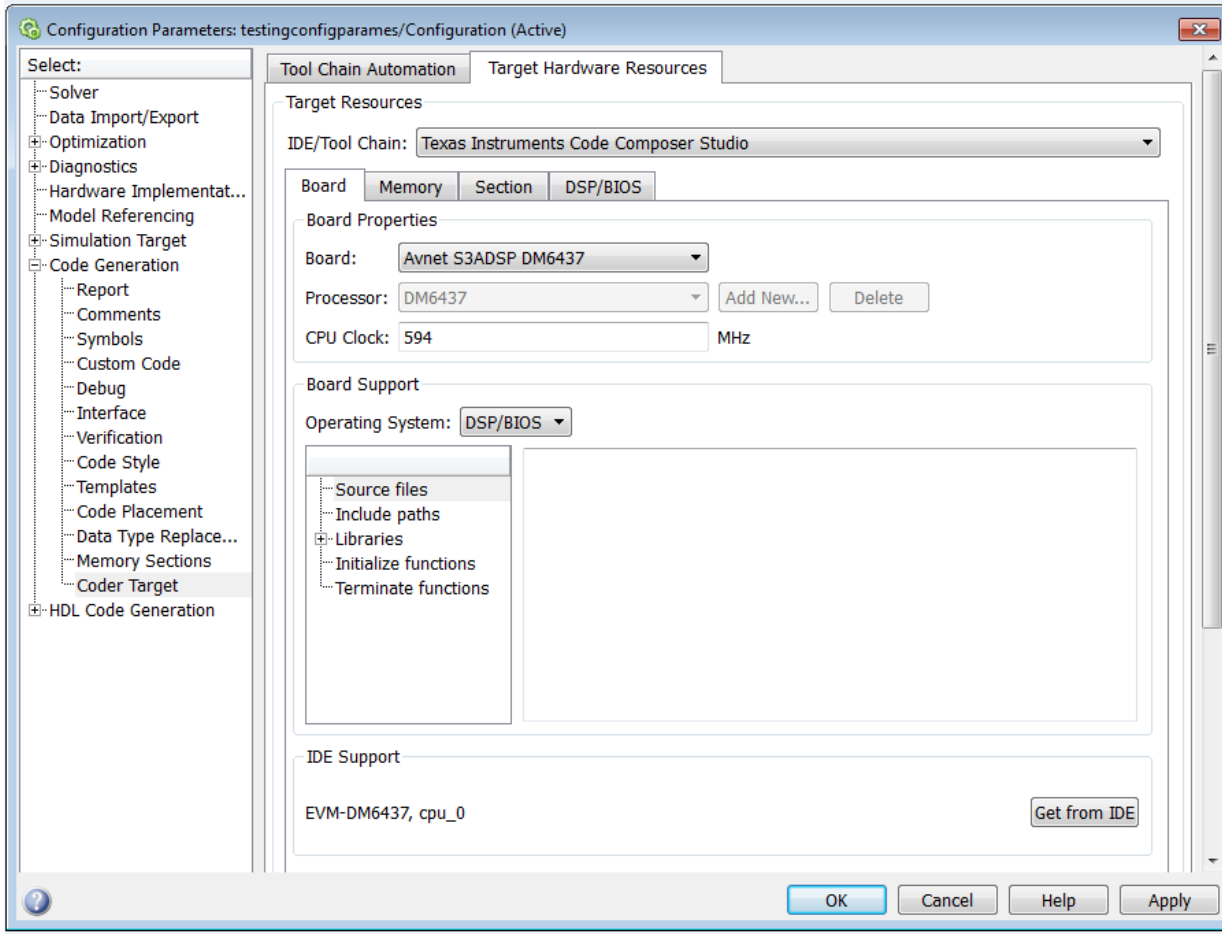
For more information, see Map Semaphore or Mutex Operations to Target-Specific Implementations.

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 13-21.

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.

