

Embedded Coder[®] Release Notes



MATLAB[®]&SIMULINK[®]

How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Embedded Coder[®] Release Notes

© COPYRIGHT 2011–2015 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.

AUTOSAR improvements including multi-runnable modeling and code efficiency	1-2
More efficient code involving model references, unit delays, and global data references	1-2
Reusable custom storage class for Model block input/output ports	1-2
Reuse input, output, and state of Unit Delay block	1-3
Enhanced variable reuse optimizations	1-4
Strategic caching of global variable references	1-5
Enhanced global variable localization optimizations	1-6
Simplified Code Replacement Library specification plus more replacements involving integer operations	1-8
Simplified Code Replacement Library specification	1-8
More replacements involving integer operations	1-9
Control of Boolean and data type limit identifiers in generated code	1-10
Combined input/output arguments with function prototype control	1-10
SIL/PIL for protected models and SIL source code debugging using Microsoft Visual Studio Express	1-10
SIL/PIL for protected models	1-11
SIL source code debugging using Microsoft Visual Studio Express	1-11
Improved MISRA-C compliance for bitwise operations on signed integers	1-12
Improved readability for shared header file 'rtwtypes.h'	1-12
Code Generation from MATLAB Code	1-15
Indent style and size control for generated C/C++ code	1-15
Improved MISRA-C compliance for bitwise operations on signed integers	1-16
Improved MISRA-C type cast compliance	1-17

Model Architecture and Design	1-19
AUTOSAR multi-runnable modeling using Simulink rate-based multitasking	1-19
Enhanced modeling with AUTOSAR system constants	1-19
AUTOSAR CompuMethod enhancements	1-19
Preprocessor conditionals for single variant choice	1-20
Data, Function, and File Definition	1-21
Names of built-in storage classes reserved	1-21
Code Generation	1-22
New and enhanced Model Advisor checks for MISRA-C compliance	1-22
Improved traceability for AUTOSAR RTE implicit read	1-22
Configurable aliveTimeout value for AUTOSAR ports	1-23
AUTOSAR calibration parameter export for COM_AXIS lookup tables	1-24
Fixed-point scaling information in Code Interface Report ..	1-24
Unsigned integer minimum data limit identifiers	1-24
Default iteration variable data type	1-25
Deployment	1-27
Code Replacement Viewer enhanced	1-27
Model configuration parameter considered for division operator code replacements	1-27
Lookup table algorithm parameter specification enhancements	1-27
Header file for Basic Linear Algebra Subroutine (BLAS) multiplication function code replacement example changed	1-28
Code replacement detection of overflow and rounding mode equivalence	1-28
Updates to Embedded Coder Support Package for ARM Cortex-A Processors	1-28
Feature being removed in a future release	1-29
Performance	1-30
Conditional compilation of Data Store Memory block memory definition and declaration	1-30

Ternary Boolean expressions transformed into assignment statements	1-31
Verification	1-32
Model block SIL/PIL parameter renamed	1-32
ERT S-Function block no longer supported for AUTOSAR ..	1-32
SIL/PIL support for replacing <code>boolean</code> data type with <code>int8</code>	1-32
SIL/PIL support for generated access methods for C++ model class root-level I/O signals	1-32
Check bug reports for issues and fixes	1-34

R2014b

AUTOSAR targeting updates including 4.1 ARXML, client/server with Simulink Functions, multi-instance components, and IFL/IFX libraries	2-2
Embedded Coder support packages for AUTOSAR, TI Concerto, and Freescale FRDM-KL25Z	2-3
Processor-in-the-loop (PIL) verification and execution profiling for MATLAB code	2-3
Reduced RAM and faster execution for modeling patterns including select-assign-iterate blocks, subsystem interfaces, and model references	2-4
Example Model	2-4
In-place assignments for select-assign-iterate pattern ..	2-6
Subsystem signal information	2-7
Variable reuse around call site	2-8
Enhanced reporting of eliminated blocks	2-9
Improved MISRA-C type cast compliance	2-10
Code Generation from MATLAB Code	2-11
Software-in-the-loop verification improvements for MATLAB Coder	2-11
Additional options for custom banners and comments in C and C++ code generated from MATLAB code	2-11
Highlighting of potential data type issues in code generation reports	2-12

Model Architecture and Design	2-17
AUTOSAR client and server modeling	2-17
Global From and Goto blocks for AUTOSAR modeling	2-17
AUTOSAR IRV branch from outport signal allowed outside runnable	2-17
Data, Function, and File Definition	2-20
Constant sample time limitation for AUTOSAR models	2-20
Iteration variable in For Iterator block uses signal name	2-20
Data type replacement specification can be used across models	2-20
Definition file for grouped custom storage classes	2-20
Type definition location for custom storage classes	2-21
GetFunction and SetFunction included in checks for identifier clash	2-21
Code Generation	2-22
Support Package for AUTOSAR Standard	2-22
AUTOSAR help navigation enhancements	2-22
Support for AUTOSAR Release 4.1	2-23
AUTOSAR 4.1 ARXML and C code generation	2-23
AUTOSAR 4.1 InitEvent support	2-24
AUTOSAR 4.1 provide-require port support	2-24
Multi-instance AUTOSAR atomic software components	2-24
AUTOSAR arxml import and export	2-25
AUTOSAR R4.x compliant data type support	2-25
AUTOSAR CompuMethod control	2-26
Improved AUTOSAR package configuration	2-28
AUTOSAR calibration component export	2-29
Simulink Min and Max mapping to AUTOSAR physical data constraints	2-29
AUTOSAR addPackageableElement replaces add*Interface functions	2-29
Code generation report with enhanced navigation and integrated access to code metrics data	2-30
Updated license requirements for viewing code generation report	2-31
Option for doxygen style comments in generated code	2-31
Dynamic memory allocation parameters renamed	2-32
Template makefile compatibility with execution time profiling	2-32

Intel Performance Primitives (IPP) platform-specific code replacement libraries for cross-platform code generation	2-33
Deployment	2-34
Relational operator replacement	2-34
Code replacement involving vector and matrix data	2-34
Trigonometry function replacement	2-34
Replacement of shift and cast operations involving vector and matrix operands	2-35
Algorithm specification for addition and subtraction operator replacement	2-35
Improved code replacement with output type cast absorption	2-35
Lookup table function code replacement extended to 30 dimensions	2-36
Rounding mode support for lookup table function replacement	2-36
Algorithm parameter value sets in code replacement table entries	2-36
coder.replace support for functions specified with varargin input variable	2-37
Documentation installation with hardware support package	2-37
Support package for Altera SoC platform	2-37
Support package for BeagleBone Black hardware	2-38
Support for Eclipse IDE has been removed	2-38
Support for Green Hills MULTI IDE has been removed	2-38
Support for Texas Instruments C5000 DSPs will be removed	2-38
Performance	2-39
Global variable localization optimizations	2-39
Verification	2-41
Top-model code testing with Model block SIL and PIL	2-41
SIL/PIL support for Simulink Function and Function Caller blocks	2-41
SIL debugging support for Linux	2-42
PIL support for test hardware approach	2-42
SIL/PIL support for model initialization dynamic memory allocation	2-42
Check bug reports for issues and fixes	2-43

Capability to merge AUTOSAR authoring tool changes into Simulink models as part of round-trip iterations	3-2
Custom storage class and optimized class declarations for C++ class code generation	3-4
Custom storage class support for C++ class code generation	3-4
Improved code for C++ model class declarations	3-4
In-place function replacement with <code>coder.replace</code> in MATLAB and lookup table code replacement for Simulink	3-4
In-place function replacement with <code>coder.replace</code> in MATLAB	3-4
Lookup table code replacement for Simulink	3-5
ARM Cortex-A optimized code generation using Ne10 library	3-5
Template to customize code generation output for MATLAB Coder	3-6
AUTOSAR 4.0 static and constant memory, AUTOSAR-typed per-instance memory, and <code>VariationPointProxy</code>	3-6
Static and constant memory	3-6
AUTOSAR-typed per-instance memory	3-6
Variation point proxy	3-7
Additional options for reuse of global variables	3-7
Code Generation from MATLAB Code	3-8
In-place function replacement with <code>coder.replace</code> in MATLAB	3-8
Single-line (<code>//</code>) comment style available for generated code	3-8
Software-in-the-loop verification for MATLAB Coder	3-9
Change of default value for <code>MATLABFcnDesc</code>	3-10
Model Architecture and Design	3-11
Specify AUTOSAR runnable symbol name distinct from short-name	3-11
Improved AUTOSAR arxml support for measurement and calibration	3-12
AUTOSAR data dictionary support	3-12
Configure AUTOSAR Interface button removed from AUTOSAR Code Generation Options	3-13

Subsystem methods of AUTOSAR arxml.importer class removed	3-13
Data, Function, and File Definition	3-15
Constant sample time limitations for root-level Output blocks	3-15
Example model rtwdemo_cppencap renamed to rtwdemo_cppclass	3-15
Unit Delay block optimization	3-15
Code Generation	3-16
Global variable usage available in the static code metrics report	3-16
Single-line (//) comment style available for generated code ..	3-16
Code indentation support for namespace declarations in generated code	3-17
AUTOSAR C code generation enhancements	3-17
Static main program module for C++ class code generation ..	3-18
Error message for data type replacement and classic call interface conflict	3-18
Deployment	3-20
Lookup table code replacement for Simulink	3-20
Replacement of functions that take vector and matrix arguments	3-20
Logical data type support for arguments of replaced functions	3-21
Code replacement data alignment for complex types	3-21
Intel IPP (ANSI) and Intel IPP (ISO) code replacement libraries are combined	3-21
Support for Eclipse IDE will be removed	3-21
Support for Green Hills MULTI IDE will be removed	3-22
Support package for ARM Cortex-A processors	3-22
Support package for Texas Instruments C6000 processors ..	3-23
Updates to support package for Texas Instruments C2000 processors	3-24
Updates to support package for Xilinx Zynq-7000 platform ..	3-24
Updates to support package for STMicroelectronics STM32F4 Discovery board	3-24
Wind River Tornado (VxWorks 5.x) example main program option to be removed in future release	3-25

Performance	3-26
Enhanced global variable optimization options	3-26
for loops used to initialize arrays to zero	3-26
Verification	3-27
Software-in-the-loop simulation for physical models	3-27
SIL verification for subsystem code generation	3-27
SIL and PIL support for fixed-point data type override	3-30
SIL and PIL support for Invoke AUTOSAR Server Operation block	3-30
SIL and PIL support for structure parameters with storage class <code>SimulinkGlobal</code>	3-30
Model block SIL and PIL with export-function and asynchronous function-call models	3-30
Model block SIL and PIL with disabled inline parameters ..	3-31
SIL and PIL block improvements	3-32
Check bug reports for issues and fixes	3-33

R2013b

Code Generation from MATLAB Code	4-2
Software-in-the-loop verification for MATLAB Coder	4-2
Custom generated identifiers for <code>emxArray</code> utility functions .	4-2
Model Architecture and Design	4-4
Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior	4-4
Enhanced modeling and simulation of AUTOSAR multiple runnables	4-4
Enhanced ARXML import of AUTOSAR software component internal behavior	4-4
Ability to model AUTOSAR mode receiver ports and events	4-5
AUTOSAR dual-scaled parameter	4-5

Programmatic interface for configuring AUTOSAR properties and Simulink-AUTOSAR mapping	4-5
Reorganization of Model Advisor Embedded Coder checks . . .	4-6
Model Advisor fixed-point checks with additional coverage and optimization awareness	4-7
Protected model Web view	4-7
RTW.AutosarInterface class to be removed in a future release	4-7
Subsystem methods of arxml.importer class to be removed in a future release	4-8
Data, Function, and File Definition	4-9
Simplified global types file <code>rtwtypes.h</code> with invariant content	4-9
C++ encapsulation support for name space control and template-based file customization	4-9
Name space control for scoping C++ encapsulated model classes	4-9
Template-based customization of encapsulated C++ header and source files	4-10
Shared utility naming control	4-10
Expanded support for identifier names	4-11
Terminate function setting honored for subsystems and referenced models	4-11
Code Generation	4-12
Support for AUTOSAR release 4.0.3 XML and generated code	4-12
Indent style and size control for code generation	4-12
Subsystem functions return value in generated code	4-12
Model reference step function void input and output arguments	4-13
Deployment	4-14
ARM Cortex-M optimized code with STM32F4-Discovery board example	4-14
Support package for ARM Cortex processors	4-14
Support package for STMicroelectronics STM32F4- Discovery Board	4-14
Wind River VxWorks 6.9 support	4-15
Compatibility Considerations	4-15

Support package for Texas Instruments C2000 processors . . .	4-16
Compatibility Considerations	4-16
Coder Target pane in Configuration Parameters dialog box . .	4-17
ZedBoard hardware support	4-18
Simplified multi-instance code interface and dynamic memory allocation for ERT targets	4-18
Addition and Subtraction Operator Code Replacement Assumes Cast-Before-Operation Behavior	4-20
Performance	4-22
Reusable custom storage class to reduce root I/O memory . .	4-22
Subsystem functions reused independently of output connection	4-22
Verification	4-23
SIL and PIL support fixed-point data types wider than 32 bits	4-23
SIL and PIL protected model support	4-24
Code execution profiling improvements	4-24
Standalone code generation with function profiling . . .	4-24
Display of code section invocations	4-24
SampleOffset and SamplePeriod removed	4-25
Check bug reports for issues and fixes	4-26

R2013a

Code Generation from MATLAB Code	5-2
Improved code replacement traceability for MATLAB code generation	5-2
Static code metrics report for MATLAB Coder	5-2
Model Architecture and Design	5-4

AUTOSAR user interface and round trip ARXML file import and export improvements	5-4
Improved graphical user interfaces for AUTOSAR configuration	5-4
Round-trip preservation of AUTOSAR elements and UUIDs	5-6
Code generation for variable-size scalar signals	5-7
Data, Function, and File Definition	5-8
Shortened system-generated identifier names	5-8
Improved data initialization with custom storage classes	5-8
Default specification for global types	5-8
Subsystem block parameter Function packaging option renamed	5-8
Code Generation	5-9
Model Advisor checks for code generation	5-9
Deployment	5-10
Concurrent execution API to target embedded multicore platforms	5-10
Semaphore and mutex code replacement for multicore target environments	5-10
Hardware timer function replacement	5-10
Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box	5-11
Downloadable support and blocks for Analog Devices DSPs	5-12
Texas Instruments C2000 Clocking Options	5-12
Support for Texas Instruments C2802x and Texas Instruments C2803x variants	5-13
Downloadable support and blocks for Xilinx Zynq-7000 platform	5-14
Support ending for Eclipse IDE in a future release	5-14
Support ending for remoteBuild method in a future release	5-15
Performance	5-16
Optimized function arguments for nonreusable subsystems	5-16
Reduced data copies for tunable parameter expressions	5-16
Removal of unused global variables	5-17

Verification	5-18
Debugging during SIL simulations	5-18
Simulation of multiple SIL Model blocks in a top model ...	5-18
API for testing <code>rtiostream</code> communications	5-18
SIL and PIL support for targets with multicore processors .	5-19
Additional code annotation for justifying Polyspace checks .	5-20
Code execution profiling improvements	5-20
Comprehensive measurement and reporting of function execution times	5-20
Viewing and comparing execution time plots with the Simulation Data Inspector	5-20
Specification of hardware timer through the Code Replacement Tool	5-21
Code-to-model traceability links for reusable subsystems in libraries	5-21
Check bug reports for issues and fixes	5-23

R2012b

Cyclomatic complexity measurement in static code metrics report	6-2
Custom code substitution for MATLAB functions using code replacement libraries	6-2
SIL and PIL support for signal logging, encapsulated C++, and AUTOSAR calibration parameters	6-2
Signal logging for SIL and PIL simulations	6-3
Use SIL and PIL simulations to verify encapsulated C++ code	6-3
Improved SIL and PIL verification for AUTOSAR-compliant code	6-3
AUTOSAR 4.0 nonscalar data support	6-4
Code annotation for justifying Polyspace checks	6-4

Texas Instruments Code Composer Studio IDE 5.1 support .	6-4
External mode support for ERT targets with static main . . .	6-5
Downloadable support for Green Hills MULTI	6-5
Support for Texas Instruments C2806x processors	6-6
Performance enhancement of Simulink data objects	6-7
AUTOSAR software component import and export	
enhancements	6-8
Import validation	6-8
Faster import and export of arxml files	6-8
Explicit access mode for AUTOSAR Sender and Receiver	
ports	6-9
Import port-based calibration parameters	6-9
Highlight virtual blocks in model Web view of code	
generation report	6-9
Code Execution Profiling Improvements	6-9
Updated Code Execution Profiling API	6-9
Code Execution Profiling Supports Single Object Output . . .	6-12
Incremental Compilation with Changes in Code Coverage	
Settings	6-13
Check bug reports for issues and fixes	6-14

R2012a

AUTOSAR Enhancements	7-2
AUTOSAR Release 4.0	7-2
Support for Schema 2.0 Removed	7-2
Code Efficiency Enhancements	7-2
For Each Subsystem Loop Bound Passed by Value	7-2
Fully Inlined S-functions from Legacy Code Tool	7-3

Element-Wise Operations as Inputs to Intrinsic Functions . . .	7-3
Enhancements to Custom Storage Classes in Simulink and mpt Packages	7-4
Code Generation Report Includes Simulink Web View	7-4
LDRA Testbed Code Coverage Annotations in Code Generation Report	7-5
Generated Identifiers Enhancements	7-5
Simplified Identifiers for Model Reference Code	7-5
Consistent Identifiers for Comparing Generated Code	7-5
Code Replacement Enhancements	7-6
Target Function Libraries Renamed to Code Replacement Libraries	7-6
Enhanced Code Replacement Traceability	7-7
Code Replacement Support for Simulink Matrix Division and Inversion Operators	7-8
Code Replacement Support for MATLAB Coder fix, hypot, round, and sign Functions	7-8
Integer Functions Now Return Real-World Values	7-8
SIL and PIL Enhancements	7-8
SIL and PIL Test Harness Files in Code Generation Report . . .	7-9
PIL Support for Code Coverage with LDRA Testbed	7-10
Seamless Switching Between SIL and PIL for Top-Model and Model Block	7-10
Enhanced Hardware Implementation Support	7-10
Top-Model Output Limitations Removed	7-11
Model Block SIL/PIL Support for Absolute Time	7-12
Changes for ERT and ERT-Based Targets	7-12
Changes for Embedded IDEs and Embedded Targets	7-13
Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE	7-13
Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors	7-13
Support Removed for Freescale MPC5xx	7-14

Limitation: Parallel Builds Not Supported for Embedded Targets	7-14
New and Enhanced Demos	7-14
Check bug reports for issues and fixes	7-16

