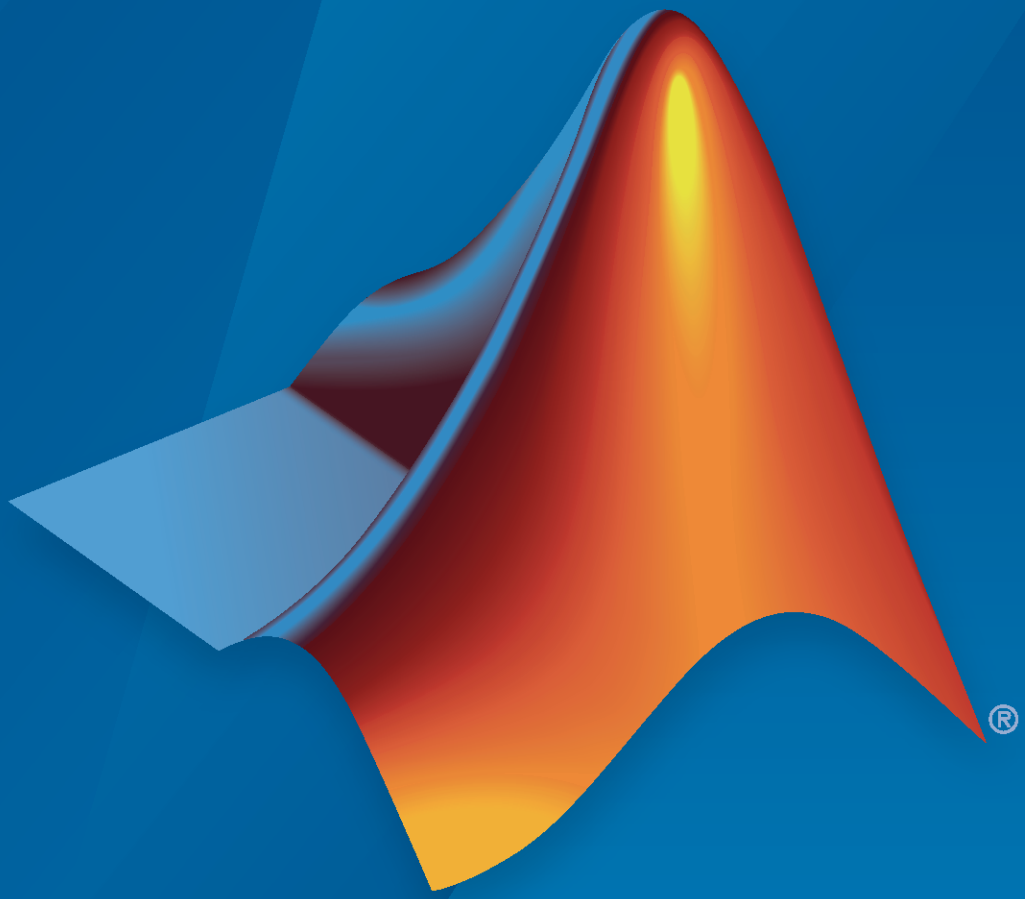
Embedded Coder® Release Notes



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

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

Phone: 508-647-7000

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

Embedded Coder® Release Notes

Check Bug Reports for Issues and Fixes

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

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

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

Contents

R2026a

R2025b

R2025a

R2024b

R2024a

R2023b

R2023a

R2022b

R2022a

R2021b

R2026a

Code Interface Configuration and Integration

Create new storage classes to represent Boolean, fixed-point, or integer data as structures of bit fields

Starting in R2026a, you can create new storage classes to represent Boolean, fixed-point, or integer data as structures of bit fields in the generated code.

In the Embedded Coder Dictionary, in the **Storage Class** section, enable the **Use bit fields** property to represent members of structures as bit fields. Then use the Code Mappings Editor to assign root-level I/O, parameters, signals, and states to storage classes with this configuration.

For an example, see “Represent Integer and Fixed-Point Data Using Bit Field Storage Class”.