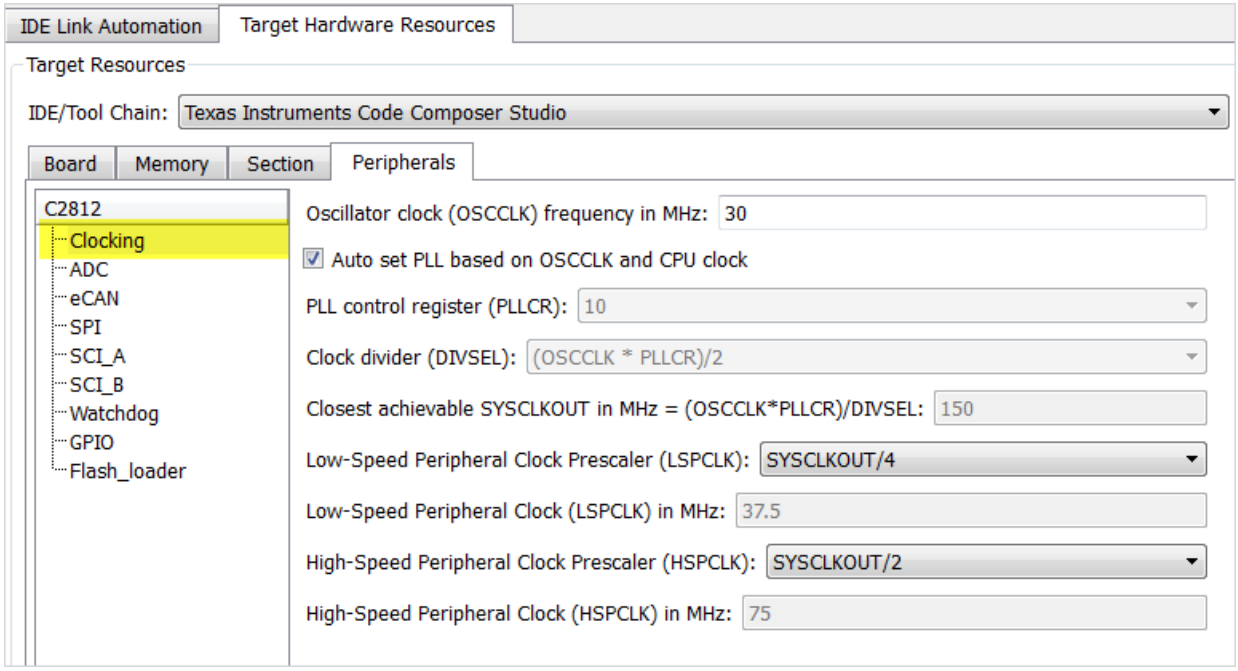
After installing the support package, you can open the block library. In the MATLAB Command Window, enter `adivdsplib`. The “Embedded Coder Support Package for Analog Devices DSPs” block library is also available in the Simulink Library Browser.

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.

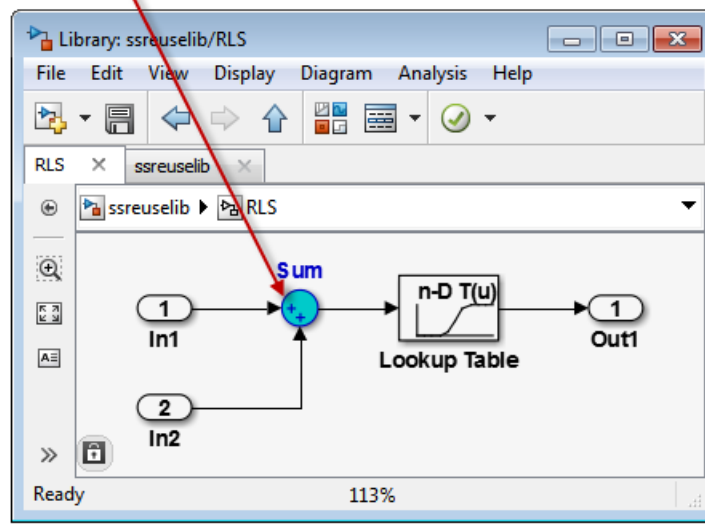
For more information, see Specify Hardware Timer.