R2011b

Static Code Metrics in Code Generation Report	8-2
AUTOSAR Enhancements	8-2
Import and Export of AUTOSAR Sensor/Actuator Components	8-2
Improved Simulink Library Support for Multiple Runnables .	8-2
AUTOSAR Schema Version 3.2	8-3
Export AUTOSAR XML as Single File	8-3
SIL and PIL Enhancements	8-3
Code Execution Profiling of Functions in Subsystems and Model Blocks	8-3
Code Coverage with LDRA Testbed	8-3
BitField and GetSet Custom Storage Classes	8-3
Model Blocks with Variable-Size Signals	8-4
Verification of Generated C++ Code	8-4
Generate Multitasking Code for Concurrent Execution on Multicore Processors	8-4
Changes for Embedded IDEs and Embedded Targets	8-5
64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0	8-5
Support Added for Wind River VxWorks 6.8	8-6
Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335	8-6
New Target Function Library for Intel IPP/SSE (GNU)	8-6

Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors	8-6
Support Removed for Altium TASKING	8-7
Support Removed for Infineon C166	8-7
Support Ending for Green Hills MULTI in a Future Release .	8-7
Support Ending for Freescale MPC5xx in a Future Release . .	8-7
Saturation Control of Stateflow Data	8-7
Custom Storage Class Properties for Managing Data Ownership and Definition	8-8
Export Data Declarations to Shared Header File for Code Generation with Model Reference	8-9
Target Function Library Code Replacement Enhancements	8-9
Code Replacement Tool for Creating and Managing TFL Tables	8-9
Ability to Align Data Objects to TFL-Specified Boundaries to Boost Code Performance	8-10
Support for Replacing Element-wise Matrix Multiply	8-11
Code Generation Enhancements	8-11
Redundant Condition Checks	8-11
Loop Fusion	8-12
Invariant Condition Check Lifting	8-12
Parameter Pooling for Stateflow and Interpreted MATLAB Function Blocks	8-12
Readability Improvement for Reusable Subsystem Input and Output	8-12
Enhanced Code Generation Optimization Using Minimum and Maximum Values	8-12
New Model Advisor Check for Code Efficiency of Logic Blocks	8-13
Control of Default Case Generation for Switch Statements in Generated Code for Stateflow Charts	8-13
Improvement to Build Process for Conflicting Identifiers .	8-14

Update to Code Generation Verification Class <code>cgv.Config</code>	8-15
License Names Not Yet Updated for Coder Product Restructuring	8-15
New and Enhanced Demos	8-15
Check bug reports for issues and fixes	8-17

R2011a

Coder Product Restructuring	9-2
Product Restructuring Overview	9-2
Resources for Upgrading from Real-Time Workshop Embedded Coder	9-3
Migration of Embedded MATLAB Coder Features to MATLAB Coder	9-4
Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder	9-4
Interface Changes Related to Product Restructuring	9-5
Simulink Graphical User Interface Changes	9-5
Data Management Enhancements and Changes	9-6
Memory Section Enhancements	9-6
No Longer Able to Set <code>RTWInfo</code> or <code>CustomAttributes</code> Property of Simulink Data Objects	9-6
Parts of Data Class Infrastructure Not Available	9-7
No Longer Generating Pragma for Data Defined with Built-In Storage Class <code>ExportedGlobal</code> , <code>ImportedExtern</code> , or <code>ImportedExternPointer</code>	9-8
Simulink. <code>CustomParameter</code> and Simulink. <code>CustomSignal</code> Data Classes To Be Deprecated in a Future Release	9-9
AUTOSAR Enhancements	9-9
Calibration Parameters	9-9
Multiple Runnables from Virtual Subsystems	9-9
Support for Code Descriptor Elements	9-10

SIL and PIL Enhancements	9-11
Code Execution Profiling	9-11
PIL Block Parameter Tuning	9-11
Top-Model SIL/PIL and PIL Block Parameter Initialization ..	9-11
Model Block Parameter Tuning and Model Initialization ...	9-11
Code Generation Enhancements	9-12
Improved Code for Data Store Memory In-place Assignment	9-12
Improvements to Target Function Library Replacements ..	9-12
Improved Loop Fusion	9-12
Improved Array Indexing	9-12
Improvement on Matrix Parameter Pooling	9-13
Readability Improvements Involving Data References	9-13
Code Generation Verification (CGV) API Updates	9-13
Support for Adding Multiple Callback Functions	9-13
New Functionality Added to the cgv.CGV Class	9-13
MISRA-C Code Generation Objective	9-16
New Model Advisor Check for Code Efficiency of Lookup Table Blocks	9-17
Enhanced Code Generation Optimization	9-17
Target Function Library Replacement Based on Computation Method for Reciprocal Sqrt, Sine, and Cosine	9-18
Target Function Library Support for abs, min, max, and sign functions	9-18
C++ Encapsulation Allowed for Referenced Models in For Each Subsystems	9-18
Improved Code Generation for Portable Word Sizes	9-19
Improved Comments in the Generated Code	9-19
Replacement Data Types and Simulation Mode for Referenced Models	9-19
Changes for Embedded IDEs and Embedded Targets	9-19
Feature Support for Embedded IDEs and Embedded Targets	9-20

Execution Profiling during PIL Simulation	9-21
Location of Blocks for Embedded Targets	9-21
Location of Demos for Embedded IDEs and Embedded Targets	9-22
Multicore Deployment with Rate-Based Multithreading . . .	9-23
Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)	9-24
Changes to Frame-Based Processing	9-24
New Support for Analog Devices Blackfin BF50x and BF51x Processors	9-25
Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors	9-26
Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI .	9-26
Support for Texas Instruments Delfino C2834x Processors .	9-26
Ending Support for Altium TASKING in a Future Release .	9-27
Ending Support for Freescale MPC5xx in a Future Release .	9-27
Ending Support for Infineon C166 in a Future Release	9-27
Removed Methods and Arguments	9-27
Changes to ver Function Product Arguments	9-27
New and Enhanced Demos	9-28
Check bug reports for issues and fixes	9-29

R2015a

Version: 6.8

New Features

Compatibility Considerations

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 1-19
 - “Enhanced modeling with AUTOSAR system constants” on page 1-19
 - “AUTOSAR CompuMethod enhancements” on page 1-19
- Under Code Generation:
 - “Improved traceability for AUTOSAR RTE implicit read” on page 1-22
 - “Configurable aliveTimeout value for AUTOSAR ports” on page 1-23
 - “AUTOSAR calibration parameter export for COM_AXIS lookup tables” on page 1-24

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.

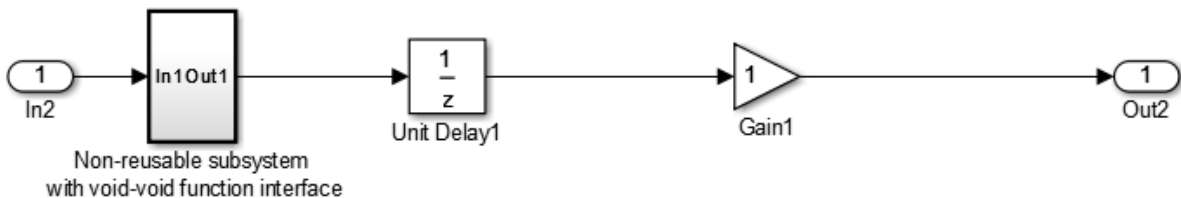
For more information on how to configure your model to take advantage of this optimization, see “Buffer Reuse for Model Block Boundary and Unit Delay”.

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();
}

```

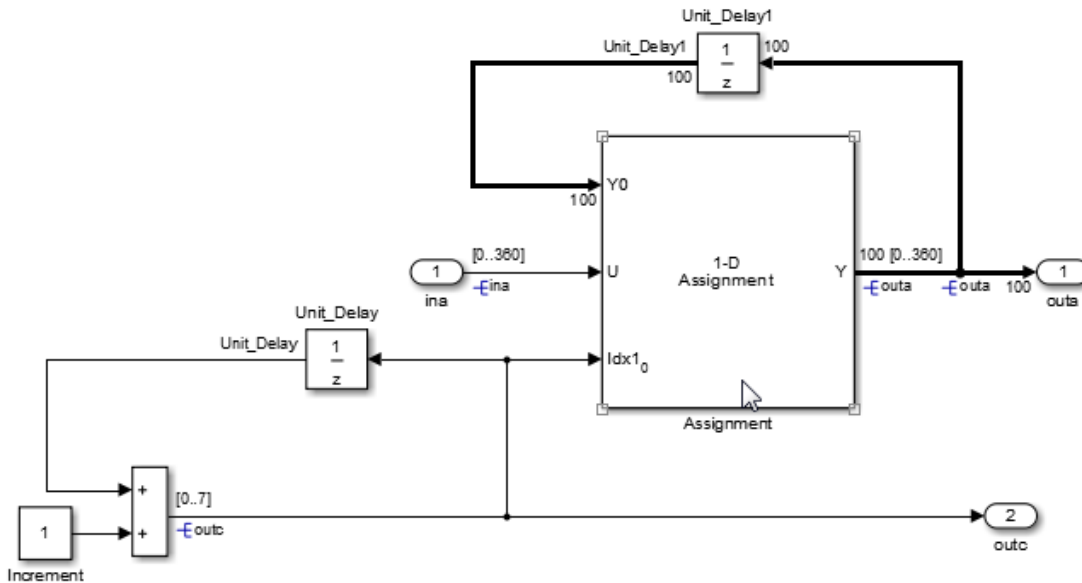
For more information on how to configure your model to use this optimization, see “Buffer Reuse for Model Block Boundary and Unit Delay”.

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;

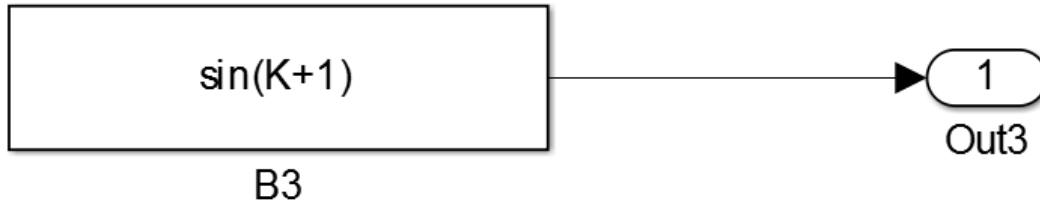
```

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, consider the following model.



The code generator produced this code in R2014b:

```
/* Output: '<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:

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

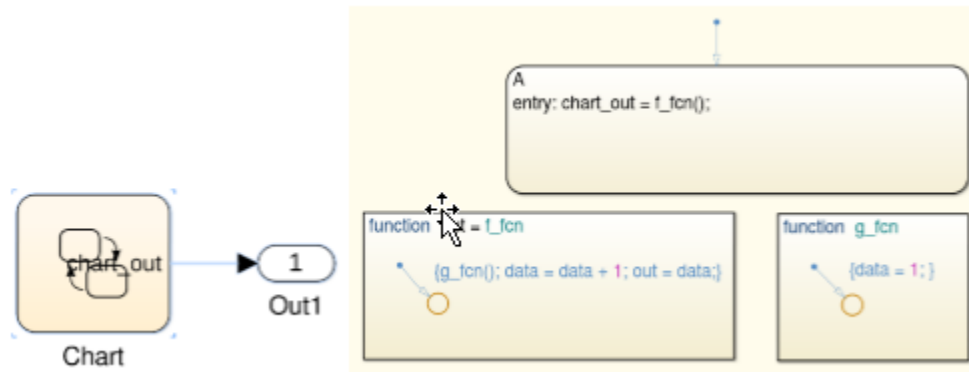
Enhanced global variable localization optimizations

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;
}
```

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

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”.

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”.

SIL/PIL for protected models and SIL source code debugging using Microsoft Visual Studio Express

- “SIL/PIL for protected models” on page 1-11

- “SIL source code debugging using Microsoft Visual Studio Express” on page 1-11

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.

Description

Create a protected model(.slxp) that allows read-only view, simulation, and code generation of the model with optional password protection.

Allow user of protected model to

Open read-only view of model Enter password (option... Enter password (optio...

Simulate Enter password (option... Enter password (optio...

Use generated code Enter password (option... Enter password (optio...

Code interface: Model reference

Content type: Model reference

 Top model

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”.

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**.

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`.

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 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”.

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

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

New and enhanced Model Advisor checks for MISRA-C compliance

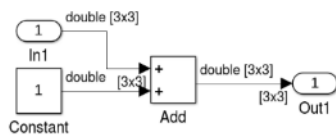
To improve MISRA-C compliance, you can run the following Model Advisor checks:

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_TmpSignalConversionAtIn10Out;
real_T tmp[9];
int32_T
/* SignalConversion: '<Root>/TmpSignal ConversionAtIn10Output1' incorporate
Inport: '<Root>/In1' */
rtb_TmpSignalConversionAtIn10Out = Rte_IRead_Runnable_Step_Input_Element0();
/* Sum: '<Root>/Add' incorporates:
 * Constant: '<Root>/Constant'
 */
for (i = 0; i < 9; i++) {
    tmp[i] = rtb_TmpSignalConversionAtIn10Out [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 **Outputs** table.

Outputs

Block Name	Code Identifier	Data Type	Scaling	Dimension
<Root>/Out1	<i>Defined externally</i>	uint32_T	ufix32_En14	1
<Root>/Out2	<i>Defined externally</i>	int32_T	sfix32_En12	1

You must have a Fixed-Point Designer™ license to see fixed-point scaling information in the report. For more information on how scaling is represented in the table, see “Fixed-Point Data Type and Scaling Notation”.

Unsigned integer minimum data limit identifiers

The following unsigned integer minimum data limit identifiers are no longer defined in `rtwtypes.h`:

- MIN_uint8_T
- MIN_uint16_T
- MIN_uint32_T

- `MIN_uint64_T`

Previously, the unsigned integer minimum data limit identifiers defined in `rtwtypes.h` were potentially not used in the generated code:

- Standard C header files do not provide an unsigned integer minimum data limit constant.
- In most instances, the code generator did not replace `0` with the unsigned integer minimum limit identifier.

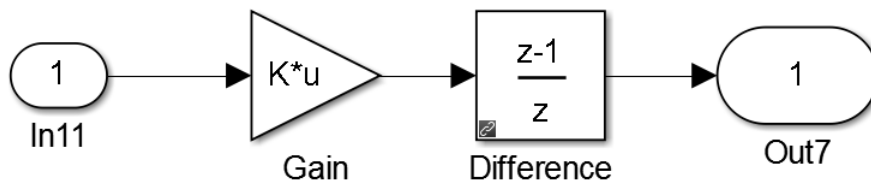
Compatibility Considerations

If you previously used unsigned integer minimum data limit identifiers in custom code, for example in an S-Function, replace the limit with `0`.

Default iteration variable data type

The default data type for iteration variables in the generated code is a 32-bit integer. Previously, the default data type was `int` with an unspecified bit size.

For example, consider the following model.



The code generator produced this code in R2014b:

```
{
  int_T i;
  for (i = 0; i < 16; i++) {
    test1_2a_Y.Out7[i].re = (0L);
    test1_2a_Y.Out7[i].im = (0L);
  }
}
```

The code generator produces this code in R2015a:

```
{
  int32_T i;
  for (i = 0; i < 16; i++) {
    test1_2a_Y.Out7[i].re = (0L);
    test1_2a_Y.Out7[i].im = (0L);
  }
}
```

Deployment

Code Replacement Viewer enhanced

- MATLAB command for invoking the Code Replacement Viewer is renamed from `RTW.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_cr1.h`. The associated include path for this header file changed to `matlab/toolbox/rtw/rtwdemos/cr1_demo`. For more information, see “Improved readability for shared header file `rtwtypes.h`”.

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.

Updates to Embedded Coder Support Package for ARM Cortex-A Processors

The Embedded Coder Support Package for ARM[®] Cortex[®]-A Processors now provides the capability to create and validate a custom target based on ARM Cortex-A processors. You specify the custom target name and decide which hardware platforms and target features to support.

To install or update this support package, perform the steps described in “Install Support for ARM Cortex-A Processors”.

For more information, see “Develop a Target”.

Feature being removed in a future release

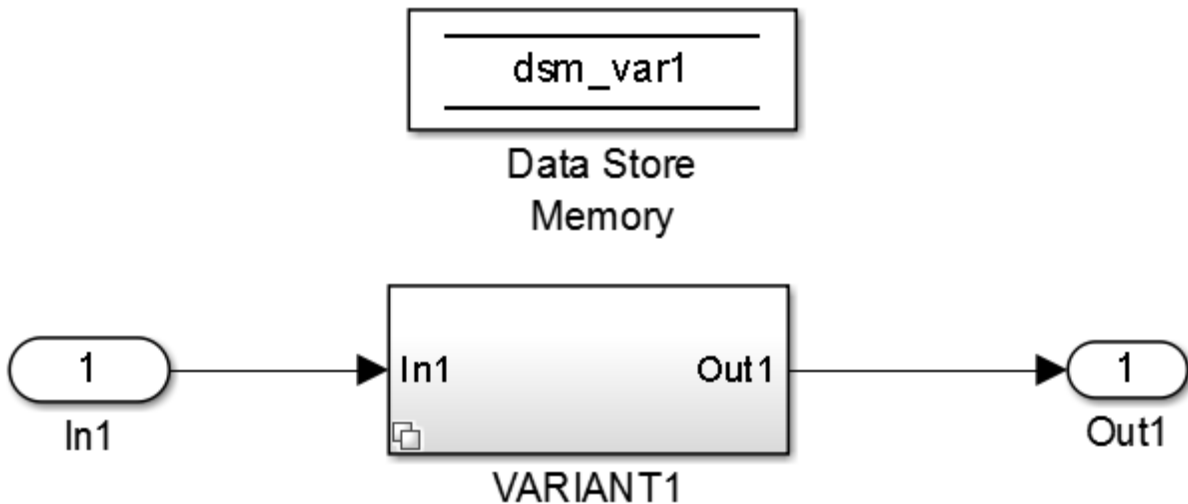
The Filter Design and Analysis Tool option to target the Code Composer Studio™ IDE will be removed in a future release. The Filter Design and Analysis Tool is available with Signal Processing Toolbox™.

Performance

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

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

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 2-35.)

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

- “Support for AUTOSAR Release 4.1” on page 2-23
- “AUTOSAR client and server modeling” on page 2-17
- “Multi-instance AUTOSAR atomic software components” on page 2-24
- Code Replacement for AUTOSAR
- “Support Package for AUTOSAR Standard” on page 2-22
- “AUTOSAR help navigation enhancements” on page 2-22

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.

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.