Assign data in For Each Subsystem blocks to structured storage classes in Embedded Coder Dictionary

Starting in R2026a, you can assign internal signals and states in For Each Subsystem blocks to structured storage classes defined in the Embedded Coder Dictionary. Doing so enables you to represent internal data in the for-each subsystem as members of global structures in the generated code. For more information, see Embedded Coder Dictionary.

Support for variable-size signals in service interfaces that use built-in storage classes

Starting in R2026a, you can generate code for variable-size signals in service interfaces that use the Direct Access data communication method and the `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` storage class. To generate variable-size signals, select the **Support: variable-size signals** model configuration parameter. Before R2026a, this parameter was disabled for models that defined service interfaces.

For more information about generating code for variable-size signals, see “Variable-Size Signals in Generated Code”.

Multitask data store diagnostic enabled for models configured to use service interfaces

Starting in R2026a, the model configuration diagnostic parameter **Multitask data store** is enabled for models that are configured to use service interfaces. You can set the parameter to `error` or `warn`. The value `none` is not a valid setting.

For more information about the parameter, see **Multitask data store**.

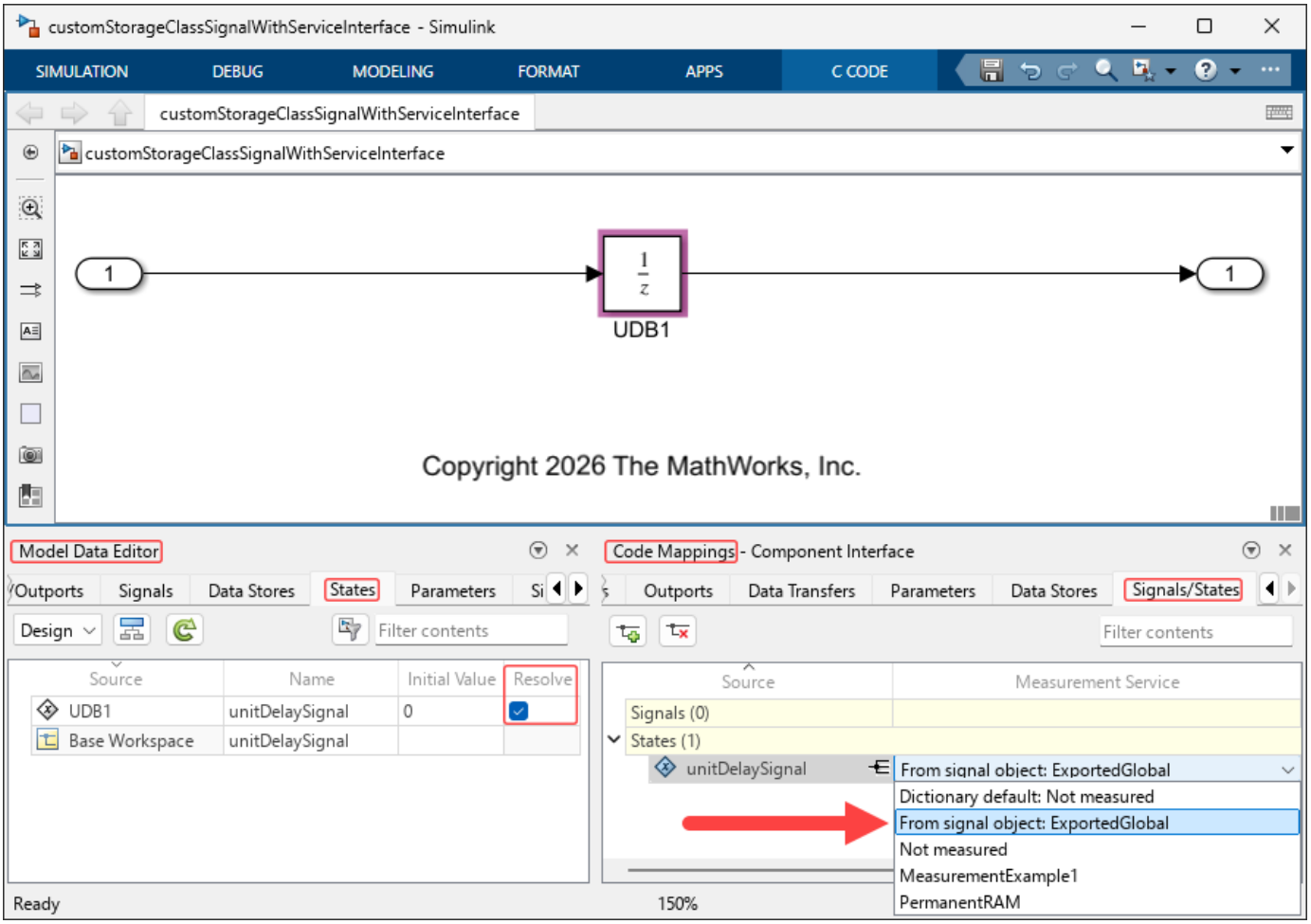
Resolve data elements to signal objects with non-Auto storage class in models with service interface configuration