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_binlxpw(*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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2012b

Version: 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., `logsout`) 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 14-8.

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 14-4.

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
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.ExecutionProfileSection.getMaxTicks	Call is forwarded to coder.profile.ExecutionTimeSection.MaximumExecutionTimeInTicks without warning message.	`MaximumExecutionTimeInTicks`	Uses property syntax
rtw.pil.ExecutionProfileSection.getName	Call is forwarded to coder.profile.ExecutionTimeSection.Name without warning message.	`Name`	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.pil.ExecutionProfile-Section.getNumCalls	Call is forwarded to coder.profile.Execution-TimeSection.NumCalls without warning message.	`NumCalls`	Uses property syntax
rtw.pil.ExecutionProfile.getSectionNumber	Call is forwarded to coder.profile.Execution-Time.Number without warning message.	`Number`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getTicks	Call is forwarded to coder.profile.Execution-TimeSection.Execution-TimeInTicks 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.Execution-TimeSection.TotalExecutionTimeInTicks without warning message.	`TotalExecution-TimeInTicks`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getSampleOffset	Call is forwarded to coder.profile.Execution-TimeSection.SampleOffset without warning message.	`SampleOffset`	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2012a

Version: 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 in the Simulink documentation.

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_sfun_ndarray_add):'<S1>/rtwdemo_sfun_ndarray_add' */

array3d_add(rtb_Output1,tmp,tmp_0,1,2,3);
```

Now, the generated code is:

```
int32_T i;

/* S-Function (rtwdemo_sfun_ndarray_add):'<S1>/rtwdemo_sfun_ndarray_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.viewTfl(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

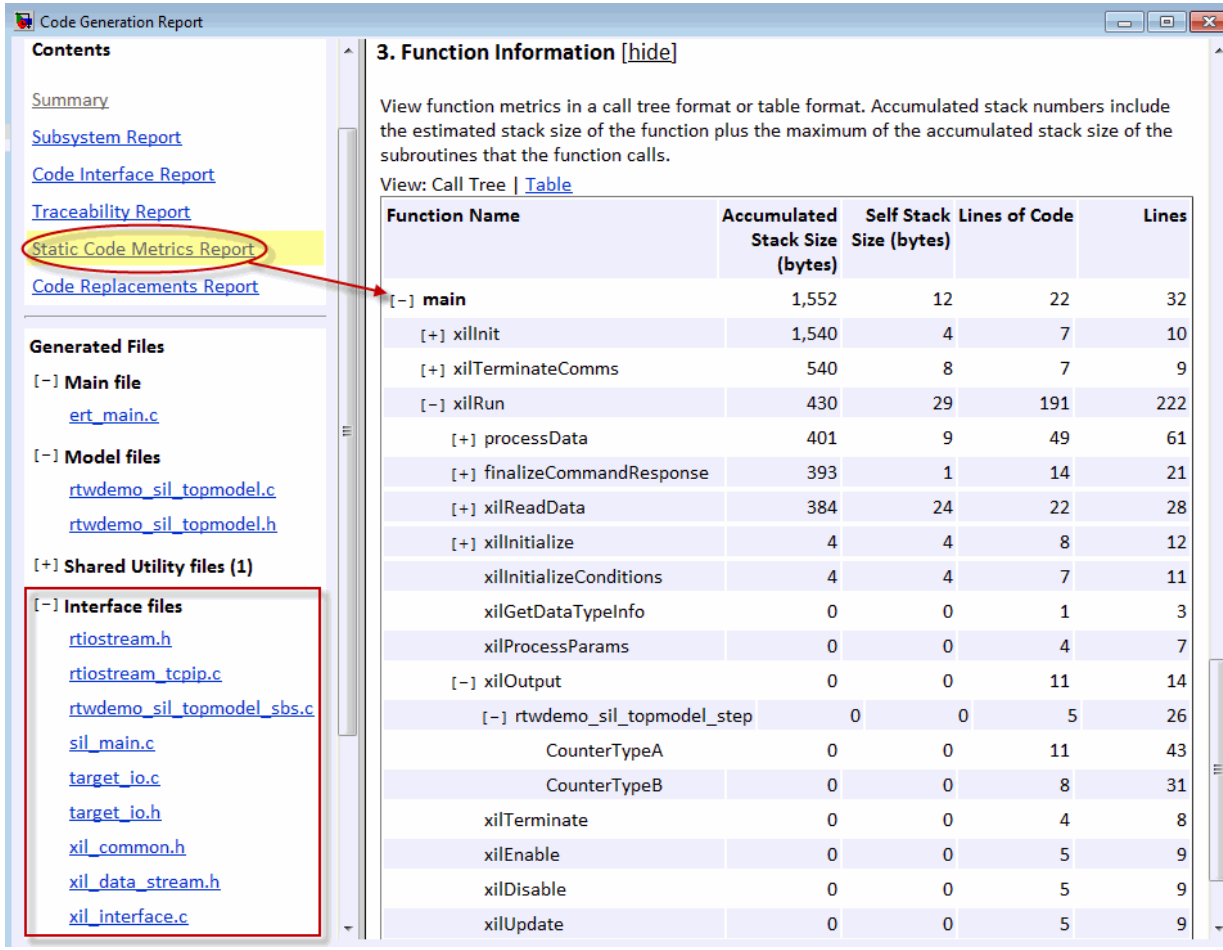
In code generation with MATLAB Coder or Simulink Coder, if you used a CRL to replace a cast in your replacement function, silent incorrect numerical results may occur. The numerical results will not change if the input fi object has binary-point scaling and zero fractional length. To optimize code generation, 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.