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

- “Example Model” on page 2-4
- “In-place assignments for select-assign-iterate pattern” on page 2-6
- “Subsystem signal information” on page 2-7
- “Variable reuse around call site” on page 2-8

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:



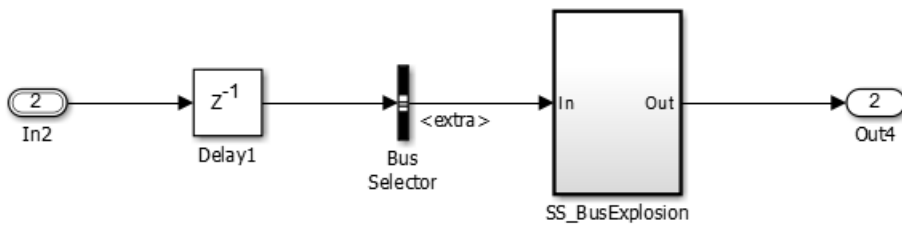
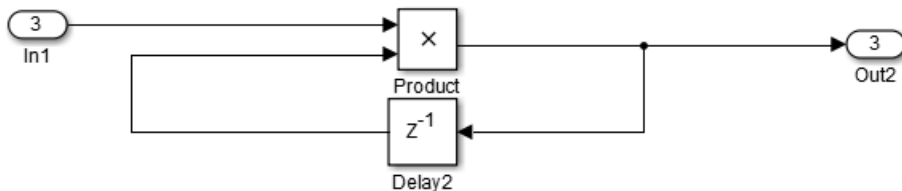
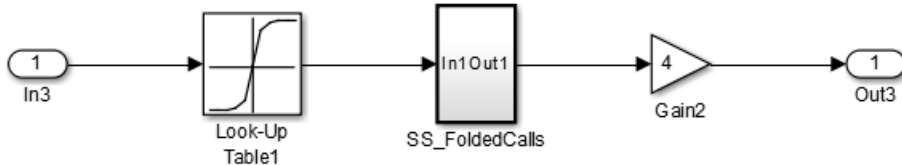
SS_InPlaceSCAssign



Data Store
Memory



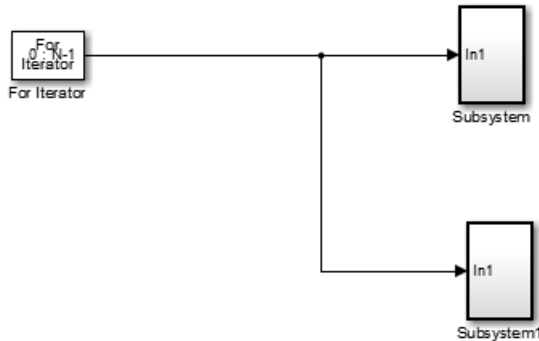
Data Store
Memory1



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' */
}

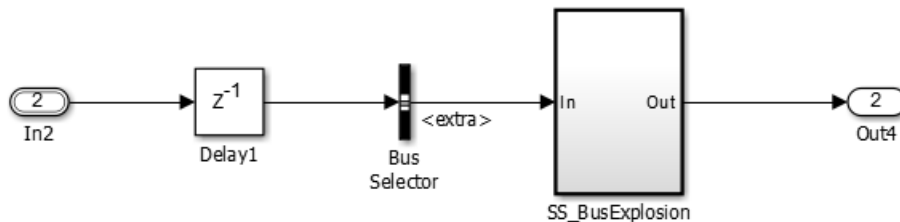
```

The code generator produces less code, does not use any 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' */

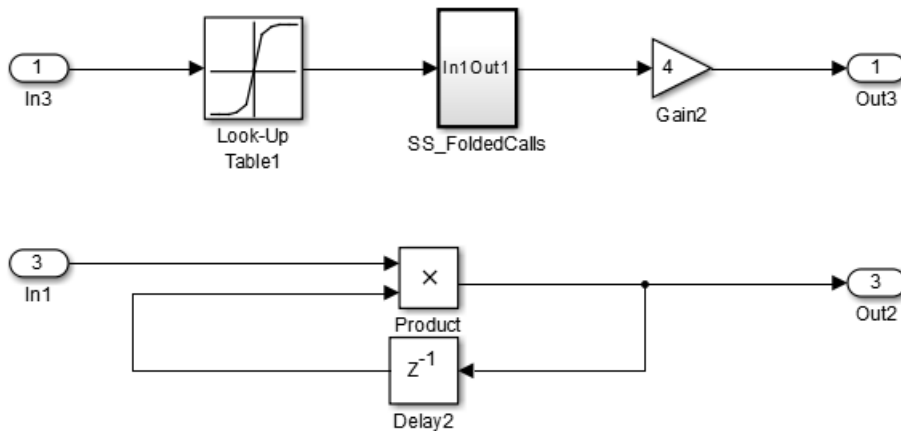
```

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.

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](#).

Code Generation from MATLAB Code

Software-in-the-loop verification improvements for MATLAB Coder

The following table lists software-in-the-loop (SIL) execution improvements.

Feature		R2014b support	Previous support
Code debugging during SIL execution		Linux [®] : GNU [®] Data Display Debugger (DDD) Windows: Microsoft Visual Studio debugger	Windows: Microsoft Visual Studio debugger
Interface types	Multiple entry points	Yes	No
Size	Static variable-size arrays	Yes	Limited to function arguments that were fixed-size structures with variable-size fields.

For more information, see:

- Code Debugging During SIL Execution
- SIL/PIL Execution Support and Limitations

Additional options for custom banners and comments in C and C++ code generated from MATLAB code

In a code generation template (CGT) file, you can now specify the following:

- Custom banners for shared utility functions
- Custom comments before individual code sections such as `Include Files` and `Function Declarations`
- doxygen style comments

The `style` attribute options for doxygen style comments are `doxygen` and `doxygen_qt`. The `TargetLang` and `CommentStyle` code configuration object

properties determine the use of C or C++ style comments with the doxygen style comments.

doxygen with C style comments

```
/**  
 * multiple line comments  
 * second line  
 */
```

doxygen with C++ style comments

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

doxygen_qt with C style comments

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

doxygen_qt with C++ style comments

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

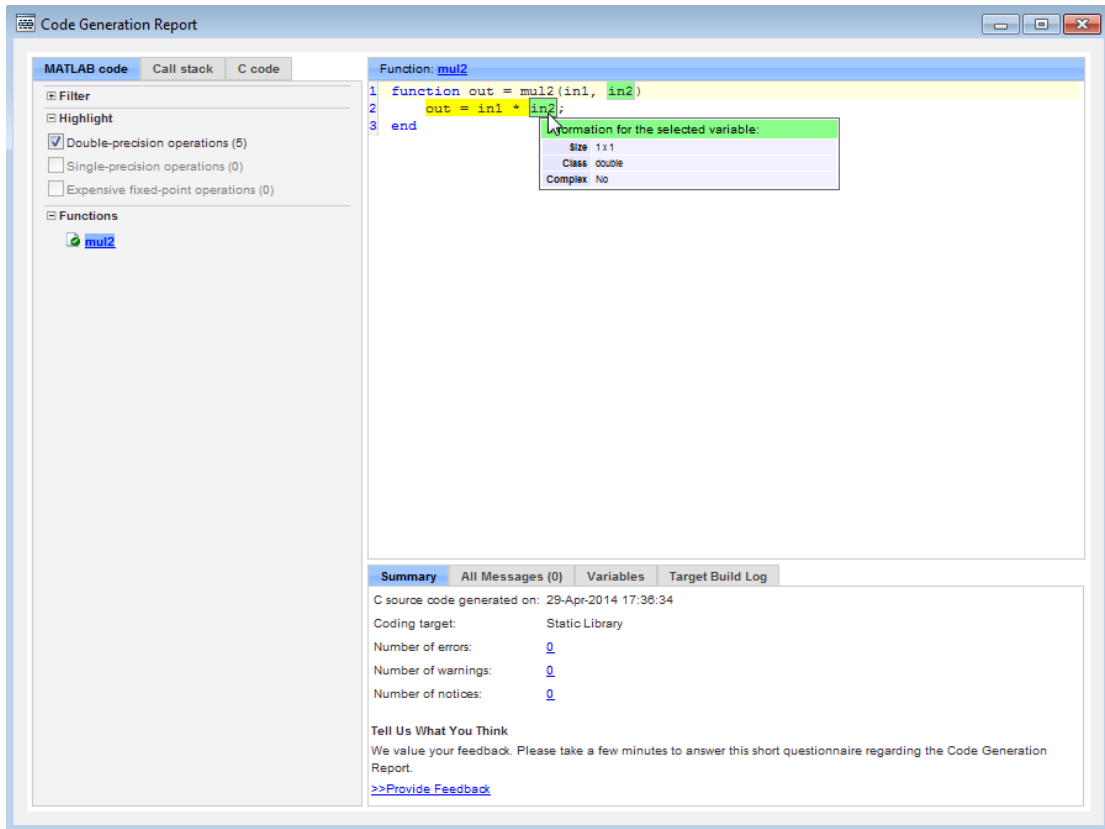
See Code Generation Template (CGT) Files for MATLAB.

Highlighting of potential data type issues in code generation reports

When you generate standalone code from MATLAB code, you now have the option to highlight potential data type issues in the code generation report. The report highlights MATLAB code that results in single-precision and double-precision operations in the generated C/C++ code. If you have a Fixed-Point Designer license, the report also highlights expressions in the MATLAB code that result in expensive fixed-point operations in the generated code. The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB

code that result in cumbersome multiplication and division, and expensive rounding in generated C/C++ code.

The following example report highlights MATLAB code that results in double-precision operations in the generated C code.



The checks are disabled by default.

To enable the checks in a project, on the **Debugging** tab, select the **Always create a code generation report** and **Highlight potential data types issues** check boxes.

To enable the checks at the command line:

- 1 Create a configuration object to generate standalone C/C++ code for an embedded target. For example:

```
cfg = coder.config('lib','ecoder',true);
```

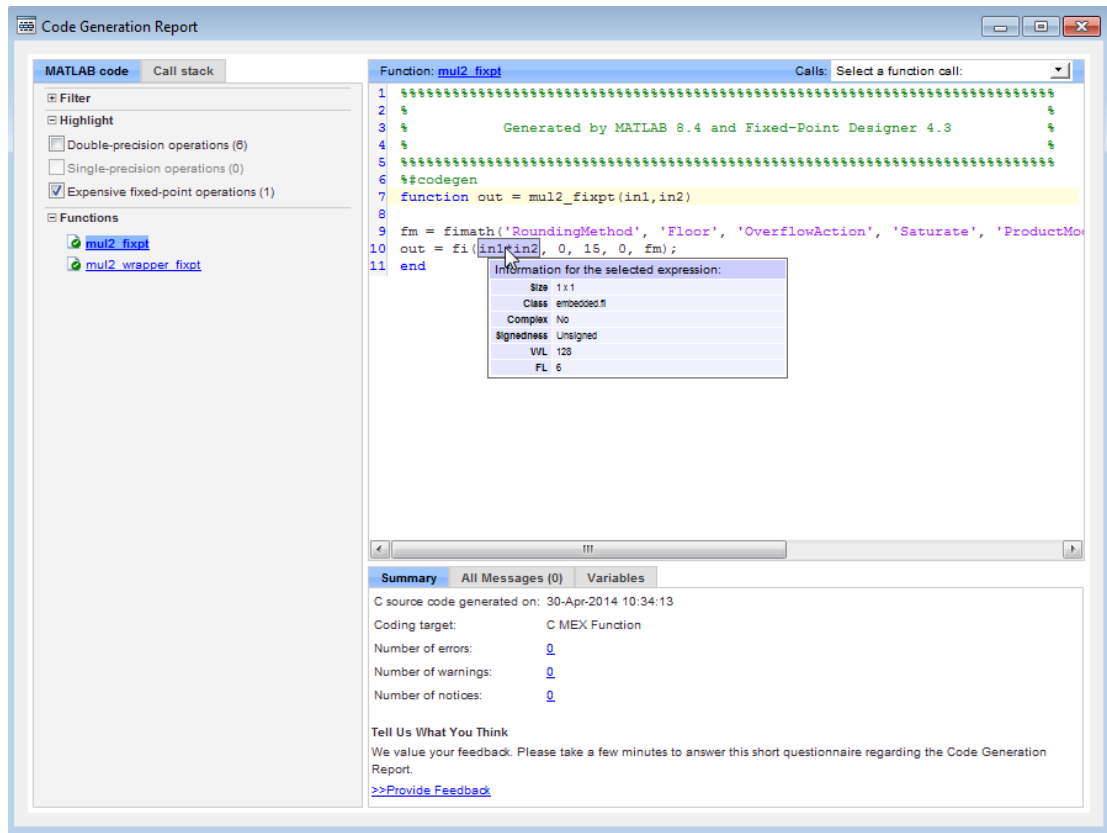
- 2 Set the `HighlightPotentialDataTypeIssues` property to `true`:

```
cfg.HighlightPotentialDataTypeIssues = true;
```

See [Highlight Potential Data Type Issues in a Report and Find Potential Data Type Issues in Generated Code](#).

If you have a Fixed-Point Designer license, you have the option to highlight potential data type issues in the generated HTML report that is available after the fixed-point type validation step of the fixed-point conversion process. An Embedded Coder license is not required to highlight potential data types issues in this report. The report highlights MATLAB code that requires single-precision, double-precision, or expensive fixed-point operations.

The following example report highlights MATLAB code that requires expensive fixed-point operations.



The checks are disabled by default. To enable the checks in a project:

- 1 In the Fixed-Point Conversion Tool, click **Advanced** to view the advanced settings.
- 2 Set **Highlight potential data type issues** to **Yes**.

To enable the checks at the command line:

- 1 Create a floating-point to fixed-point conversion configuration object:


```
fxptcfg = coder.config('fixpt');
```
- 2 Set the `HighlightPotentialDataTypeIssues` property to `true`.


```
fxptcfg.HighlightPotentialDataTypeIssues = true;
```

See Data Type Issues in Generated Code.

Model Architecture and Design

AUTOSAR 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.

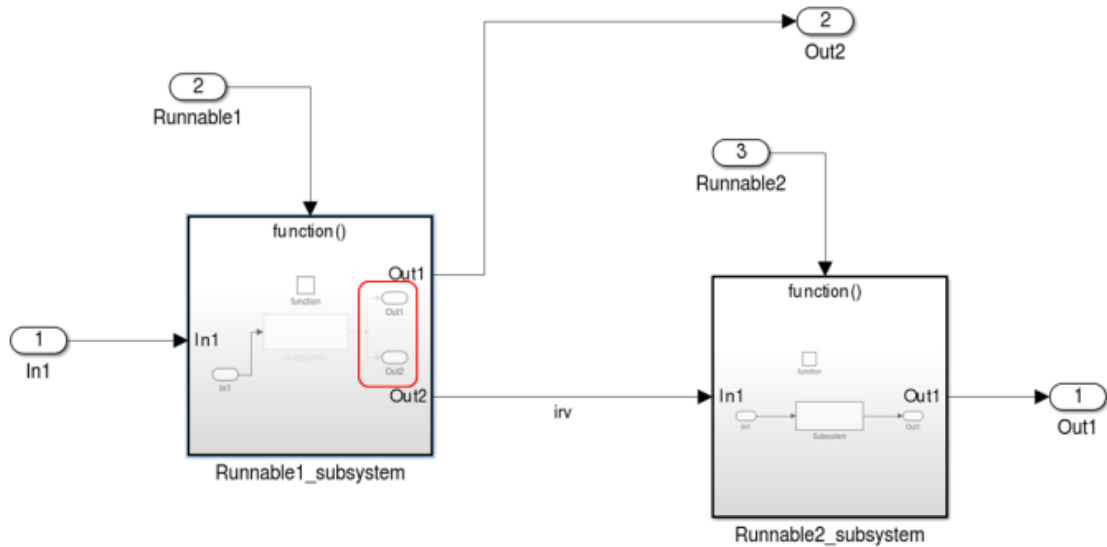
For more information, see [Client-Server Interface and Configure AUTOSAR Client-Server Communication](#).

Global From and Goto blocks for AUTOSAR modeling

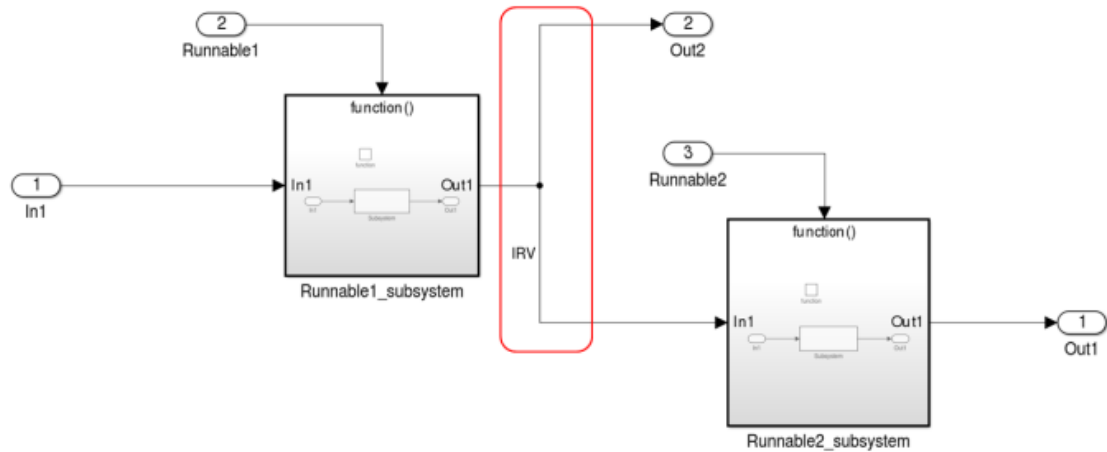
Beginning in R2014b, you can use global From and Goto blocks in a model configured for AUTOSAR. With From and Goto blocks, you can pass a signal from one block to another without actually connecting them. You can model AUTOSAR runnables with more flexibility and cleaner separation of components and interfaces.

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 interrunnable 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 output outside a runnable boundary.
- When a runnable output signal branches to an IRV and a root output outside the runnable subsystem:
 - Only Goto and From blocks are allowed between the source and the destination of the signal.
 - You cannot conditionally write to the IRV or root output.
- When a runnable output signal does not branch, only Goto/From and Merge blocks are allowed between the source and the destination of the signal.

Data, Function, and File Definition

Constant sample time limitation for AUTOSAR models

Previously, for models using the AUTOSAR target, the compiler reported a warning if you configured a root-level 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

Support Package for AUTOSAR Standard

Beginning in R2014b, Embedded Coder software provides add-on support for the AUTOSAR standard via the Embedded Coder Support Package for AUTOSAR Standard. With the support package installed, you can create and modify an AUTOSAR configuration for a model, model AUTOSAR elements, and generate ARXML and AUTOSAR-compatible C code from a model.

To download and install the support package,

- 1 On the MATLAB Toolstrip, click **Add-Ons > Get Hardware Support Packages**.
- 2 Select **Install from Internet** and click **Next**.
- 3 From the list of available support packages, select **AUTOSAR Standard**.
- 4 To complete the installation, follow the instructions provided by Support Package Installer.

For more information, see Support Package Installation.

Compatibility Considerations

AUTOSAR models and scripts that worked without a support package before R2014b now require Embedded Coder Support Package for AUTOSAR Standard. Install the support package before working with AUTOSAR models and scripts.

AUTOSAR help navigation enhancements

To make it easier to find AUTOSAR topics within MATLAB documentation, R2014b introduces the following AUTOSAR documentation enhancements:

- New AUTOSAR landing page in MATLAB Help — Encapsulates the entire Embedded Coder AUTOSAR workflow.

AUTOSAR

Develop AUTOSAR software components for automotive systems

[Modeling Patterns for AUTOSAR](#)

Modeling AUTOSAR Software Components for simulation and code generation

[AUTOSAR Component Creation](#)

Create Simulink® model representation of AUTOSAR software component

[AUTOSAR Component Development](#)

Develop and validate AUTOSAR software component

[AUTOSAR Code Generation](#)

Export component XML description and C code for AUTOSAR run-time environment

- New *Embedded Coder AUTOSAR* book in PDF format — Collects AUTOSAR concepts, examples, how-to topics, and reference material in a PDF file to help Simulink users learn how to model AUTOSAR components.