Before R2026a, data elements in models with service interface configuration could only resolve to signal objects with `Auto` storage class. Starting in R2026a, these types of model elements can resolve to signal objects with a valid, non-`Auto` storage class:

- Root-level inports and outports
- Signals
- States
- Data stores

When you resolve a model element to a signal object with a storage class other than `Auto`, you need to specify the **Sender Service**, **Receiver Service**, or **Measurement Service** of the element, in the code mappings, as `From signal object: storage class`, where `storage class` is the storage class of the signal object.

When you switch the model configuration from data to service interface, the **Sender Service**, **Receiver Service**, or **Measurement Service** of each model element that is resolved to a signal object with a storage class other than `Auto` is automatically updated in the code mappings to `From signal object: storage class`.

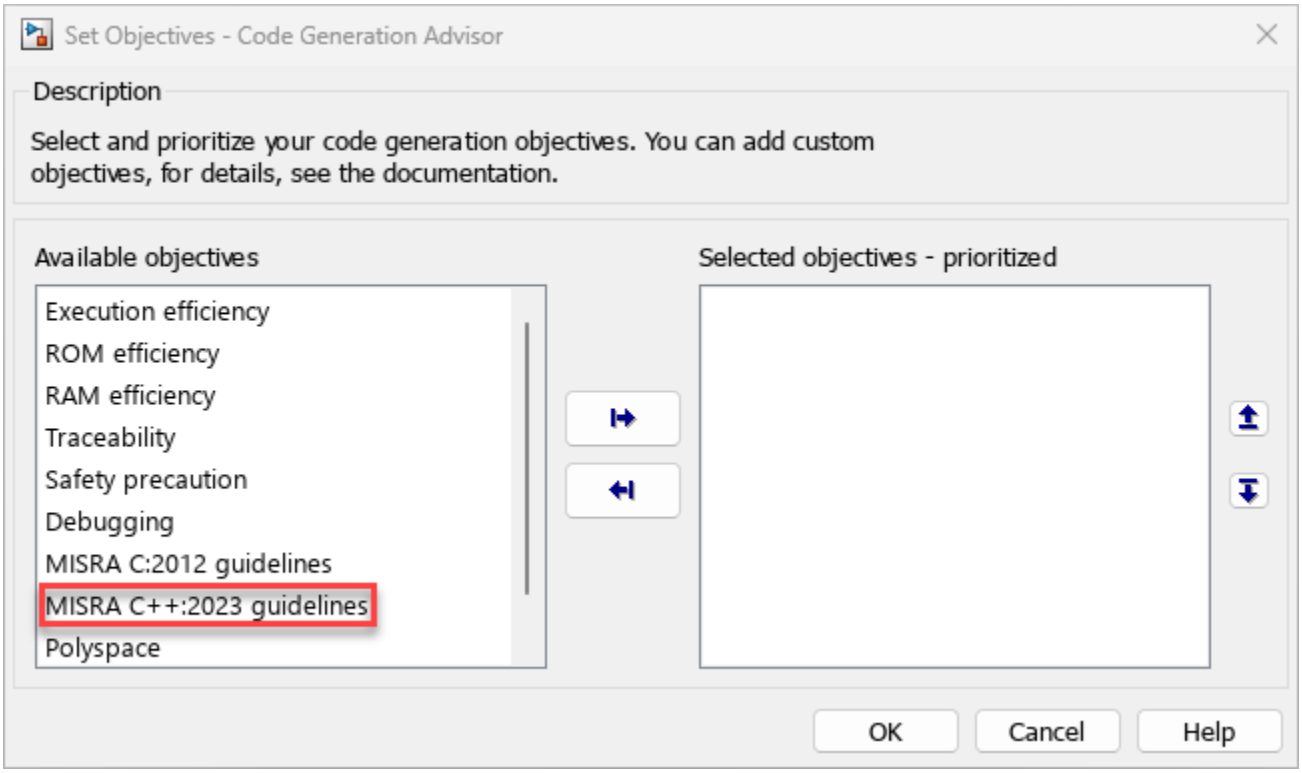


Code Generation

★Configure model for MISRA C++ 2023 compliance using compliance tables and Code Generation Advisor objective

In R2026a, you can verify compliance of generated C++ code with mandatory and required MISRA™ C++ 2023 guidelines listed in “MISRA C++:2023 Compliance Summary Tables”. Configure your model to avoid or mitigate violations using the “Explanatory Notes” section.

Automatically update model configuration parameters to improve compliance with MISRA C++ 2023 guidelines by using the Code Generation Advisor.



For details on specifying objectives and updating the model, see “Configure Model for Code Generation Objectives by Using Code Generation Advisor”.

★Use C99 macros for minimum-width integer constants

When the model configuration parameter **Data type replacement** is set to `Use C data types with fixed-width integers`, the code generator now produces C99 macros for minimum-width integer constants for values that have data types wider than `int`. For example, consider the `INT64_C` and `UINT64_C` macros in this generated code snippet:

```
int64_t myVar1 = -INT64_C(100);
uint64_t myVar2 = UINT64_C(100);
```

Previously, the code generator produced constants with `UL`, `ULL`, `L`, or `LL` suffixes. For example:

```