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, seeHardware 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` in the MATLAB Reference documentation.

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 `logsout`.
- 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 in the Simulink documentation.

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 15-14
- "Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors" on page 15-14
- "Support Removed for Freescale MPC5xx" on page 15-14
- "Limitation: Parallel Builds Not Supported for Embedded Targets" on page 15-15

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 `rtwdemo_roll` model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. `rtwdemo_roll` replaces `rtwdemo_f14`.
`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 `targetinstaller` or `supportPackageInstaller` to install the *Embedded Coder Support Package for Texas Instruments C2000 Processors*.

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).

Demo...	Now...
`rtwdemo_crl_script`	• 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 `rtwdemo_eml_aero_radar`, which contains a MATLAB script.
`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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

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 in the Embedded Coder documentation.

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 in the Embedded Coder documentation.

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 16-5
- “Support Added for Wind River VxWorks 6.8” on page 16-6
- “Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335” on page 16-6
- “New Target Function Library for Intel IPP/SSE (GNU)” on page 16-6
- “Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors” on page 16-6
- “Support Removed for Altium TASKING” on page 16-7
- “Support Removed for Infineon C166” on page 16-7
- “Support Ending for Green Hills MULTI in a Future Release” on page 16-7
- “Support Ending for Freescale MPC5xx in a Future Release” on page 16-7

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 Configurationand 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_pil demo.

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 ^ 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

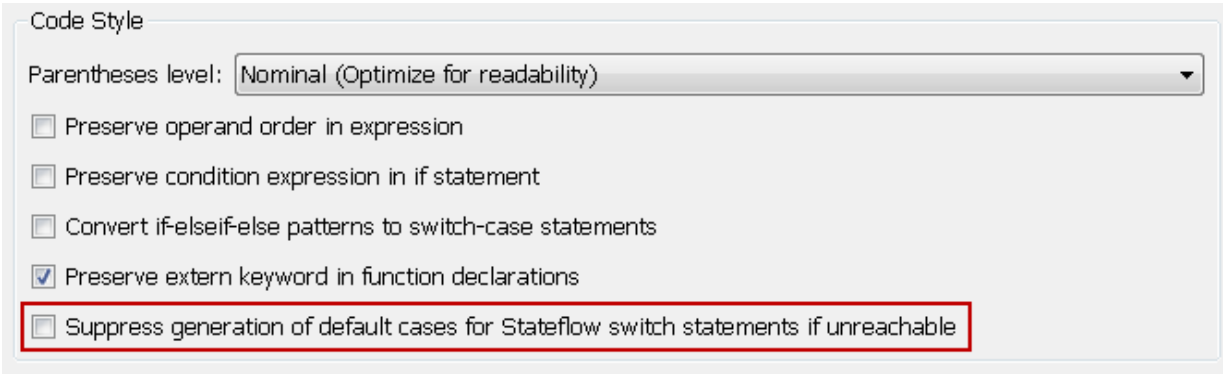
Running this Model Advisor check helps you identify logic blocks that do not use `boolean` for the output data type.