PDF Documentation for Embedded Coder

[Embedded Coder Getting Started Guide](#)

[Embedded Coder User's Guide](#)

[Embedded Coder AUTOSAR](#)

[Embedded Coder Reference](#)

[Embedded Coder Release Notes](#)

Support for AUTOSAR Release 4.1

AUTOSAR 4.1 ARXML and C code generation

The software now supports AUTOSAR Release 4.1 (schema version 4.1.1) for import and export of arxml files and generation of AUTOSAR-compatible C code.

If you import schema version 4.1.1 arxml code into Simulink, the arxml importer detects and uses the schema version, and sets the schema version parameter in the model to 4.1.

For information on specifying an AUTOSAR schema version for code generation, see [Select an AUTOSAR Schema](#).

AUTOSAR 4.1 InitEvent support

Beginning in R2014b, you can model AUTOSAR initialization events (**InitEvents**), as defined in AUTOSAR schema version 4.1. You can use an **InitEvent** to designate an AUTOSAR runnable as an initialization runnable, and then map an initialization function to the runnable.

In previous releases, you could use AUTOSAR mode management to set up software component initialization. For example, you could define a **ModeDeclarationGroup** with a mode for setting up and initializing a software component. **InitEvent** provides a potentially lighter-weight alternative to the mode-based approach.

If you import **arxml** code that describes a runnable with an **InitEvent**, the **arxml** importer configures the runnable in Simulink as an initialization runnable.

Alternatively, you can configure a runnable to be the initialization runnable in Simulink. For more information, see [Configure AUTOSAR Initialization Runnable](#).

AUTOSAR 4.1 provide-require port support

Beginning in R2014b, you can model AUTOSAR provide-require ports (**PRPorts**), as defined in AUTOSAR schema version 4.1. **PRPorts** are a third type of port, in addition to provide ports (**PPorts**) and require ports (**RPorts**), that can be associated with an AUTOSAR sender-receiver interface. For example, you can:

- Map a Simulink inport/outport pair to a data element of an AUTOSAR provide require port. Generated code complies with Simulink and AUTOSAR semantics.
- Import AUTOSAR provide-require ports for sender-receiver interfaces from ARXML files.
- Export AUTOSAR provide-require ports to ARXML files.

For more information, see [Configure AUTOSAR Provide-Require Port](#).

Multi-instance AUTOSAR atomic software components

In previous releases, AUTOSAR software components (SWCs) modeled in Simulink were single-instance. Beginning in R2014b, you can model multi-instance AUTOSAR SWCs in Simulink. For example, you can:

- Map and configure a Simulink model as a multi-instance AUTOSAR SWC, and validate the configuration.
- Generate C code with reentrant runnable functions and multi-instance RTE API calls.

- Verify AUTOSAR multi-instance C code with SIL and PIL simulations.
- Import and export multi-instance AUTOSAR SWC description XML files.

For more information and limitations, see Multi-Instance Atomic Software Components.

AUTOSAR arxml import and export

AUTOSAR R4.x compliant data type support

AUTOSAR data types workflow improvements

R2014b provides enhanced AUTOSAR Release 4.x compliant data type support.

- For round-trip workflows involving AUTOSAR components originated outside MATLAB, the `arxml` importer and exporter preserve data type information and mapping for each imported AUTOSAR data type.
- For AUTOSAR components originated in Simulink, the software generates AUTOSAR application, implementation, and base types to preserve the information contained within Simulink data types.

For more information, see Release 4.x Data Types.

Application data type export control

For AUTOSAR data types created in Simulink, by default, the software generates application base types only for fixed-point data types and enumerated data types with storage types.

Beginning in R2014b, if you want to override the default behavior for generating application types, you can configure the `arxml` exporter to generate an application type, along with the implementation type and base type, for each exported AUTOSAR data type. For more information, see Control Application Data Type Generation.

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:

Additional Packages	
ApplicationDataType Package:	<input type="text"/>
SwBaseType Package:	<input type="text"/>
DataTypeMappingSet Package:	<input type="text" value="/pkg/dt/DataTypeMappings"/>

The exported `arxml` uses the specified package. The default mapping set short-name is the component name `ASWC` prefixed to `DataTypeMappingsSet`. You can specify a short name for a data type mapping set using the AUTOSAR property function `addPackageableElement`.

For more information, see `Configure 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 (`ApplicationValueSpecifications`).

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

- The `arxml` importer uses `ApplicationValueSpecifications` found in imported `arxml` files to initialize the corresponding data objects in the Simulink model.
- If you select AUTOSAR schema 4.0 or later for a model that contains AUTOSAR data typed by application data type, code generation exports `arxml` code that uses `ApplicationValueSpecifications` to specify initial values for AUTOSAR data.

AUTOSAR CompuMethod control

CompuMethod direction for linear functions

In previous releases, Embedded Coder software imported AUTOSAR computational methods (`CompuMethods`) described in `arxml` code and preserved them across round-

trips between an AUTOSAR authoring tool (AAT) and Simulink. For designs originated in Simulink, the arxml exporter created schema-compliant `CompuMethods`, but did not allow users control over `CompuMethod` attributes, including the direction of `CompuMethod` conversion between internal and physical representations of a value. For `CompuMethods` originated in Simulink, the exporter generated only the forward, internal-to-physical direction.

Beginning in R2014b, you can control how conversion direction is described in exported `CompuMethods`. Using either the Configure AUTOSAR Interface dialog box or the AUTOSAR property `set` function, you can specify one of the following `CompuMethod` direction values:

- `InternalToPhys` (default) — Generate `CompuMethod` sections for conversion of internal values into their physical representations.
- `PhysToInternal` — Generate `CompuMethod` sections for conversion of physical values into their internal representations.
- `Bidirectional` — Generate `CompuMethod` sections for both internal-to-physical and physical-to-internal conversion directions.

For more information, see `CompuMethod` Direction for Linear Functions.

CompuMethod generated for each ApplicationDataType

In previous releases, the arxml exporter preserved AUTOSAR computational methods (`CompuMethods`) that you imported into Simulink, but for designs originated in Simulink, generated `CompuMethods` only for fixed point application types.

Beginning in R2014b, the exporter generates `CompuMethods` for every primitive application type. Measurement and calibration tools can monitor and interact with more application data. For more information, see `CompuMethod` Categories for Data Types.

Unit reference generated for each CompuMethod

In previous releases, exported `CompuMethods` did not contain unit references. Beginning in R2014b:

- The arxml importer preserves unit and physical dimension information found in imported `CompuMethods`. The software preserves `CompuMethod` unit and physical dimension information across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.
- For designs originated in Simulink, the exporter generates a unit reference for each `CompuMethod`.

Providing a unit for each exported `CompuMethod` helps support measurement and calibration tool use of exported AUTOSAR data. For more information, see [CompuMethod Unit References](#).

Rational function `CompuMethod` for dual-scaled parameter

R2014b provides greater control over the AUTOSAR `CompuMethods` generated for AUTOSAR dual-scaled parameters. For an AUTOSAR dual-scaled parameter, which stores two scaled values of the same physical value, the software generates the `CompuMethod` category `RAT_FUNC`. The computation method can be a first-order rational function. For more information, see [Rational Function `CompuMethod` for Dual-Scaled Parameter](#).

Improved AUTOSAR package configuration

In previous releases, the `arxml` exporter generated a fixed file and package structure for packaging AUTOSAR elements. Beginning in R2014b, Embedded Coder software provides more flexible configuration and management of AUTOSAR packages. For example:

- AUTOSAR packages and their elements now are fully preserved across round-trips between an AUTOSAR authoring tool (AAT) and Simulink.
- AUTOSAR XML options in Simulink include ten new packaging parameters (`XmlOptions` properties). You can now easily group AUTOSAR elements of the following categories into packages:
 - Application data types (schema 4.x)
 - Software base types (schema 4.x)
 - Data type mapping sets (schema 4.x)
 - Constants and values
 - Physical data constraints (referenced by application data types or data prototypes)
 - System constants (schema 4.x)
 - Software address methods
 - Mode declaration groups
 - Computational methods
 - Units and unit groups (schema 4.x)

For more information, see [Configure AUTOSAR Package Structure](#).

AUTOSAR calibration component export

In previous releases, the software exported an AUTOSAR calibration component (`ParameterSwComponent`) only if it had been created in an AUTOSAR authoring tool (AAT) and imported into Simulink from an `arxml` file.

Beginning in R2014b, the software can export an AUTOSAR calibration component originated in Simulink. To configure AUTOSAR parameters for export in a calibration component, use the custom storage class (CSC) `Ca1Prm` 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 2-25.

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	UseOperatorNewForModelRefRegistrati

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 can no longer select these libraries:

- Intel IPP
- Intel IPP/SSE with GNU99 extensions

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

See [Choose a Code Replacement Library](#).

Deployment

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 2-34
- “Replacement of shift and cast operations involving vector and matrix operands” on page 2-35

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 Altera[®] SoC Platform 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

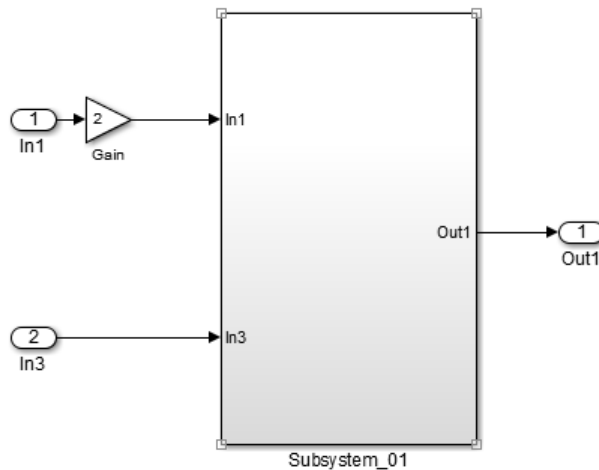
Support for Texas Instruments C5000™ DSPs will be removed in a future release.

Performance

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.

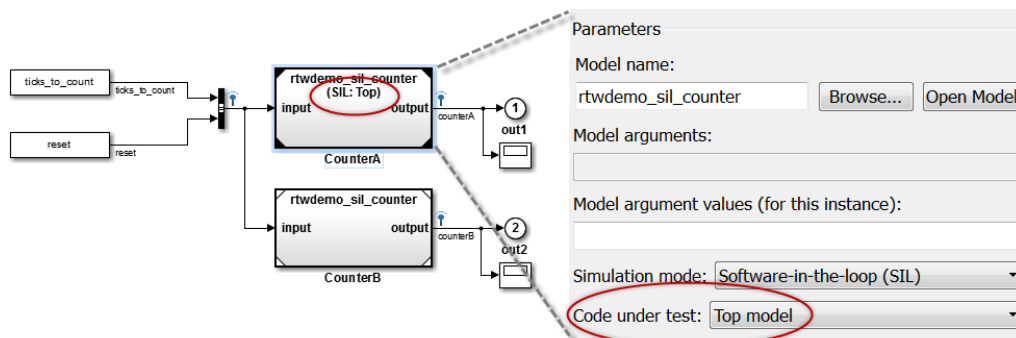
```
Gain[0] = 2.0 * exlocal_U.In1[0];  
Gain[1] = 2.0 * exlocal_U.In1[1];  
Gain[2] = 2.0 * exlocal_U.In1[2];  
Gain[3] = 2.0 * exlocal_U.In1[3];  
  
exlocal_Subsystem_03(Gain, exlocal_U.In3, &exlocal_Y.Out1);
```

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

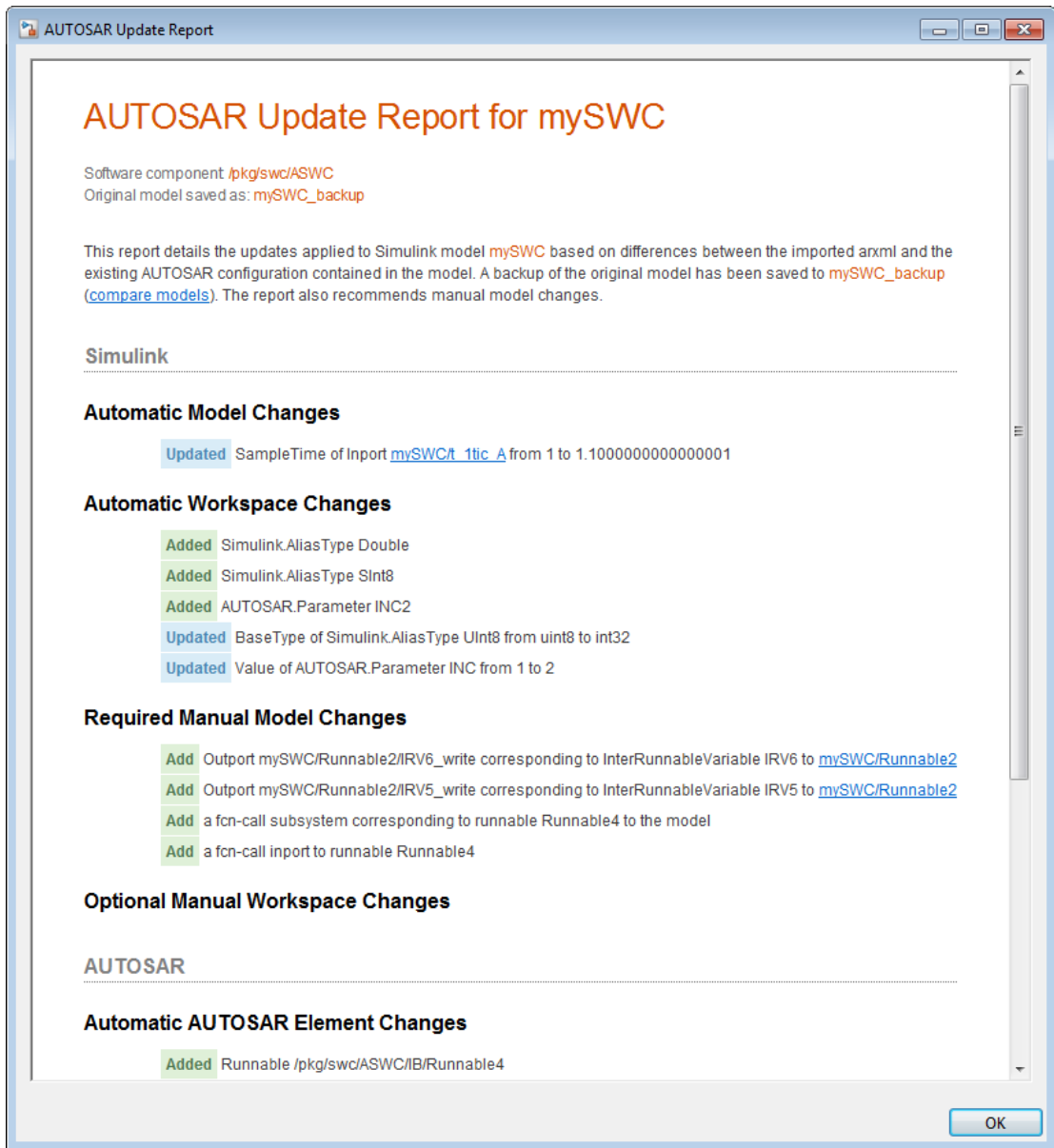
Compatibility Considerations

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](#).



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](#).

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=f00(x)`.

For more information, see [Specify In-Place Code Replacement](#) and `coder.replace`.

Lookup table code replacement for Simulink

In R2014a, you can replace these lookup table functions during code generation for Simulink models.

<code>interp1D</code>	<code>interp4D</code>	<code>lookup2D</code>	<code>lookup5D</code>
<code>interp2D</code>	<code>interp5D</code>	<code>lookup3D</code>	<code>lookupND_Direct</code>
<code>interp3D</code>	<code>lookup1D</code>	<code>lookup4D</code>	<code>prelookup</code>

When you create a replacement table entry for one these functions, you must specify a set of algorithm properties in addition to the usual code replacement function key, conceptual arguments, and other applicable math mode information. Specify the algorithm properties by using new algorithm property fields in the code replacement tool or the new `addAlgorithmProperty` function. Conceptual arguments and algorithm parameters must match for replacement to occur.

For more information, see [Map Lookup Table Functions to Application Implementations](#).

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

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.

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](#).

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](#).

Code Generation from MATLAB Code

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=f00(x)`.

For more information, see [Specify In-Place Code Replacement](#) and `coder.replace`.

Single-line (`//`) comment style available for generated code

In earlier releases, C and C++ code generation always used a multi-line (`/*...*/`) comment style. Beginning in R2014a, you can select a single-line (`//...`) comment style for generated code.

Set the comment style in one of the following ways:

- In a project, in the Project Settings dialog box **Code Appearance** tab, set **Comment Style** to one of the following values.

Value	Description
Auto(Use standard comment style of the target language)	For C, generate multi-line comments. For C++, generate single-line comments. (default)
Single-line (Use C++-style comments)	Generate single-line comments preceded by <code>//</code> .
Multi-line (Use C-style comments)	Generate single or multi-line comments delimited by <code>/*</code> and <code>*/</code> .

- At the command prompt, create a code generation configuration object. Set the `CommentStyle` parameter to one of the following values.

Value	Description
'Auto'	For C, generate multi-line comments. For C++, generate single-line comments. (default)

Value	Description
'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.

Feature		R2014a support	Previous support
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

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.