int64_t myVar1 = -100LL;
uint64_t myVar2 = 100ULL;
```

The integer literals in the generated code are portable between LP64 and LLP64 platforms. On a 64-bit Windows® development computer, when **Enable portable word sizes** is selected, you can generate code for 64-bit Linux® based target hardware and run software-in-the-loop simulations to test the generated code.

The code generator does not produce the macros if:

- The **Type limit identifier replacement header file** model configuration parameter is not empty.
- One or more custom replacement names for data types are specified.
- An AUTOSAR Classic Platform system target file is selected.
- The value of the **Data type replacement** model configuration parameter is `Use coder typedefs`.

For more information, see “Manage Replacement of Simulink Data Types in Generated Code”.

Configure Simulink function prototype to pass arguments by reference in generated C++ code

Starting in R2026a, you can configure the function prototype of scoped Simulink functions to pass arguments by reference in the generated C++ code.

In the C/C++ Function Interface dialog box for the Simulink Function or Function Caller block, under the **C/C++ Type Qualifier** column, assign input arguments to be passed by `Const Reference`, and output and in-out arguments to be passed by `Reference`. Alternatively, call the `setFunction` function with the desired function prototype.

For models with these settings that are exported to a previous release, the **C/C++ Type Qualifier** property for input arguments is set to `Auto` and for output and in-out arguments are set to `Pointer`.

For an example, see “Configure C++ Entry-Point Function Interfaces for Simulink Function and Function Caller Blocks”.

Change in location to Generate Halide code parameter

Starting in R2026a, the **Generate Halide code** configuration parameter is relocated from the **Code Generation** pane to the **Optimization** pane in the Configuration Parameters dialog box.

For more information on Halide, see “Speed Up Generated Code Execution with Halide Code”.

Improved casting compliance for MISRA C

Starting in R2026a, the code generator generates C code with improved compliance to the MISRA C *Essential type model* guidelines (Rules 10.1 to 10.8) related to casting operations.