For more information about the Model Advisor, see Consulting the Model Advisor in the Simulink documentation.

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...
`rtwdemo_pmsmfoc_script`	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine
`rtwdemo_sil_pil_script`	Incorporates code execution profiling
`rtwdemo_tfl_script`	Shows how you can align nonscalar data passed into a target function library (TFL) code replacement function
`fuelsys_pil`	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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

R2011a

Version: 6.0

New Features

Bug Fixes

Compatibility Considerations

Coder Product Restructuring

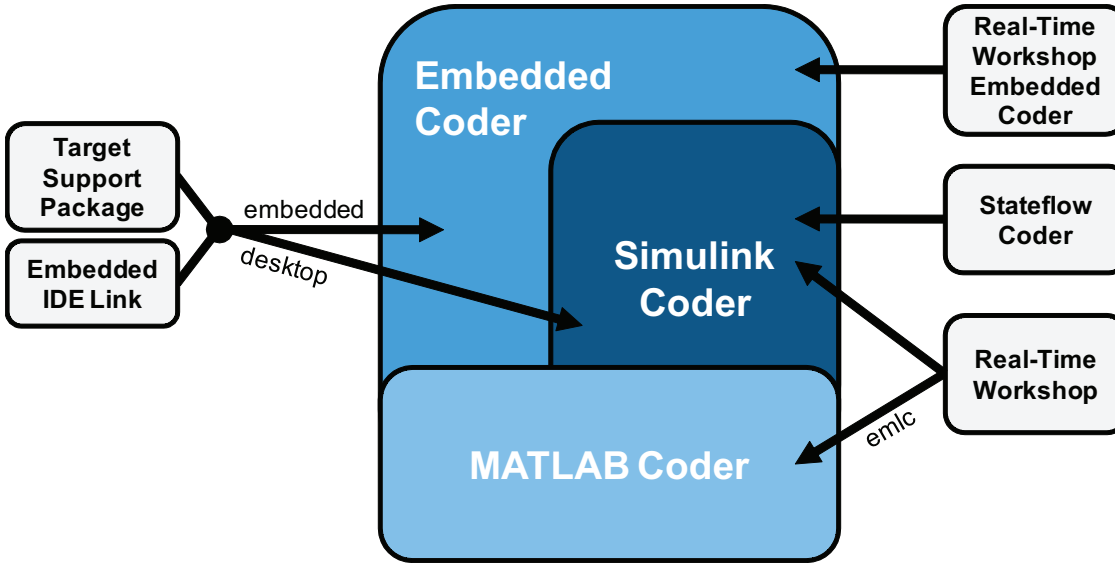
- “Product Restructuring Overview” on page 17-2
- “Resources for Upgrading from Real-Time Workshop Embedded Coder” on page 17-3
- “Migration of Embedded MATLAB Coder Features to MATLAB Coder” on page 17-4
- “Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder” on page 17-4
- “Interface Changes Related to Product Restructuring” on page 17-5
- “Simulink Graphical User Interface Changes” on page 17-5

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 MATLAB Coder function `codegen` replaces the Real-Time Workshop function `emlc`. The `emlc` function still works in R2011a but generates a warning, and will be removed in a future release. For more information, see Generating C/C++ Code from MATLAB Code in the MATLAB Coder documentation.