`rtwdemo_autosar_multirunnables.arxml`

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

`rtwdemo_autosar_multirunnables.c`

```
/* Output function for RootInportFunctionCallGenerator:
   '<Root>/RootFcnCall_InsertedFor_Runnable1_at_output_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

Constant sample time limitations for root-level Output blocks

In R2014a, the sample time of root-level Output blocks is checked in the following ways:

- For models using Function Prototype Control or a C++ class interface, the validation check reports an error if a root-level Output block has a constant sample time.
- For models using the AUTOSAR target, the compiler reports a warning if a root-level Output block is configured to inherit a constant sample time from its sources. The compiler then sets the sample time of the root-level Output block to the fundamental rate of the model. This warning will become an error in a future release.
- When importing an AUTOSAR model from an XML description of a single runnable, the import tool sets the sample time of root-level Output blocks to the fundamental rate of the model.
- The Upgrade Advisor adds a check identifying root-level Output blocks with a constant sample time. If a model uses the AUTOSAR target, Function Prototype Control, or a C++ class interface, the check lists the Output blocks with a constant sample time. The check also includes possible actions to fix the blocks.

Example model `rtwdemo_cppencap` renamed to `rtwdemo_cppclass`

As part of the C++ class code interface packaging changes described in Simulink Coder release note Improved control of C and C++ code interface packaging, C++ class example model `rtwdemo_cppencap` has been renamed to `rtwdemo_cppclass`.

Unit Delay block optimization

In R2014a, when you specify a nonzero initial value or a global storage class, global block output reuse might eliminate the Unit Delay state in the generated code. Eliminating the Unit Delay state reduces data copies.

Code Generation

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

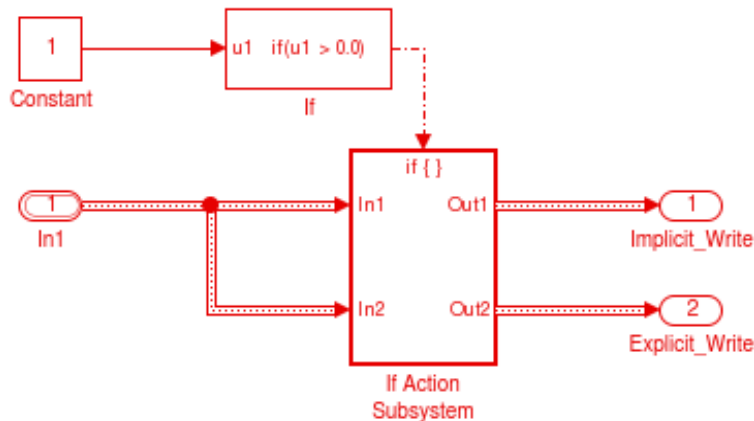
Previously, when specifying a namespace for a model class, the generated namespace code might be incorrectly indented if you selected K&R for the **Indent style** on the **Code Generation > Code Style** pane. In R2014a, the generated namespace code follows coding standards when you select the K&R style.

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());
  }
  /* Output: '<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

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

Enhanced global variable optimization options

In R2014a, you can choose a global variable reference optimization for the generated code.

In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, the Optimize global data access drop-down list provides the following options:

- None

Use default optimizations.

- Use global to hold temporary results

Maximize use of global variables.

- Minimize global data access

Minimize use of global variables by using local variables to hold intermediate values.

With an Embedded Coder license, if you select an embedded target such as `ert.tlc`, the software replaces the **Minimize data copies between local and global variables** check box with the **Optimize global data access** list. When **Minimize data copies between local and global variables** is selected, **Optimize global data access** is set to Use global to hold temporary results.

For more information, see Optimize Global Variable Usage.

for loops used to initialize arrays to zero

For signals with custom storage, code generation creates a `for` loop to initialize an array with matching values, such as all zeroes or ones. This initialization method reduces code size, especially for larger arrays. Previously, the generated code initialized each element individually on a separate line.

Verification

Software-in-the-loop simulation for physical models

You can run software-in-the-loop (SIL) simulations of models that use Simscape™ blocks.

SIL verification for subsystem code generation

You can use the SIL block approach to verify code generated from top-models and subsystems. In R2014a, SIL block verification supports the following features:

- Profiling of task and function execution times.
- Source-level debugging with the Microsoft Visual C++® debugger.

Compatibility Considerations

The table describes SIL block verification features that differ from the previous release. If you want to revert to previous SIL block behavior, in the Command Window, run:

```
silblocktype('legacy');
```

To restore R2014a SIL block behavior, run:

```
silblocktype('unified');
```

Feature	R2014a Details
Validation checks	<p>The software performs, with reference to your host computer architecture, stricter checks on active Hardware Implementation settings. If the software detects mismatches, the software generates errors.</p> <p>For example, if your host computer is a 64-bit Linux machine, you cannot specify the following combination of settings:</p> <ul style="list-style-type: none"> • Device vendor: Generic • Device type: 32-bit x86 compatible <p>To resolve the mismatch errors, do one of the following:</p>

Feature	R2014a Details
	<ul style="list-style-type: none"> • In the Configuration Parameters > Code Generation > Verification pane, select the Enable portable word sizes check box. • In the Configuration Parameters > Hardware Implementation pane, through the Production hardware or Test hardware section, specify settings that correspond to your host computer architecture. <p>The software generates an error if:</p> <ul style="list-style-type: none"> • The generated code for the component under test has been updated since the creation of the SIL block. • The MATLAB version has changed since the creation of the SIL block.
GenerateErtSFunction parameter	<p>The GenerateErtSFunction parameter has the following command-line behavior:</p> <ul style="list-style-type: none"> • <code>set_param(model, 'GenerateErtSFunction', 'on')</code> generates a warning that the parameter will be removed in a future release. As a replacement, use the command <code>set_param(model, 'CreateSILPILBlock', 'SIL')</code>. • <code>set_param(model, 'GenerateErtSFunction', 'off')</code> does not change the parameter. As a replacement, use the command <code>set_param(model, 'CreateSILPILBlock', 'None')</code>. • <code>get_param(model, 'GenerateErtSFunction')</code> returns the value <code>off</code>. As a replacement, use the command <code>get_param(model, 'CreateSILPILBlock')</code>.
Parameter tuning	<p>During a SIL block simulation, the software does not support the tuning of block dialog box parameters. Through the SIL block dialog box, you can view the list of tunable global parameters</p>

Feature	R2014a Details
	<p>The software does not support SIL block creation if all of the following apply:</p> <ul style="list-style-type: none"> • Code Generation > Interface > Code interface packaging is set to Reusable function. • Optimization > Signals and Parameters > Inline parameters is not selected. • The model contains parameters with storage class Auto or SimulinkGlobal.
Data definition and initialization	<p>In the SIL test harness, for signals that are internal with respect to the SIL block or models referenced by the SIL block, the software does not automatically define and initialize signals with imported storage classes.</p>
	<p>The software does not support automatic data definition and data transfer for local data stores in the SIL block.</p>
C++ class code (previously called C++ encapsulated code)	<p>For C++ class code:</p> <ul style="list-style-type: none"> • You must set External I/O access parameter to None. • Parameters are not tunable if Block parameter visibility is private and Block parameter access is either Method or Inlined method. <p>You can specify these settings through the Code Generation > Interface pane.</p>
Code generation report	<p>The code generation report does not display test harness files for your SIL block.</p>
Multiword fixed-point data	<p>At the SIL block interface, the software does not support multiword fixed-point data types. The software supports:</p> <ul style="list-style-type: none"> • At the block interface, single word data types that are wider than 32 bits. • Within the SIL block, multiword fixed-point data types.
Boolean data type replacement	<p>At the SIL block boundary, the software does not support the replacement of the boolean data type by integers.</p>

Feature	R2014a Details
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 `mxAarray`) utility functions. When you generate code that uses variable-size data, the code generation software exports utility functions to interact with `emxArray` data structures. Customize utility function identifiers to avoid name collisions when a function that uses variable-size data calls a library function that uses variable-size data.

To customize generated identifiers for `emxArray` utility functions:

- In a project

On the Project Settings dialog box **Code Appearance** tab, under **Identifier Format**, in the **EMX Array Utility Functions** field, enter the identifier format. For example, `'myemxMN'`.

- At the command line

Create a code generation configuration object and set the `CustomSymbolStrEMXArrayFcn` parameter to the identifier format. For example:

```
cfg = coder.config('lib');  
cfg.CustomSymbolStrEMXArrayFcn='myemx$M$N';
```

For details about the identifier format, see `coder.EmbeddedCodeConfig`.

Model Architecture and Design

Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior

R2013b enhances AUTOSAR modeling, component import, and programmatic control. See also “Support for AUTOSAR release 4.0.3 XML and generated code”.

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 import 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 interrunnable variables (IRVs) imported from the AUTOSAR software component.

Ability to model AUTOSAR mode receiver ports and events

R2013b provides the ability to model AUTOSAR mode receiver ports and mode-switch events in Simulink. Specifically, you can:

- Model the mode receiver port for an AUTOSAR software component using a Simulink inport.
- Specify a mode-switch event to trigger an initialize function runnable or an exported function-call subsystem runnable.

For more information, see [Configure AUTOSAR Mode Receiver Ports and Events](#).

AUTOSAR dual-scaled parameter

The new `AUTOSAR.DualScaledParameter` class extends the capabilities of the `AUTOSAR.Parameter` class. You can define a parameter object that stores two scaled values of the same physical value. Suppose you want to store temperature measurements as Fahrenheit or Celsius values. You can define a parameter that stores the temperature in either measurement scale with a computational method to convert between the dual-scaled values.

You can use `AUTOSAR.DualScaledParameter` objects in your model for both simulation and code generation. The parameter computes the internal value before simulation or code generation via a computational method, which can be a first-order rational function. This offline computation results in leaner generated code.

Embedded Coder also generates an XML file for use by a calibration tool. This file contains the dual-scaled values and the corresponding computational method.

For more information, see [AUTOSAR.DualScaledParameter](#).

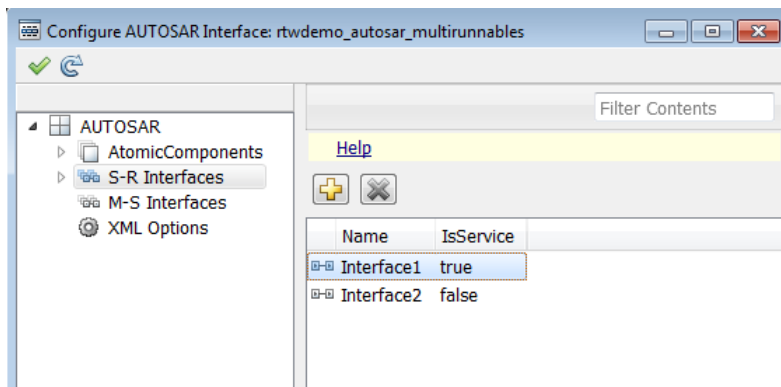
Programmatic interface for configuring AUTOSAR properties and Simulink-AUTOSAR mapping

R2013b provides a programmatic interface for configuring AUTOSAR properties and Simulink mapping information using MATLAB functions. You can programmatically get, set, add, and remove the same component properties and mapping information displayed

in the **AUTOSAR Properties Explorer** and **Simulink-AUTOSAR Mapping Explorer** views of the Configure AUTOSAR Interface dialog box.

In the function syntax, you can use fully or partially qualified names to locate properties. For example, the following code sets the `IsService` property for the sender-receiver interface located at path `Interface1` in the example model `rtwdemo_autosar_multirunnables` to `true`. In this case, specifying the name `Interface1` is enough to locate the property.

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> set(propObj, 'Interface1', 'IsService', true);
```



If you added a sender-receiver interface to the component, you would specify a fully qualified path, for example:

```
>> propObj = autosar.api.getAUTOSARProperties('rtwdemo_autosar_multirunnables');
>> addSRInterface(propObj, '/pkg/if/Interface3', 'IsService', true);
```

The new AUTOSAR configuration functions also validate syntax and semantics for requested AUTOSAR property and mapping changes.

Reorganization of Model Advisor Embedded Coder checks

Checks previously provided with Simulink in the Model Advisor Embedded Coder folder are now available with either Simulink Coder or Embedded Coder. For a list of checks available with each product, see:

- Simulink Coder Checks
- Embedded Coder Checks

Model Advisor fixed-point checks with additional coverage and optimization awareness

The Model Advisor fixed-point checks now cover blocks in base Simulink and System Toolboxes, the MATLAB Function block, System objects, Stateflow, and `fi` objects. These improved checks take into consideration model settings such as hardware configuration and code generation settings. These updated checks also avoid false negative results.

For more information, see:

- Identify blocks that generate expensive rounding code
- Identify questionable fixed-point operations
- Identify blocks that generate expensive fixed-point and saturation code

Protected model Web view

In R2013b, a read-only Web view of protected models is now available.

To include the Web view in the protected model, right-click the model reference block, and then select **Subsystem & Model Reference > Create Protected Model for Selected Model Block**. Select the **Open read-only view of model** check box and click **Create**.

To enter a password, right-click the protected model shield icon and select **Authorize**. Enter the password and click **OK**. To show the Web view for a protected model, right-click the shield icon of the protected model and select **Show Web view**.

RTW.AutosarInterface class to be removed in a future release

In R2013b, a new programmatic interface for configuring AUTOSAR properties and mapping information for a Simulink model has replaced the `RTW.AutosarInterface` class used in earlier releases. The `RTW.AutosarInterface` class will be removed in a future release.

Compatibility Considerations

If you are using the `RTW.AutosarInterface` class and methods to programmatically control and validate the AUTOSAR configuration of a model, you should migrate to using

the new AUTOSAR property and mapping functions listed in AUTOSAR Component Development. The new functions are designed to work with the component properties and mapping information displayed in the **AUTOSAR Properties** Explorer and **Simulink-AUTOSAR Mapping** Explorer views of the Configure AUTOSAR Interface dialog box.

Subsystem methods of arxml.importer class to be removed in a future release

Beginning in R2013b, you can model AUTOSAR multi-runnables as function-call subsystems at the top level of a model, rather than as function-call subsystems within a wrapper subsystem that represents the AUTOSAR software component. The following methods of the `arxml.importer` class will be removed in a future release:

- `arxml.importer.createComponentAsSubsystem` — Create AUTOSAR atomic software component as Simulink atomic subsystem
- `arxml.importer.createOperationAsConfigurableSubsystems` — Create configurable Simulink subsystem library for client-server operation

Compatibility Considerations

If you are using `createComponentAsSubsystem` or `createOperationAsConfigurableSubsystems`, you should migrate to using the top model oriented approach described in [Configure Multiple Runnables](#).

Data, Function, and File Definition

Simplified global types file `rtwtypes.h` with invariant content

Previously, during rebuilds of a model hierarchy, the code generation process might have updated the content of the shared header file `rtwtypes.h`. If a model in the hierarchy changed, or the code generator detected a new model in the hierarchy, `rtwtypes.h` could be overwritten. When `rtwtypes.h` changes, you must recompile the code.

In R2013b, the code generator separates some of the `rtwtypes.h` content into separate header files that are generated only when certain model settings or components are present. Separate header files are generated, however, `rtwtypes.h` is unchanged. When certain model settings or components are present, the code generator creates the following new header files.

Model setting or component	Content generated to header file
Multiword data types	<code>multiword_types.h</code>
Model reference target	<code>model_reference_types.h</code>
Model reference blocks	<code>model_reference_types.h</code>
MAT-file logging is selected	<code>builtin_typeid_types.h</code> <code>multiword_types.h</code>
C API	<code>builtin_typeid_types.h</code>
Interface is set to External mode	<code>multiword_types.h</code>

For more information on files created during code generation, see [Files Created During the Build Process](#).

C++ encapsulation support for name space control and template-based file customization

Name space control for scoping C++ encapsulated model classes

R2013b adds name space control for scoping model classes generated using C++ encapsulation. You can use the **Namespace** parameter in the Configure C++ Encapsulation Interface dialog box to specify a name space for a model class. If specified, the name space is emitted in the generated code for the model class. To scope the C++

encapsulated model classes in a model reference hierarchy, you can specify a different name space for each referenced model. For more information, see [Use Name Spaces to Scope C++ Encapsulated Model Classes](#).

For more information on configuring C++ encapsulated model classes, see [Configure C++ Encapsulation Interfaces Using Graphical Interfaces](#).

Template-based customization of encapsulated C++ header and source files

Embedded Coder now supports the **Code Generation > Templates** pane of the Configuration Parameters dialog box for models that use C++ encapsulation. You can use the code and data templates to control the appearance of C++ code in generated model header and source files. For example, you can customize file and function banners to meet organization standards.

However, the following template file features that are supported for other language selections are not supported for C++ encapsulation:

- Free-form text outside template sections
- Custom tokens
- TLC commands (<! > tokens)

Shared utility naming control

You can customize a shared utility name. On the **Code Generation > Symbols** pane enter text and formatting characters in the **Shared utilities** box.

The default token string is \$N\$C.

Token	Description
\$N	The code generator inserts the shared utility function name.
\$C	When the combined text and utility name exceed the maximum identifier length, the code generator inserts an eight-character conditional checksum. This checksum ensures that the name is unique.

If the shared utility identifier exceeds the maximum length, characters are deleted from \$N and the eight-character conditional checksum is inserted.

For more information see

- Shared utilities
- Identifier Format Control
- Exceptions to Identifier Formatting Conventions

Expanded support for identifier names

When specifying temporary local variables, you can now use **\$A** to insert the data type acronym into your variable name. This capability provides you with a more consistent naming scheme.

- You can include **\$A** in naming for local temporary variables where previously it was supported only for local block output variables and field names of global types. For more information, see Identifier Format Control, Local temporary variables and Field name of global types.
- You can customize identifier names by specifying **\$A** which maps to the data type replacement setting. Previously the generated code changed the types, but not the identifier names. For more information, see Data Type Replacement.

Terminate function setting honored for subsystems and referenced models

In previous releases, model builds did not uniformly honor the setting of the model option **Terminate function required** when generating code for subsystems or referenced models. A model build could generate termination code for a subsystem or referenced model when **Terminate function required** was cleared.

Beginning in R2013b, model builds honor the setting of **Terminate function required** for subsystems and referenced models. When **Terminate function required** is cleared, the build suppresses subsystem and referenced model termination code.

Compatibility Considerations

If an existing model relies on subsystem or referenced model termination code being generated despite the model option **Terminate function required** being cleared, consider turning on the **Terminate function required** option.

Code Generation

Support for AUTOSAR release 4.0.3 XML and generated code

R2013b adds AUTOSAR release 4.0.3 support, as follows:

- ARXML import and export support AUTOSAR release 4.0.3 XML files.
- The AUTOSAR target generates AUTOSAR release 4.0.3 compliant C code.
- Selecting the value **4.0** for the AUTOSAR model parameter **Generate XML file** from schema version now selects schema revision 4.0.3, rather than 4.0.2. Also, the parameter now defaults to value **4.0**, rather than **3.0** or an earlier version.

See also “Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior”.

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 specify

- The **Function packaging** parameter for your subsystem to **Nonreusable function**
- The **Function interface** parameter to **Allow arguments**

The code generator might generate a subsystem function that returns a scalar output value. Previously, subsystem functions returned `void`.

Model reference step function void input and output arguments

Since R2010a, when a reusable subsystem fed the output, code generation might create output arguments for model reference step functions.

In R2013b, code generation produces model reference step functions with void input and void output when the model reference block:

- Is a single instance.
- Has exported globals on its input and output ports.

Deployment

ARM Cortex-M optimized code with STM32F4-Discovery board example

Build ARM Cortex-M optimized executables from Simulink models. Automatically run executables on STMicroelectronics STM32F4-Discovery boards.

Note: In addition to the basic math optimizations provided by *Embedded Coder Support Package for ARM Cortex-M Processors*, you can obtain advanced optimizations for ARM DSP filters using the *DSP System Toolbox™ Support Package for ARM Cortex Processors*. For more information, see the DSP System Toolbox release notes for R2013b.

Support package for ARM Cortex processors

Use the *Embedded Coder Support Package for ARM Cortex-M Processors* to:

- Build and run CMSIS-optimized executables on ARM Cortex-M QEMU emulator.
- Use the capabilities and features described in Supported Features for ARM Cortex-M Processors

To download and install this feature, perform the steps described in Install Support for ARM Cortex-M Processors.

For more information, see the Support Package for ARM Cortex-M Processors topic.

Support package for STMicroelectronics STM32F4-Discovery Board

Use the *Embedded Coder Support Package for STMicroelectronics STM32F4 Discovery Board* to automatically build (makefile-based), download, and run an executable on Discovery board processors.

Use blocks from the *Embedded Coder Support Package for STMicroelectronics STM32F4 Discovery Board* block library:

- ADC — Convert analog signal to digital signal.
- GPIO Read — Configure input pin to read pin status.
- GPIO Write — Configure output pin to output pin status.