In standard C, using types in expressions relies on the compiler for implicit promotions or conversions, which can lead to unexpected results or data loss. The MISRA guidelines address these problems by requiring such conversions to be explicit to provide safer and more predictable behavior. The generated code from Embedded Coder now introduces explicit casts based on the essential types,

as well as the Signed Type of Lowest Rank (STLR) and Unsigned Type of Lowest Rank (UTLR) concepts, to minimize implicit type promotions in accordance with MISRA recommendations.

When the **Casting mode** configuration parameter is set to `Standard Compliant`, the code generator adds explicit casts to expressions to ensure MISRA compliance and also casts constants explicitly to their appropriate STLR or UTLR type. For example:

```
/* Violation (without cast): */
u32Var = 1U << u32Var;

/* Corrected: */
u32Var = (uint32_T)1U << u32Var;
```

Explicit casts over constants, even with `U` suffix, are necessary to ensure the correct essential type and to prevent violations of MISRA rules.

Similarly, when casting composite expressions, the code generator introduces intermediate casts to ensure compliance. For example:

```
/* Compliant casting: */
u32Var = (uint32_T)(uint16_T)(u16Var | u16Var)
```

In this case, although the `(uint16_T)` cast may appear redundant, removing it would cause a violation of MISRA C Rule 10.8, which prohibits direct casting of a composite expression to a wider essential type.

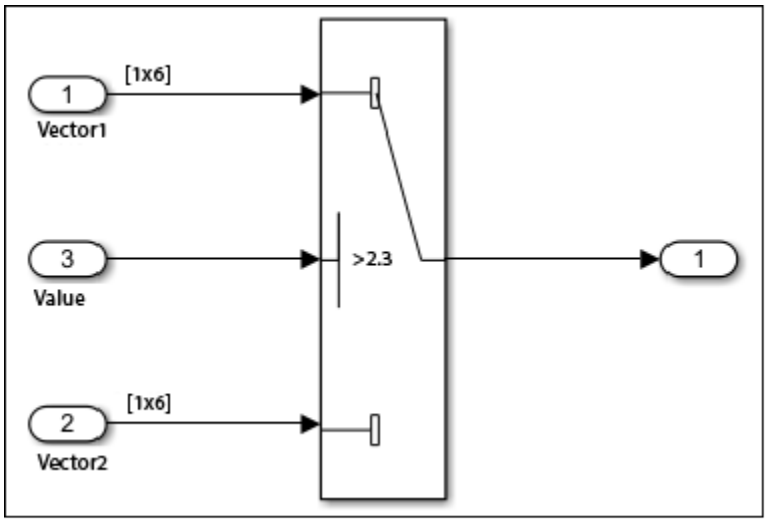
For more information, see “How Embedded Coder Resolves MISRA C 10.x Rule Violations”.

Improve execution speed in generated code through loop-invariant condition optimization

Starting in R2026a, the code generator optimizes `for` loops that contain `if-else` statements with conditions that evaluate to the same value in every iteration (loop-invariant conditions). By evaluating the condition once before entering a `for` loop and placing `for` loop inside separate `if` and `else` branches, the generated code eliminates redundant checks and improves execution speed. This optimization also supports `for` loops that contain `if` statements with loop-invariant conditions, including cases where the loop body contains global variables and function calls.

To enable this optimization, set model configuration parameter **Level** to `Maximum` and **Priority** to `Maximize execution speed`.

Consider the model `LoopUnswitching_IfInvariant`, which uses an `if` block to select between two input `1x6` vectors (`Vector1` and `Vector2`) based on whether the input value is greater than the scalar `2.3`.