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

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
	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**

Where...	Previously...	Now...
Subsystem Parameter dialog box	Following parameters on main pane: • **Real-Time Workshop system code** • **Real-Time Workshop function name options** • **Real-Time Workshop function name** • **Real-Time Workshop file name options** • **Real-Time Workshop file name (no extension)**	On new **Code Generation** pane and renamed: • **Function packaging** • **Function name options** • **Function name** • **File name options** • **File name (no extension)**

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 17-7
- “No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects” on page 17-7
- “Parts of Data Class Infrastructure Not Available” on page 17-8
- “No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer” on page 17-8
- “Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release” on page 17-9

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...
`ExportedGlobal`	Data definition and declaration
`ImportedExtern`	Data declaration
`ImportedExternPointer`	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 in the Embedded Coder documentation.

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 in the Embedded Coder documentation.

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 in the Embedded Coder documentation.

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 performs 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 `cgv.CGV`	Errors	`LogMode` parameter in `cgv.Config`	To check your model before running CGV, pass the `LogMode` parameter to the constructor for `cgv.Config`. Then call the cgv.Config.configModel method to adjust the model configuration.
`Processor` removed from `cgv.CGV`	Errors	`Processor` parameter in `cgv.Config`	To check your model before running CGV, pass the `Processor` parameter to the constructor for `cgv.Config`. Then call the cgv.Config.configModel method to adjust the model configuration.
`SaveModel` removed from `cgv.CGV`	Errors	`SaveModel` parameter in `cgv.Config`	To check your model before running CGV, pass the `SaveModel` parameter to the constructor for `cgv.Config`. Then call the cgv.Config.configModel method to adjust the model configuration.

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
`ConfigModel` removed from `cgv.CGV`	Warns if set to `off` Errors if set to `on`	cgv.Config.configModel method	To check your model before running CGV, replace the `cgv.CGVConfigModel` parameter with a call to the cgv.Config.configModel method
`CheckInterface` parameter from `cgv.CGV`	Warns if set to `off` Errors if set to `on`	`CheckOutports` parameter in `cgv.Config`	To check your model before running CGV, pass the `CheckOutports` parameter to the constructor for `cgv.Config`. Then call the cgv.Config.configModel method to adjust the model configuration.
`tasking` and `custom` values removed from the `Connectivity` parameter of `cgv.CGV`	Errors	`pil`, a new value for the `cgv.CGV` `Connectivity` parameter	Replace calls to the `cgv.CGV` 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 `cgv.Config`	Defaults to `on`. Compiles the model. Then checks that the model outport configuration is compatible with the `cgv.CGV` object.	If your script fixes errors reported by `cgv.Config`, you can set `CheckOutports` to `off`.
`LogMode` parameter from `cgv.Config`	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 in the Simulink documentation.

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 Outport 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 in the Embedded Coder documentation.

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 in the Embedded Coder documentation.

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 in the Simulink documentation.

Changes for Embedded IDEs and Embedded Targets

- “Feature Support for Embedded IDEs and Embedded Targets” on page 17-21
- “Execution Profiling during PIL Simulation” on page 17-21
- “Location of Blocks for Embedded Targets” on page 17-21
- “Location of Demos for Embedded IDEs and Embedded Targets” on page 17-23
- “Multicore Deployment with Rate-Based Multithreading” on page 17-24
- “Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)” on page 17-24
- “Changes to Frame-Based Processing” on page 17-24
- “New Support for Analog Devices Blackfin BF50x and BF51x Processors” on page 17-26
- “Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors” on page 17-26
- “Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI” on page 17-26
- “Support for Texas Instruments Delfino C2834x Processors” on page 17-26
- “Ending Support for Altium TASKING in a Future Release” on page 17-27

- “Ending Support for Freescale MPC5xx in a Future Release” on page 17-27
- “Ending Support for Infineon C166 in a Future Release” on page 17-27
- “Removed Methods and Arguments” on page 17-27