This support package automatically installs the following third-party software:

- STM32F4DISCOVERY peripheral firmware examples <http://www.st.com/internet/evalboard/product/252419.jsp>
- OpenOCD <http://www.freddiechopin.pl/en/download/category/4-openocd>
- GNU Tools for ARM Embedded Processors <https://launchpad.net/gcc-arm-embedded>
- QEMU <http://lassauge.free.fr/qemu/>
- CMSIS <http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php>

To download and install this support package, perform the steps described in Install Support for STMicroelectronics STM32F4 Discovery Board.

For more information, see the Support Package for STMicroelectronics STM32F4 Discovery Board topic.

Wind River VxWorks 6.9 support

You can automatically generate code from Simulink models and execute it on VxWorks 6.9 RTOS.

To use this feature, install the corresponding support package:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`.
- 2 Use Support Package Installer to install the *Embedded Coder Support Package for Wind River VxWorks RTOS*.

This feature includes the Embedded Coder Support Package for Wind River VxWorks RTOS block library, which contains the following blocks:

- UDP Send and UDP Receive — Enable UDP communication with networked devices using an Ethernet port.
- VxWorks Task — Spawn task function as a separate VxWorks thread.

For more information, see the Support Package for Wind River VxWorks RTOS topic.

Compatibility Considerations

Previous versions of Embedded Coder software had built-in support for the VxWorks 6.7 and 6.8. The current version of Embedded Coder does not have built-in support for VxWorks 6.7 and 6.8. To get support for VxWorks 6.7, 6.8, and 6.9, install the *Embedded Coder Support Package for Wind River VxWorks RTOS*.

Support package for Texas Instruments C2000 processors

You can automatically generate code from Simulink models and execute it on Texas Instruments C2000 processors.

To use this feature, install the corresponding support package:

- 1 In a MATLAB Command Window, enter `supportPackageInstaller`.
- 2 Use Support Package Installer to install *Embedded Coder Support Package for Texas Instruments C2000 Processors*.

This feature includes the Embedded Coder Support Package for Texas Instruments C2000 Processors block library, which contains:

- C2802x (c2802xlib) block library
- C2803x (c2803xlib) block library
- C2806x (c2806xlib) block library
- C280x (c280xlib) block library
- C281x (c281xlib) block library
- C2834x (c2834xlib) block library
- C28x3x (c2833xlib) block library
- Memory Operations block library
- Optimization — C28x DMC (c28xdmclib) block library
- Optimization — C28x IQmath (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.

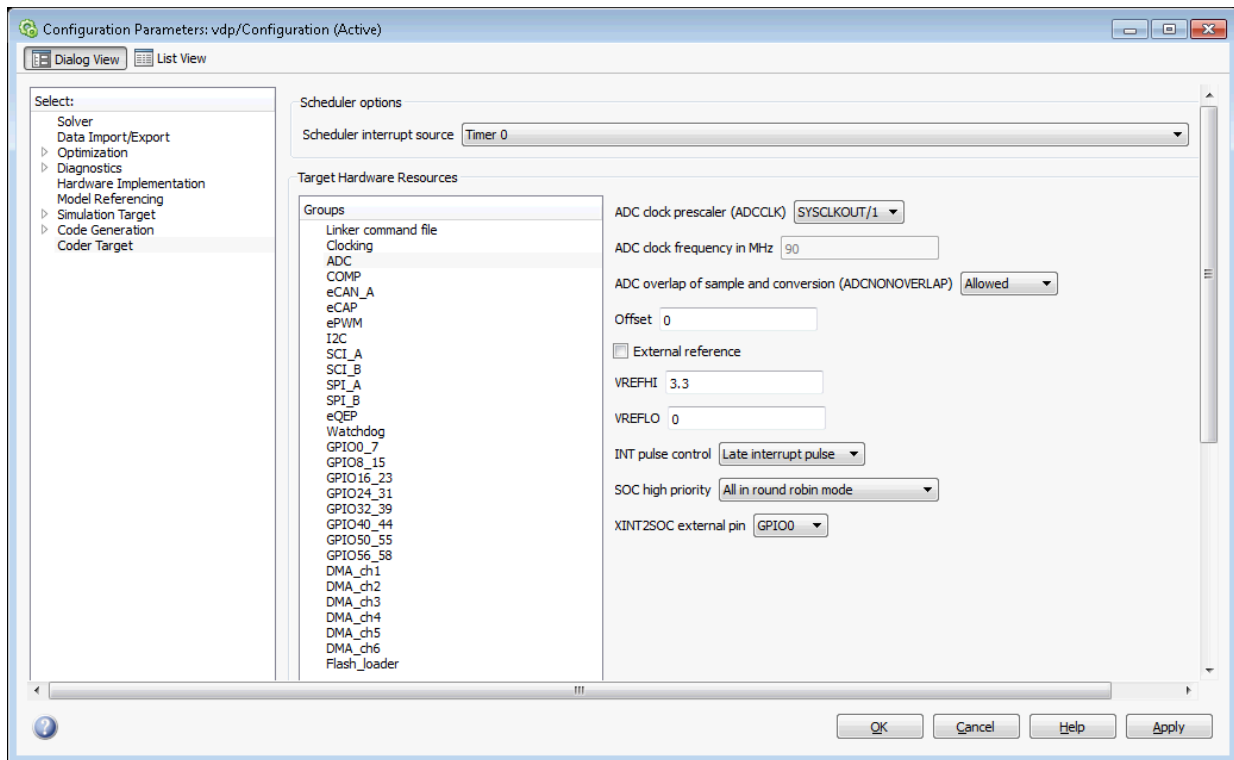
Compatibility Considerations

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 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 no longer occur.

Addition functions in libraries that implement full-precision addition, such as the ANSI C library, are no longer 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.
```

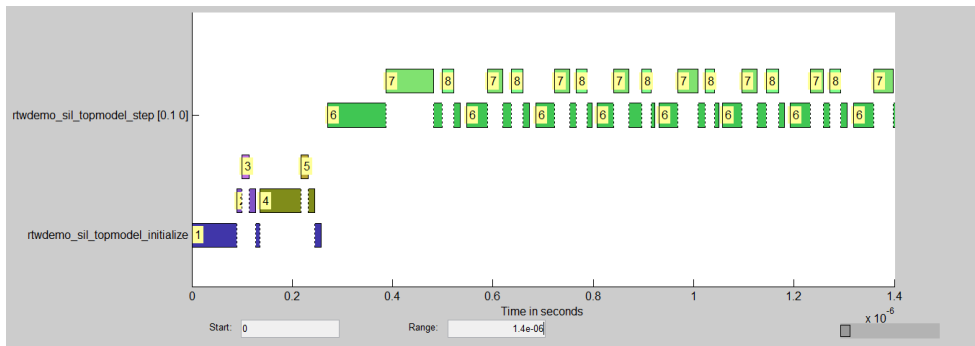
Previously, if function profiling was enabled for a SIL or PIL simulation, building your model produced an error message. For example:

```
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

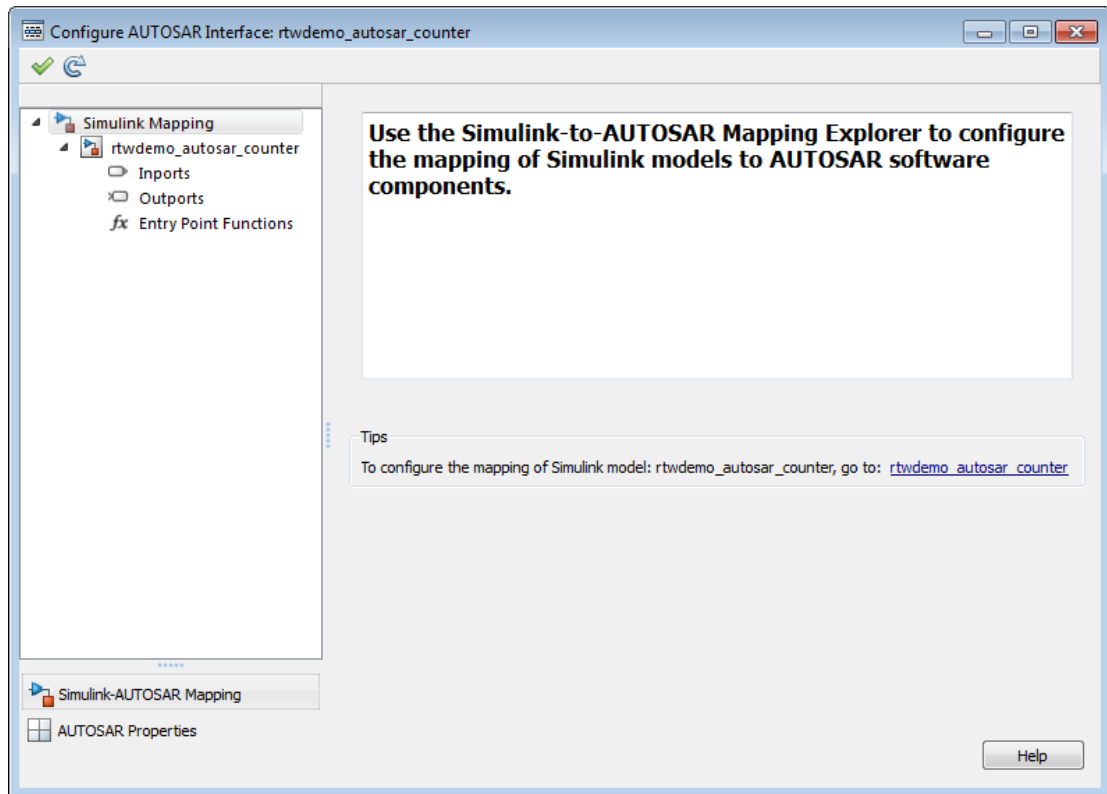
AUTOSAR user interface and round trip ARXML file import and export improvements

Improved graphical user interfaces for AUTOSAR configuration

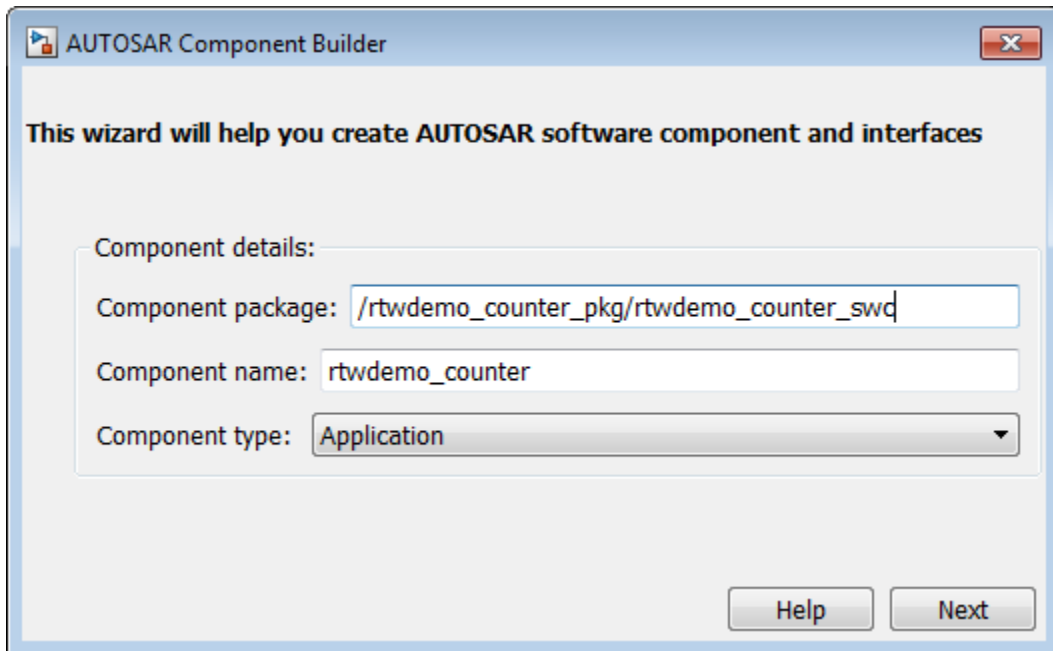
Embedded Coder software provides graphical user interfaces that allow you to add AUTOSAR elements to a Simulink model and map model components and interfaces to AUTOSAR components and interfaces. R2013a provides several improvements to the graphical user interfaces for AUTOSAR configuration:

- 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

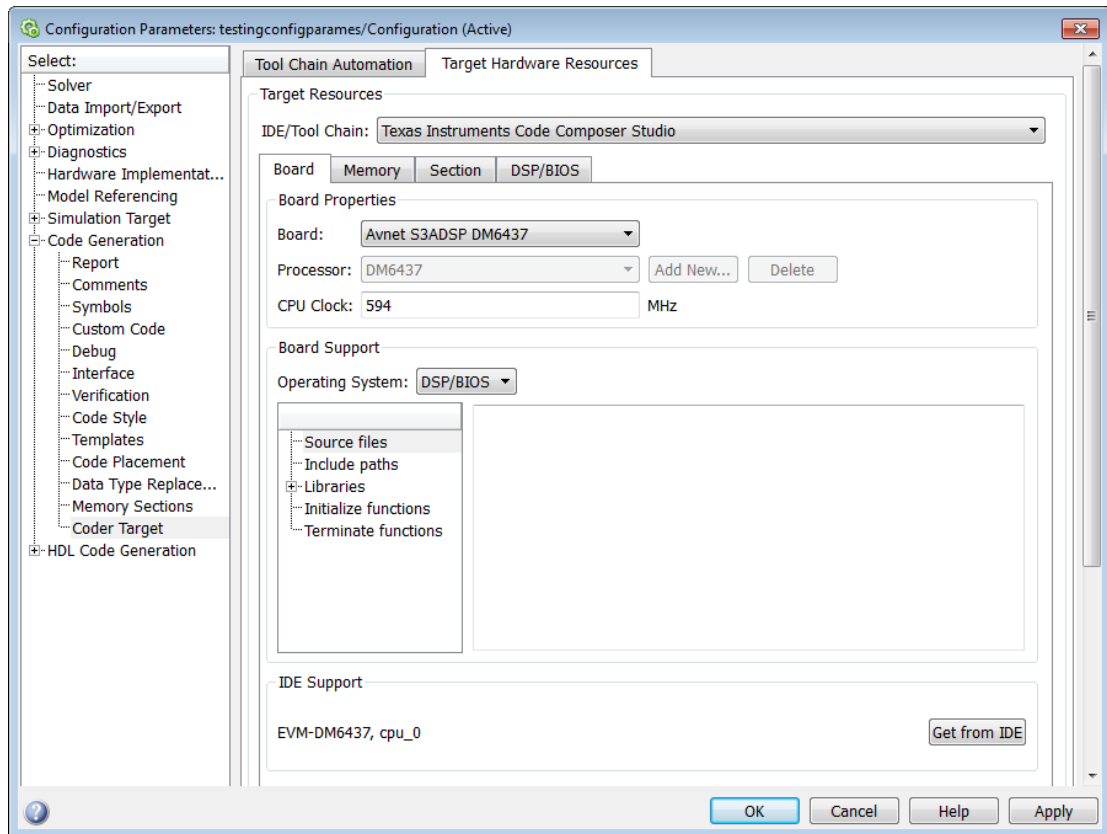
For more information, see [Map Semaphore or Mutex Operations to Target-Specific Implementations](#).

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”.

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.

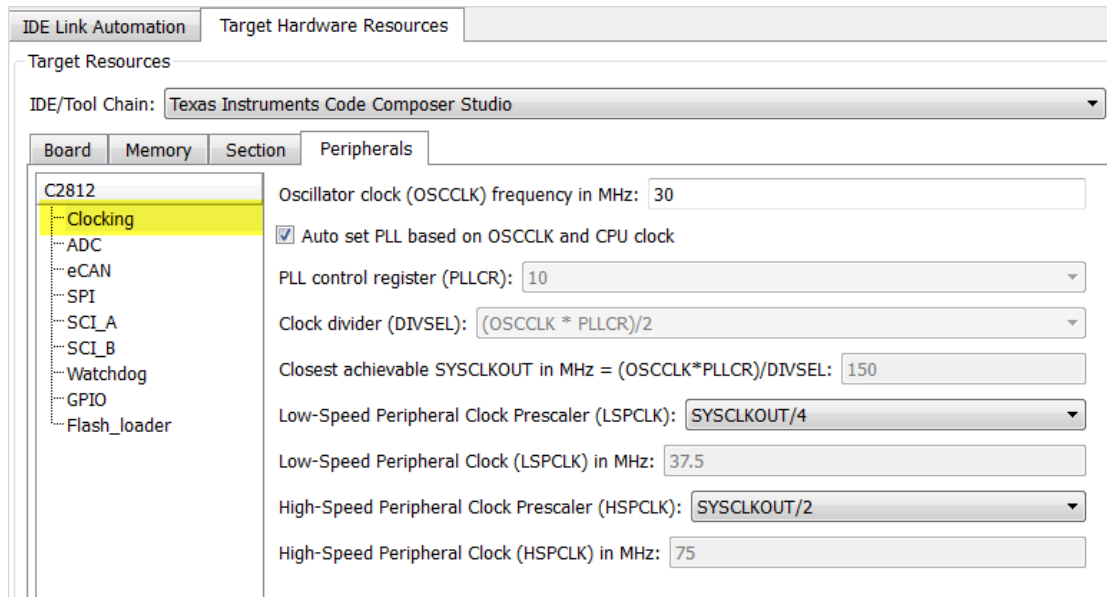
After installing the support package, you can open the block library. In the MATLAB Command Window, enter `adivdsp1ib`. The “Embedded Coder Support Package for Analog Devices DSPs” block library is also available in the Simulink Library Browser.

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

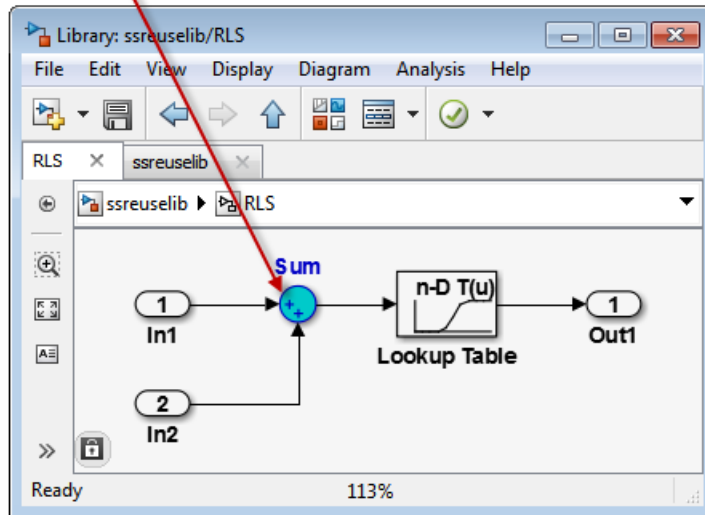
Code-to-model traceability links are now available in the generated code for a reusable library subsystem. Code-to-model traceability links for a reusable library subsystem appear in the comments of the generated code in the code generation report. The traceability link is the name of the library.