Previously, the generated code for this model checked the condition `LoopUnswitching_IfInvariant_U.Value > 2.3` during every iteration of the loop. Since the value of `LoopUnswitching_IfInvariant_U.Value` remains constant throughout the loop, these repeated checks increase computational overhead, particularly in loops with a large number of iterations, reducing execution efficiency.

```
void LoopUnswitching_IfInvariant_step(void)
{
  int32_T i;
  for (i = 0; i < 6; i++) {
    if (LoopUnswitching_IfInvariant_U.Value > 2.3) {
      LoopUnswitching_IfInvariant_Y.Out1[i] =
        LoopUnswitching_IfInvariant_U.Vector1[i];
    } else {
      LoopUnswitching_IfInvariant_Y.Out1[i] =
        LoopUnswitching_IfInvariant_U.Vector2[i];
    }
  }
}
```

In R2026a, when the condition `LoopUnswitching_IfInvariant_U.Value > 2.3` is invariant across loop iterations, the code generator evaluates the condition once before entering the loop. The code generator then moves the condition outside the loop, generating separate `if` and `else` branches, each containing the loop body, thus reducing redundant checks.

```
void LoopUnswitching_IfInvariant_step(void)
{
  int32_T i;
  if (LoopUnswitching_IfInvariant_U.Value > 2.3) {
    for (i = 0; i < 6; i++) {
      LoopUnswitching_IfInvariant_Y.Out1[i] =
        LoopUnswitching_IfInvariant_U.Vector1[i];
    }
  } else {
    for (i = 0; i < 6; i++) {
      LoopUnswitching_IfInvariant_Y.Out1[i] =
        LoopUnswitching_IfInvariant_U.Vector2[i];
    }
```

```
  }
}
```

To understand how code generation optimizations can reduce redundant condition checks, see “Use Conditional Input Branch Execution”.

Specify code generation template tokens for model references

In R2026a, you can specify code generation template (CGT) tokens to generate file and function banners for model references. For more information, see “Generate Custom File and Function Banners for C/C++ Code”.

Add type name prefixes for C-style arrays in C++ bus data initialization code

In R2026a, if you set the model configuration parameters **Code interface packaging (component)** to `C++ class` and **Language standard** to `C++11 (ISO)` or later, the code generator now includes type name prefixes in bus data initialization code that uses C-style arrays.

Previously, the code generator only included type name prefixes if the **Static array container type** model configuration parameter is set to `std::array`. See “Add type name prefixes for C++ bus data initialization code” on page 5-12.

Functionality being removed or changed

Changes to Simulink Coder app context menu options

Behavior change

To access Embedded Coder app options in the new Simulink® Context Menu design, point to **Select Apps** and click the Embedded Coder button  . A new menu section containing options related to the Embedded Coder app appears in the menu. For more information about the new context menu design, see “New Simulink context menus prioritize frequently used functionality”.

Starting in 26a, these Embedded Coder app options are available in Simulink context menus:

- **Build**
- **Generate Code**
- **C/C++ Code generation settings**
- **Open Report**
- **Navigate to C/C++ Code**
- **Generate S-Function**

In 26a, these Embedded Coder options are removed from Simulink context menus:

- **Export Functions**
- **Build This Subsystem**

If you have an atomic subsystem from which you want to export functions or generate code, it is recommended that you first convert the subsystem to a referenced model. To do so, right-click the

subsystem and select **Convert to > Referenced Model**. You can still export functions and generate code for an individual atomic subsystem by passing the subsystem to the `slbuild` function.

Model configuration parameters renamed
Behavior change

Starting in R2026a, the model configuration parameters named **M-function** have been renamed to **MATLAB function**. The programmatic names of the parameters remain the same. For more information, see **MATLAB function**.

Additionally, the model configuration parameter named **Remove code that protects against division arithmetic exceptions** has been renamed to **Remove code that protects against integer division arithmetic exceptions**. The corresponding Model Advisor check has also been renamed to Check safety-related optimization settings for integer division arithmetic exceptions (Simulink Check).

Model configuration parameter Combine signal/state structures will be removed in a future release
Behavior change in future release

Model configuration parameter **Combine signal/state structures** (`CombinedSignalStatesStruct`) will be removed in a future release.

TLC code uses custom data type names instead of built-in data type names
Behavior change

If you specify custom data type names by using the model configuration parameter **Specify custom data type names**, the custom names propagate to TLC code, replacing built-in data type names.

Custom TLC code, which previously worked, might produce an error when these conditions apply:

- The model configuration parameter **Data type replacement** is set to `Coder typedefs`.
- The model configuration parameter **Specify custom data type names** is selected and custom data type names are specified.
- The custom TLC code uses hard-coded, built-in data type names in `switch` statement `case` labels or comparison operands.