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

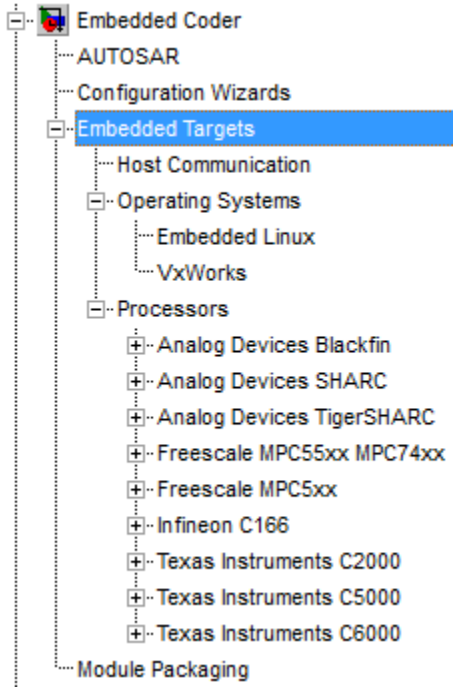
Note You can only use these features in the 32-bit version of your MathWorks products. To use these features on 64-bit hardware, install and run the 32-bit versions of your MathWorks products.

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.



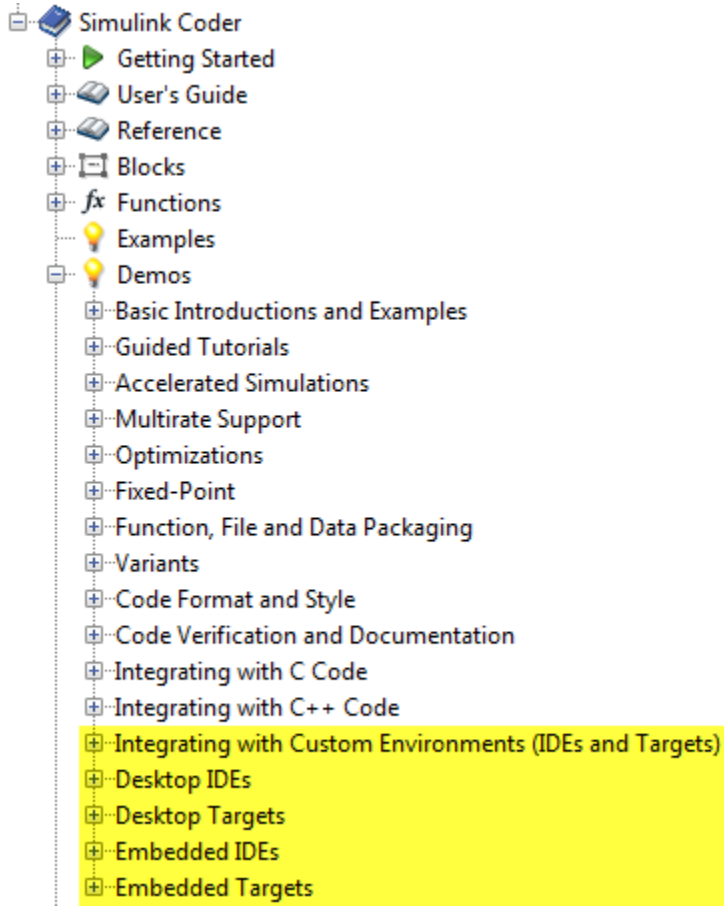
Embedded Targets includes the following types of blocks:

- Host Communication
- Operating Systems
  - Embedded Linux
  - VxWorks
- Processors
  - Analog Devices Blackfin
  - Analog Devices SHARC
  - Analog Devices TigerSHARC
  - Freescale MPC55xx 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 `slupdate` function. This function detects blocks that have **Input processing** set to `Inherited (this choice will be remove - 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 `slupdate` function. Therefore, upgrade your existing modes using `slupdate` 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...
`coderdemo_tfl`	Use target function libraries (TFLs) to replace operators and functions in code generated by MATLAB Coder.
`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/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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