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 = looki_binlxy(*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., `logouts`) 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 interrunnable 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 6-8.

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 6-4.

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
<code>coder.profile.Timer</code>	<code>coder.profile.Timer</code>

Interface	Method or Property
coder.profile.ExecutionTime	display
	Sections
	TimerTicksPerSecond
	report
coder.profile.ExecutionTimeSection	ExecutionTimeInTicks
	MaximumExecutionTimeCallNum
	MaximumExecutionTimeInTicks
	MaximumSelfTimeCallNum
	MaximumSelfTimeInTicks
	Name
	Number
	NumCalls
	SampleOffset
	SamplePeriod
	SelfTimeInTicks
	TotalExecutionTimeInTicks
	TotalSelfTimeInTicks

Functionality Being Removed or Changed

The following functionality is being removed or changed:

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.connectivity.Timer	Call is forwarded to coder.profile.Timer without warning message.	coder.profile.Timer	All methods are the same as rtw.connectivity.Timer.
rtw.pil.ExecutionProfile-.display	Call is forwarded to coder.profile.ExecutionTime.display without warning message.	display	None

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
<code>rtw.pil.ExecutionProfile-</code>	Call is forwarded to <code>coder.profile.ExecutionTime.report</code> without warning message.	<code>report</code>	None
<code>rtw.pil.ExecutionProfile-</code> <code>rtw.pil.ExecutionProfile-</code>	Call is forwarded to <code>coder.profile.ExecutionTime.Sections</code> without warning message.	<code>Sections</code>	Uses property syntax
<code>rtw.pil.ExecutionProfile-</code> <code>rtw.pil.ExecutionProfile-</code>	Calls are forwarded to property <code>coder.profile.ExecutionTime.TimerTicksPerSecond</code> without warning message.	<code>TimerTicksPerSecond</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getMaxTicks</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.MaximumExecutionTimeInTicks</code> without warning message.	<code>MaximumExecutionTimeInTicks</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getName</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.Name</code> without warning message.	<code>Name</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getNumCalls</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.NumCalls</code> without warning message.	<code>NumCalls</code>	Uses property syntax
<code>rtw.pil.ExecutionProfile-</code>	Call is forwarded to <code>coder.profile.ExecutionTime.Number</code> without warning message.	<code>Number</code>	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
<code>rtw.pil.ExecutionProfileSection.getTicks</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.ExecutionTimeInTicks</code> without warning message.	<code>ExecutionTimeInTicks</code>	Uses property syntax
<code>rtw.pil.ExecutionProfile-</code>	Call is forwarded to the legacy <code>getTimes</code> function without warning message.	Calculate execution time in seconds by the formula $\text{ExecutionTimeInSecs} = \text{ExecutionTimeInTicks} / \text{TimerTicksPerSecond}$.	No equivalent to <code>getTimes</code> in new interface.
<code>rtw.pil.ExecutionProfileSection.getTotalTicks</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.TotalExecutionTimeInTicks</code> without warning message.	<code>TotalExecutionTimeInTicks</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getSampleOffset</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.SampleOffset</code> without warning message.	<code>SampleOffset</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getSamplePeriod</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.SamplePeriod</code> without warning message.	<code>SamplePeriod</code>	Uses property syntax
<code>rtw.pil.ExecutionProfileSection.getTotalSelfTicks</code>	Call is forwarded to <code>coder.profile.ExecutionTimeSection.TotalSelfTimeInTicks</code> without warning message.	<code>TotalSelfTimeInTicks</code>	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_sfundarray_add): '<S1>/rtwdemo_sfundarray_add' */

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

```

Now, the generated code is:

```

int32_T i;

/* S-Function (rtwdemo_sfundarray_add): '<S1>/rtwdemo_sfundarray_add' */

array3d_add(rtb_Output1, rtwdemo_lct_ndarray_ConstP.Constant_Value,
            rtwdemo_lct_ndarray_ConstP.Constant1_Value, 1, 2, 3);

```

For more information, see [Integrate External Code Using Legacy Code Tool](#).

Element-Wise Operations as Inputs to Intrinsic Functions

In previous releases, element-wise operations were performed in temporary variables before being used as inputs in an intrinsic function call. In R2012a, element-wise operations are performed within the intrinsic function call to improve memory usage and execution speed.

For example, in previous releases when you generated code for the following MATLAB code:

```
function y = matrixExpand(u1, u2)
```

```
eml.varsize('u1', [4, 8, 10]);
eml.varsize('u2', [4, 8, 10]);
y = isnan(u1 + u2);
```

element-wise operations were stored in a temporary variable, `x_data`, which became the input to the generated intrinsic function, `muDoubleScalarIsNan`:

```
for (i = 0; i <= loop_ub; i++) {
    x_data[i] = u1_data[i] + u2_data[i];
}
...
for (i = 0; i <= loop_ub; i++) {
    y_data[i] = muDoubleScalarIsNaN(x_data[i]);
}
```

In R2012a, the temporary variable is eliminated in the generated code and the element-wise operations occur in the function call input:

```
for (i = 0; i <= loop_ub; i++) {
    y_data[i] = muDoubleScalarIsNaN(u1_data[i] + u2_data[i]);
}
```

Enhancements to Custom Storage Classes in Simulink and mpt Packages

In this release, enhancements have been made to the following custom storage classes (CSCs) in the Simulink package.

- **Owner** property added to `Const`, `Volatile`, `ConstVolatile`, `ExportToFile`
- **Definition file** property added to `Const`, `Volatile`, `ConstVolatile`, `ExportToFile`
- **Header file** property added to `Const`, `Volatile`, `ConstVolatile`, `Define`

The following enhancements have been made to CSCs in the mpt package

- **Owner** property has been added to `ExportToFile`
- Settings for the **Owner** and **Definition file** properties for `Global`, `Custom`, `Volatile`, and `ConstVolatile` CSCs have been moved from the **Other Attributes** tab to the **General** tab of the Custom Storage Class Designer.

Code Generation Report Includes Simulink Web View

R2012a supports integration of the Simulink Web view into the code generation report. You can view the generated code and model in a single web browser window without MATLAB and Simulink installed on your computer.

To generate a code generation report with the model Web view, on the **Code Generation > Report** pane of the model configuration parameters, select:

- **Create code generation report**
- **Generate model Web view**
- **Open report automatically** (optional)

For navigation between the generated code and the model in the Web view, select

- **Code-to-model**
- **Model-to-code**

For more information, see *Include Model Web View in HTML Code Generation Report*. The model Web view requires a Simulink Report Generator™ license.

LDRA Testbed Code Coverage Annotations in Code Generation Report

If you specify the LDRA Testbed® code coverage tool for a SIL/PIL simulation, the code generation report provides summary data and code annotations with LDRA Testbed coverage information. Each code annotation is associated with a code feature and indicates the nature of the feature coverage during code execution. See *Code Coverage Summary and Annotations in Code Generation Report*.

You should not use the code generation report alone to check that your coverage goals have been achieved. You must refer to the LDRA Testbed Report. See *View Code Coverage Information at the End of SIL or PIL Simulations*.

Generated Identifiers Enhancements

Simplified Identifiers for Model Reference Code

Previously, model reference identifiers were generated with the `mr_` prefix. In R2012a, code generation no longer includes the `mr_` prefix to identifiers. This naming convention is now consistent with the code generation of subsystem identifiers and other identifiers. For more information, see *Configuring Generated Identifiers*.

Consistent Identifiers for Comparing Generated Code

To generate unique identifiers in the generated code, the code generation process inserts a mangling string in an identifier name. Previously, the mangling string was generated using the full block path name, which included the model name. In R2012a, the mangling

string uses the Simulink Identifier (SID), which is unique within the model. This mangling string allows for consistent identifiers for similar or derived models, because the SID is persistent even if you change the name of the model. If you create another model using **Save As**, the SID is preserved for each block. For blocks in a subsystem, the SID is preserved whether you build the subsystem or build the model containing the subsystem.

For example, you might want to make a structural change to a model and then see the impact of the change on the generated code. You can save your model using **Save As** and make a change to the saved model. To see only the change in the generated code due to the change in the model, you can compare the generated code from the original and derived model. Before R2012a, the identifiers from the derived model were different, because the mangling string included the different model names. It was difficult to see only the difference in the generated code from the change in the model. Now, when you compare the generated code for the two models, the difference is just the code resulting from the change in the derived model.

If you have an Embedded Coder license, see [Configure Generated Identifiers in Embedded System Code](#) for more information on customizing generated identifiers.

Code Replacement Enhancements

R2012a provides the following enhancements to code replacement library support.

Target Function Libraries Renamed to Code Replacement Libraries

In R2012a, target function libraries (TFLs) are renamed to code replacement libraries (CRLs). The change is reflected in software, demos, and documentation. The changes include the following:

- The model configuration parameter **Target function library** (TargetFunctionLibrary) is renamed to **Code replacement library** (CodeReplacementLibrary). The command line parameter TargetFunctionLibrary is still supported, but when you save a model, the library value is saved using the parameter CodeReplacementLibrary.
- The code replacement demo `rtwdemo_tfl_script` is renamed to `rtwdemo_crl_script`, and the `rtwdemo_tfl*` models associated with the demo are renamed to `rtwdemo_crl*`. For example, the model `rtwdemo_tfladdsub` is renamed to `rtwdemo_crladdsub`.
- The code replacement demo `coderdemo_tfl` is renamed to `coderdemo_crl`.

- The Target Function Library (TFL) Viewer is renamed to Code Replacement Viewer.

Code replacement related items that have *not* been renamed include code replacement classes, functions, and commands. Examples include the `RTW.Tf1COperationEntry` class, the `setTf1CFunctionEntryParameters` function, and the `RTW.viewTf1` 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:

```
>> cr1=get_param('mode1','TargetFcnLibHandle')
>> RTW.viewTf1(cr1)
```

When debugging a CRL entry, you can use code replacement report information together with hits and misses information in the Code Replacement Viewer to determine why a replacement function was not used in the generated code.

For more information, see [Trace Code Replacements Generated Using Your Code Replacement Library and Determine Why Code Replacement Functions Were Not Used](#).

Code Replacement Support for Simulink Matrix Division and Inversion Operators

Embedded Coder software now provides Simulink code replacement support for the following nonscalar division and inversion operators:

Operator	Key
Matrix right division (/)	RTW_OP_RDIV
Matrix left division (\)	RTW_OP_LDIV
Matrix inversion (inv)	RTW_OP_INV

For more information, see [Map Nonscalar Operators to Target-Specific Implementations](#).

Code Replacement Support for MATLAB Coder `fix`, `hypot`, `round`, and `sign` Functions

Embedded Coder software now provides MATLAB Coder code replacement support for `fix`, `hypot`, `round`, and `sign` functions.

Integer Functions Now Return Real-World Values

The following functions now return real-world values instead of stored integer values: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`.

Compatibility Considerations

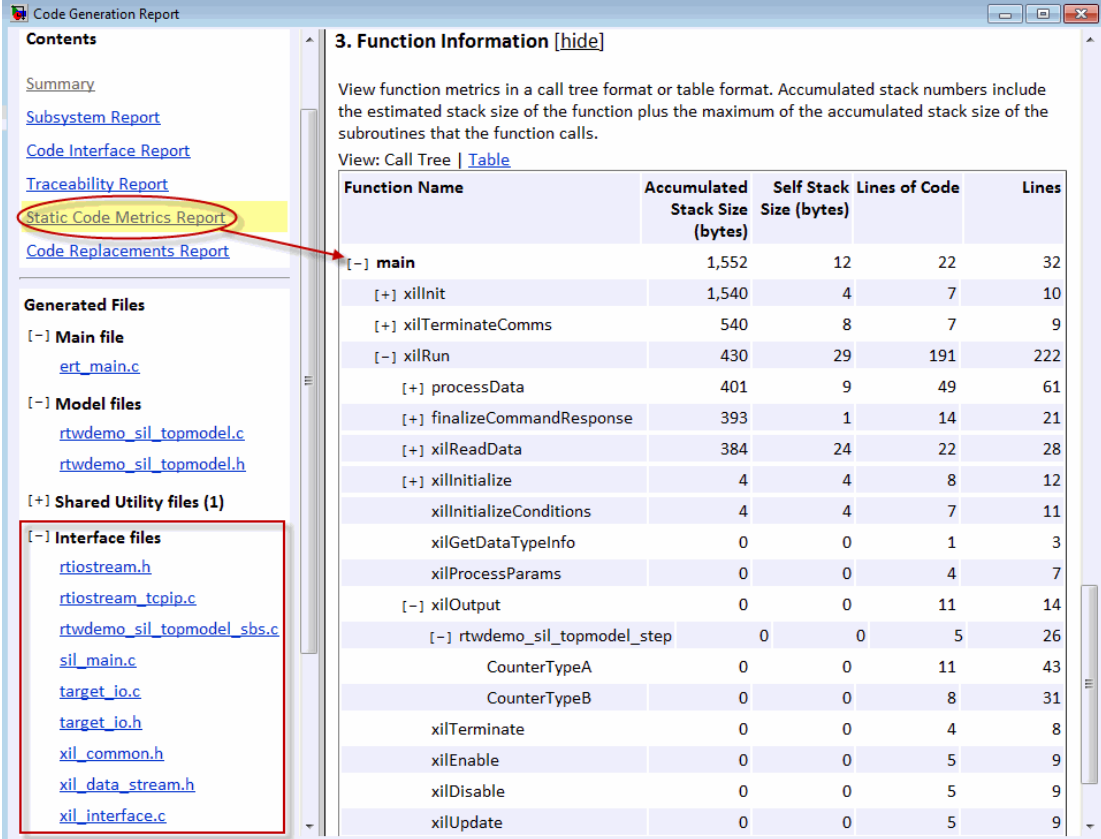
In code generation with MATLAB Coder or Simulink Coder, if you used a CRL to replace a cast in your replacement function, silent incorrect numerical results may occur. The numerical results will not change if the input `fi` object has binary-point scaling and zero fractional length. To optimize code generation, these integer functions now use floor rounding, instead of nearest rounding, when the input fraction length equals 0. You should reevaluate your integer cast replacement functions and update their replacement tables.

SIL and PIL Enhancements

R2012a supports the following enhancements for software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

SIL and PIL Test Harness Files in Code Generation Report

For top-model and Model block SIL and PIL simulations, the software now displays test harness files and the corresponding static code metrics in the code generation report.



3. Function Information [hide]

View function metrics in a call tree format or table format. Accumulated stack numbers include the estimated stack size of the function plus the maximum of the accumulated stack size of the subroutines that the function calls.

View: Call Tree | [Table](#)

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines
[-] main	1,552	12	22	32
[+] xilInit	1,540	4	7	10
[+] xilTerminateComms	540	8	7	9
[-] xilRun	430	29	191	222
[+] processData	401	9	49	61
[+] finalizeCommandResponse	393	1	14	21
[+] xilReadData	384	24	22	28
[+] xilInitialize	4	4	8	12
xilInitializeConditions	4	4	7	11
xilGetTypeInfo	0	0	1	3
xilProcessParams	0	0	4	7
[-] xilOutput	0	0	11	14
[-] rtwdemo_sil_topmodel_step	0	0	5	26
CounterTypeA	0	0	11	43
CounterTypeB	0	0	8	31
xilTerminate	0	0	4	8
xilEnable	0	0	5	9
xilDisable	0	0	5	9
xilUpdate	0	0	5	9

This feature helps you to:

- Understand and review the SIL and PIL verification process.
- See how your registered custom target connectivity files fit into the target application that runs during a SIL or PIL simulation.

This feature is not available for simulations that you run with the PIL block. For more information, see [View Test Harness Files in Code Generation Report](#).

PIL Support for Code Coverage with LDRA Testbed

The target connectivity API supports code coverage with LDRA Testbed for the following types of PIL simulation:

- Top-Model PIL
- Model block PIL

Previously, support for code coverage during a PIL simulation was only available in special cases, where your PIL application could write directly to the host file system.

You can run PIL simulations on simulator or target hardware and collect code coverage metrics to support high integrity workflows, for example, DO-178B and ISO 26262. For more information, see *Use a Code Coverage Tool in SIL and PIL Simulations*.

Seamless Switching Between SIL and PIL for Top-Model and Model Block

If you select **Configuration Parameters > SIL and PIL Verification > Enable portable word sizes**, you can switch between the SIL and PIL simulation modes without:

- Changing configuration parameters of your model
- Regenerating code (if your model is up-to-date)

This feature:

- Applies only to top-model and Model block SIL/PIL
- Requires that the code can be compiled by both the host computer and the target platform

If your target uses code that cannot be compiled on the host, then you see compilation errors when you try to simulate the model in SIL mode. You might be able to work around this problem by adding the source code files to the `SkipForSil` group in the build information object `RTW.BuildInfo`. The SIL build on the host platform does not compile source files present in the `SkipForSil` group. See *Code that the Host Cannot Compile*.

Enhanced Hardware Implementation Support

Host and Target Floating Point Data Type Sizes

The host and target floating point data type sizes must be the same. Previously, a mismatch would produce undefined behaviour resulting in a simulation failure. Now, the

software generates an error with a clear message when the host and target data types are *not*:

- 32 bits for `single`
- 64 bits for `double`

For more information, see [Hardware Implementation Support](#).

Word-Addressable Targets

Previously, the target connectivity API did not support word-addressable targets for PIL simulations or SIL simulations with `PortableWordSizes` enabled. This limitation has been removed.

In addition, data type sizes that are smaller than the target word sizes are now supported. See [Hardware Implementation Support](#).

The software uses the MATLAB host byte order when sending words through the `rtIOStream` API. For information about host byte ordering, see `computer` 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 `logouts`.
- 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 7-13
- “Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors” on page 7-13
- “Support Removed for Freescale MPC5xx” on page 7-14
- “Limitation: Parallel Builds Not Supported for Embedded Targets” on page 7-14

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...
<code>rtwdemo_roll_axis</code>	Generate code for a roll axis autopilot control system. The <code>rtwdemo_roll</code> model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. <code>rtwdemo_roll</code> replaces <code>rtwdemo_f14</code> .
<code>c28335_pmsmfoc_script</code>	Schedule a multi-rate controller for a permanent magnet synchronous machine (PMSM) motor control application that runs on a Texas Instruments F28335 processor. To get this demo, use <code>targetinstaller</code> or <code>supportPackageInstaller</code> to install the <i>Embedded Coder Support Package for Texas Instruments C2000 Processors</i> .

The following demos have been enhanced in R2012a:

Demo...	Now...
<code>coderdemo_cr1</code>	Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs).

Demo...	Now...
rtwdemo_crl_script	<ul style="list-style-type: none">• Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs).• Illustrates code replacement for Simulink matrix division and inversion operators.
rtwdemo_pmsmfoc_script	Added torque and position control modes to controller, parameterized motor and sensor data, and added support for specifying baud rate in example PIL implementation.
rtwdemo_radar	Shows how to simulate and generate code for the model <code>rtwdemo_eml_aero_radar</code> , which contains a MATLAB script.
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 8-5
- “Support Added for Wind River VxWorks 6.8” on page 8-6
- “Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335” on page 8-6
- “New Target Function Library for Intel IPP/SSE (GNU)” on page 8-6
- “Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors” on page 8-6
- “Support Removed for Altium TASKING” on page 8-7
- “Support Removed for Infineon C166” on page 8-7
- “Support Ending for Green Hills MULTI in a Future Release” on page 8-7
- “Support Ending for Freescale MPC5xx in a Future Release” on page 8-7

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 <http://www.mathworks.com/hardware-support/texas-instruments.html> and <http://www.mathworks.com/hardware-support/analog-devices.html>.

Also, check the Texas Instruments and Analog Devices Web sites for support information about using their tools on 64-bit Windows platforms.

Support Added for Wind River VxWorks 6.8

You can automatically generate and integrate code with the Wind River VxWorks 6.8 RTOS using makefiles via the XMakefiles feature. For more information, see [Choosing an XMakefile Configuration and Working with Wind River VxWorks RTOS](#).

Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335

This release adds support for Serial Communication Interface (SCI) communications during processor-in-the-loop (PIL) simulations with Texas Instruments™ C28035 and C28335 microcontrollers. Using SCI for PIL simulations is much faster than using an IDE debugger for PIL.

For more information, see [Serial Communication Interface \(SCI\) for Texas Instruments C2000, Example — Performing a Model Block PIL Simulation via SCI Using Makefiles](#), and the `fuelsys_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 Targets Desktop 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	Overflows saturate to either the minimum or	An overflow associated with a signed 8-bit integer saturates to -128 or +127.

Check Box	When to Use This Setting	Overflow Handling	Example of a Result
	explicit saturation protection in the generated code.	maximum value that the data type can represent.	
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:

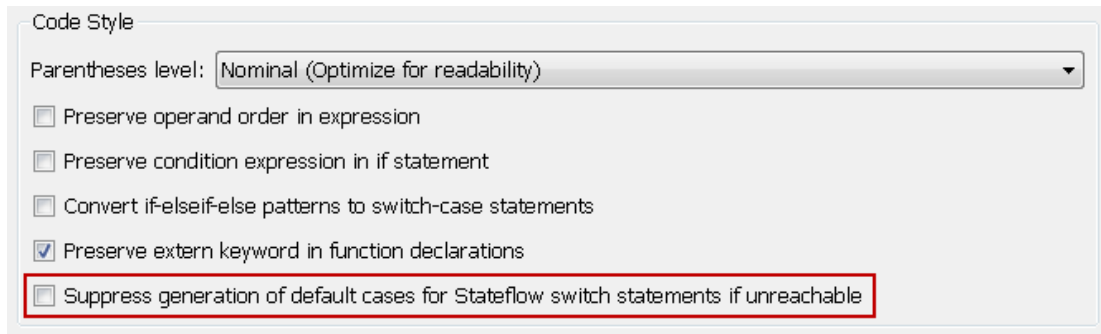
- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

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...
<code>rtwdemo_pmsmfoc_script</code>	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented

Demo...	Now...
	Control for a Permanent Magnet Synchronous Machine
<code>rtwdemo_sil_pil_script</code>	Incorporates code execution profiling
<code>rtwdemo_tfl_script</code>	Shows how you can align nonscalar data passed into a target function library (TFL) code replacement function
<code>fuelsys_pil</code>	Incorporates using serial communication interface to communicate during PIL simulation

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. 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

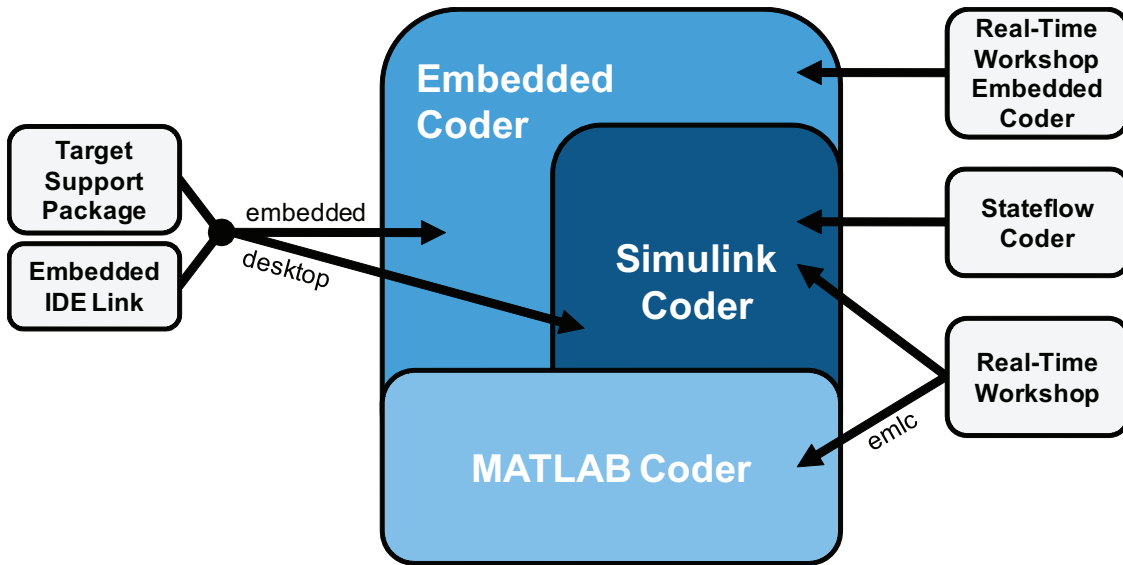
- “Product Restructuring Overview” on page 9-2
- “Resources for Upgrading from Real-Time Workshop Embedded Coder” on page 9-3
- “Migration of Embedded MATLAB Coder Features to MATLAB Coder” on page 9-4
- “Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder” on page 9-4
- “Interface Changes Related to Product Restructuring” on page 9-5
- “Simulink Graphical User Interface Changes” on page 9-5

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
	Freescall MPC5xx		x
	Infineon C166		x
	Texas Instruments C2000		x
	Texas Instruments C5000		x
	Texas Instruments C6000		x
	Linux OS	x	x

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
	Windows OS	x	
	VxWorks RTOS		x

Interface Changes Related to Product Restructuring

You will see interface changes as part of restructuring the Coder products.

- In the Simulink Configuration Parameters dialog box, changes to code generation related elements
- In Simulink menus, changes to code generation related elements
- In Simulink blocks, including block parameters and dialog boxes, and block libraries, changes to code generation related elements
- In error messages, tool tips, demos, and product documentation, references to Real-Time Workshop Embedded Coder, Real-Time Workshop, and Stateflow Coder and related terms are replaced with references to the latest software

Simulink Graphical User Interface Changes

Where...	Previously...	Now...
Configuration Parameters dialog box	Real-Time Workshop pane	Code Generation pane
Model diagram window	Tools > Real-Time Workshop	Tools > Code Generation
Subsystem context menu	Real-Time Workshop	Code Generation
Subsystem Parameter dialog box	Following parameters on main pane: <ul style="list-style-type: none"> • Real-Time Workshop system code • Real-Time Workshop function name options • Real-Time Workshop function name • Real-Time Workshop file name options 	On new Code Generation pane and renamed: <ul style="list-style-type: none"> • Function packaging • Function name options • Function name • File name options • File name (no extension)

Where...	Previously...	Now...
	<ul style="list-style-type: none"> • Real-Time Workshop 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 9-6
- “No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects” on page 9-6
- “Parts of Data Class Infrastructure Not Available” on page 9-7
- “No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer” on page 9-8
- “Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release” on page 9-9

Memory Section Enhancements

- Pragma 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 can not 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...
<code>BitFieldName</code>	<code>StructName</code>
<code>FileName+IncludeDelimiter</code>	<code>HeaderFile</code>

- Initial value of MPT data objects inside `mpt.CustomRTWInfoSignal` – *warning displayed since R2006a*

Compatibility Considerations

- When you use a removed feature, Simulink now generates an error.
- When loading a MAT-file that uses an unsupported feature, the load operation suppresses the generated error such that it is not visible. In addition, MATLAB silently deletes data that had been associated with the unsupported feature. To prevent loss of data when loading a MAT-file, load and resave the file with R2010b or earlier.

No Longer Generating Pragma for Data Defined with Built-In Storage Class `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer`

The code generator no longer generates a `pragma` around definitions or declarations for data that has the following built-in storage classes:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

Prior to R2011a, based on model configuration parameters for specifying memory sections and the built-in storage class defined for data, the code generator would do the following:

For Built-In Storage Class...	Generate pragma Around...
<code>ExportedGlobal</code>	Data definition and declaration
<code>ImportedExtern</code>	Data declaration
<code>ImportedExternPointer</code>	Data declaration

The code generator now treats data with these built-in storage classes like custom storage classes with no memory section specified.

Compatibility Considerations

To work around this change, select a custom storage class that uses the memory section of interest for the data.

Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release

In a future release, data classes `Simulink.CustomParameter` and `Simulink.CustomSignal` will no longer be supported because they are equivalent to `Simulink.Parameter` and `Simulink.Signal`.

Compatibility Considerations

If you use the data class `Simulink.CustomParameter` or `Simulink.CustomSignal`, Simulink posts a warning that identifies the class and describes one or more techniques for eliminating it. You can ignore these warnings in R2011a, but consider making the described changes now because the classes will be removed in a future release.

AUTOSAR Enhancements

The following enhancements are available in R2011a.

Calibration Parameters

Previously, the software supported only calibration parameters that were defined by a calibration component. These parameters could be accessed by all AUTOSAR Software Components. The AUTOSAR standard also specifies an internal calibration parameter that is defined and accessed by only one AUTOSAR Software Component. The software now supports:

- AUTOSAR internal calibration parameters, including the import and export of initial values of these parameters.
- A bus object data type (AUTOSAR record type) to import and export both kinds of calibration parameters.

For more information, see [Calibration Parameters and Configure Calibration Parameters](#) 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 perform automatic configuration checks. Instead, you can use the `cgv.Config` class to perform a manual configuration check of your model. Before calling `cgv.CGV.run`, perform a manual configuration check of your model. Otherwise, an error might occur later in the process. For more information, see Programmatic Code Generation Verification.

Changes to the `cgv.CGV` class parameters are listed in the following table.

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
<code>LogMode</code> removed from <code>cgv.CGV</code>	Errors	<code>LogMode</code> parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the <code>LogMode</code> parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.
<code>Processor</code> removed from <code>cgv.CGV</code>	Errors	<code>Processor</code> parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the <code>Processor</code> parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code>

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
			method to adjust the model configuration.
<code>SaveModel</code> removed from <code>cgv.CGV</code>	Errors	<code>SaveModel</code> parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the <code>SaveModel</code> parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.
<code>ConfigModel</code> removed from <code>cgv.CGV</code>	Warns if set to off Errors if set to on	<code>cgv.Config.configModel</code> method	To check your model before running CGV, replace the <code>cgv.CGVConfigModel</code> parameter with a call to the <code>cgv.Config.configModel</code> method
<code>CheckInterface</code> parameter from <code>cgv.CGV</code>	Warns if set to off Errors if set to on	<code>CheckOutputs</code> parameter in <code>cgv.Config</code>	To check your model before running CGV, pass the <code>CheckOutputs</code> parameter to the constructor for <code>cgv.Config</code> . Then call the <code>cgv.Config.configModel</code> method to adjust the model configuration.

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
tasking and custom values removed from the <code>Connectivity</code> parameter of <code>cgv.CGV</code>	Errors	<code>pil</code> , a new value for the <code>cgv.CGV Connectivity</code> parameter	Replace calls to the <code>cgv.CGV</code> constructor using the parameter-value arguments, (<code>'Connectivity'</code> , <code>'tasking'</code>) or (<code>'Connectivity'</code> , <code>'custom'</code>), with (<code>'Connectivity'</code> , <code>'pil'</code>).

Changes to the `cgv.Config` class parameters are listed in the following table:

Parameter	What Happens When You Use Parameter?	Compatibility Considerations
<code>CheckOutports</code> parameter added to <code>cgv.Config</code>	Defaults to <code>on</code> . Compiles the model. Then checks that the model outputport configuration is compatible with the <code>cgv.CGV</code> object.	If your script fixes errors reported by <code>cgv.Config</code> , you can set <code>CheckOutports</code> to <code>off</code> .
<code>LogMode</code> parameter from <code>cgv.Config</code>	Change in behavior	If you do not give a value for <code>LogMode</code> , 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 Outputport 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 9-20
- “Execution Profiling during PIL Simulation” on page 9-21

- “Location of Blocks for Embedded Targets” on page 9-21
- “Location of Demos for Embedded IDEs and Embedded Targets” on page 9-22
- “Multicore Deployment with Rate-Based Multithreading” on page 9-23
- “Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)” on page 9-24
- “Changes to Frame-Based Processing” on page 9-24
- “New Support for Analog Devices Blackfin BF50x and BF51x Processors” on page 9-25
- “Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors” on page 9-26
- “Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI” on page 9-26
- “Support for Texas Instruments Delfino C2834x Processors” on page 9-26
- “Ending Support for Altium TASKING in a Future Release” on page 9-27
- “Ending Support for Freescale MPC5xx in a Future Release” on page 9-27
- “Ending Support for Infineon C166 in a Future Release” on page 9-27
- “Removed Methods and Arguments” on page 9-27

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

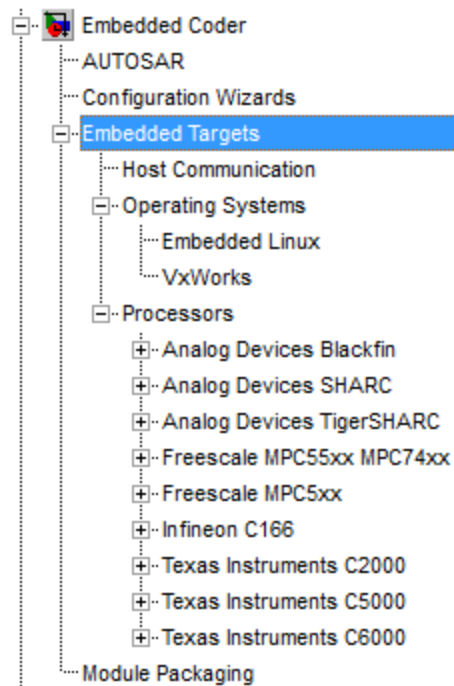
Note: You can only use these features in the 32-bit version of your MathWorks products. To use these features on 64-bit hardware, install and run the 32-bit versions of your MathWorks products.

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.

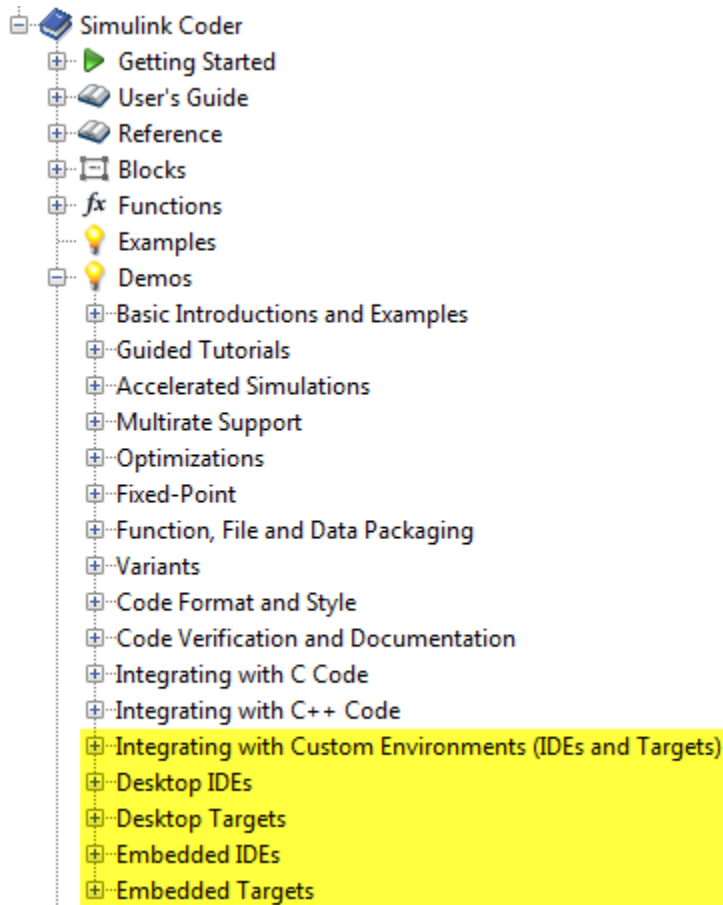


Embedded Targets includes the following types of blocks:

- 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 removed - see release notes). The function asks you whether to upgrade each block. If you select yes, the function detects the status of the frame bit on the input port of the block. If the frame bit is 1 (frames), the function sets the **Input processing** parameter to **Columns as channels** (frame based). If the bit is 0 (samples), the function sets the parameter to **Elements as channels** (sample based).

A future release will remove the **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 models 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. For additional information about supported versions, see the Support for Green Hills MULTI topic online.

Support for Texas Instruments Delfino C2834x Processors

You can now generate code for the following embedded processors when you use Embedded Coder software with Texas Instruments Code Composer Studio software:

- C28341
- C28342
- C28343
- C28344
- C28345
- C28346

The new C2834x (c2834xlib) block library contains the following blocks:

- C2000 CAN Calibration Protocol
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit

- C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger
- C28x Watchdog
- C280x/C2803x/C28x3x/c2834x eCAN Receive
- C280x/C2803x/C28x3x/c2834x eCAN Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x eCAP
- C280x/C2802x/C2803x/C28x3x/c2834x ePWM
- C280x/C2803x/C28x3x/c2834x eQEP

Ending Support for Altium TASKING in a Future Release

Support for the Altium TASKING IDE will end in a future release of the Embedded Coder product.

Ending Support for Freescale MPC5xx in a Future Release

Support for the Freescale MPC5xx processor family will end in a future release of the Embedded Coder product.

Ending Support for Infineon C166 in a Future Release

Support for the Infineon C166 processor family will end in a future release of the Embedded Coder product.

Removed Methods and Arguments

Deprecated the `type` property for the Code Composer Studio IDE object. For example, entering the following text generates an error message:

```
infolist = IDE_Obj.list(type)
```

Changes to ver Function Product Arguments

The following changes have been made to `ver` function arguments related to embedded code generation products:

- The new argument `'embeddedcoder'` returns information about the installed version of the Embedded Coder product.
- The argument `'ecoder'`, which previously returned information about the installed version of the Real-Time Workshop Embedded Coder product, no longer works. The software displays a “not found” warning.

For more information about using the function, see the `ver` documentation.

Compatibility Considerations

If a script calls the `ver` function with the `'ecoder'` argument, update the script appropriately. For example, you can update the `ver` call to use the `'embeddedcoder'` argument.

New and Enhanced Demos

The following demos have been added in R2011a:

Demo...	Shows How You Can...
<code>coderdemo_tfl</code>	Use target function libraries (TFLs) to replace operators and functions in code generated by MATLAB Coder.
<code>rtwdemo_code_coverage_script</code>	Generate model coverage and code coverage reports, and use these reports to compare model coverage and code coverage results for parts of a model.
<code>rtwdemo_pmsmfoc_script</code>	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...
<code>vipstabilize_fixpt_beagleboard</code>	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.