This table shows how you can update a custom TLC code example.

Before Update	After Update

`%assign dataTypeName = LibBlockOutputSignalAliasedThruDataTypeName(0,"")`
`%switch dataTypeName`
`%case "uint8_T"`
`%case "uint16_T"`
`%break`
`%default`
`%% Something other than uint8, uint16`
`%<LibBlockReportFatalError(block, "Expected uint8 or uint16");>`
`%break`
`%endswitch`
	`%assign dataTypeId = LibBlockOutputSignalAliasedThruD`
	`%switch dataTypeId`
	`%case ::CompiledModel.tSS_UINT8`
	`%case ::CompiledModel.tSS_UINT16`
	`%break`
	`%default`
	`%% Something other than uint8, uint16`
	`%<LibBlockReportFatalError(block, "Expected uint8 or`
	`%break`
	`%endswitch`

For information about data type replacement, see “Manage Replacement of Simulink Data Types in Generated Code”.

Quick Start and Code Generation Advisor no longer support subsystems

The capability to configure subsystems by using the Embedded Coder Quick Start and the Code Generation Advisor is no longer supported. Instead:

- Use these tools to configure the model that contains the subsystem.
- Convert the subsystem to a referenced model and use these tools to configure the referenced model. For an example, see “Convert Subsystem to Referenced Model and Generate Code”.

For more information, see “Generate Code by Using the Quick Start Tool” and “Configure Model for Code Generation Objectives by Using Code Generation Advisor”.

Deployment

★Incremental generation of ASAP2 file

Starting in R2026a, Embedded Coder enables incremental generation of ASAP2 files. When you regenerate the ASAP2 file for a model, the file is updated only if the model includes changes that affect the ASAP2 file contents.

For models with referenced model hierarchy, only the sections of the ASAP2 file related to the updated referenced model are regenerated.

Export variant parameters from referenced models to ASAP2 file

Starting in R2026a, Embedded Coder enables you to configure and export variant parameters from referenced models to the generated ASAP2 file.

For more information, see “Generate Variant Coding Section in ASAP2 File”.

Removal of duplicate instances of computation method names in ASAP2 file

Starting in R2026a, computation method naming in the ASAP2 file follows the pattern `CM_`*`datatype_unit`*.

In earlier releases, computation method's naming used the model name as a prefix, which led to duplicate instances in ASAP2 files generated for models with a referenced model hierarchy.

For more information, see “Customize Computation Method Names”.

Implement service code interface library builds using C2000 Microcontroller Blockset and Raspberry Pi Blockset

You can configure a model to use a service code interface when you deploy C code as a component or subcomponent (see “Deploy Export-Function Component Configured for C Service Interface Code Generation”). Previously, if you tried to implement service code interface library builds by using the C2000™ Microcontroller Blockset or the Raspberry Pi® hardware support package:

- The build process failed to produce a library file for the component or subcomponent.
- Running a processor-in-the-loop (PIL) simulation did not produce valid results.

In R2026a, the listed limitations are removed for the C2000 Microcontroller Blockset and the Raspberry Pi Blockset.

Performance

Keyboard shortcuts for Code Replacement Tool

In R2026a, you can use keyboard shortcuts to interact with the Code Replacement Tool. Create, edit, validate, and register code replacement libraries by using the keyboard shortcuts. For more information, see “Code Replacement Tool Keyboard Shortcuts”.

Replace code for element-wise division operators

In R2026a, you can use custom implementation code to replace generated division operators that perform element-wise division on vector or matrix inputs. Replacing the default generated operators can help you to optimize the code for your target hardware. For more information, see “Code You Can Replace From Simulink Models”.

Replace code for RMS block and rms function

In R2026a, you can use custom code to replace generated code for the RMS block and the `rms` function. Replacing the default generated root mean square (RMS) code can optimize the code for your target hardware. For more information, see “Root Mean Square (RMS Block) Code Replacement” and “Root Mean Square Code Replacement”.

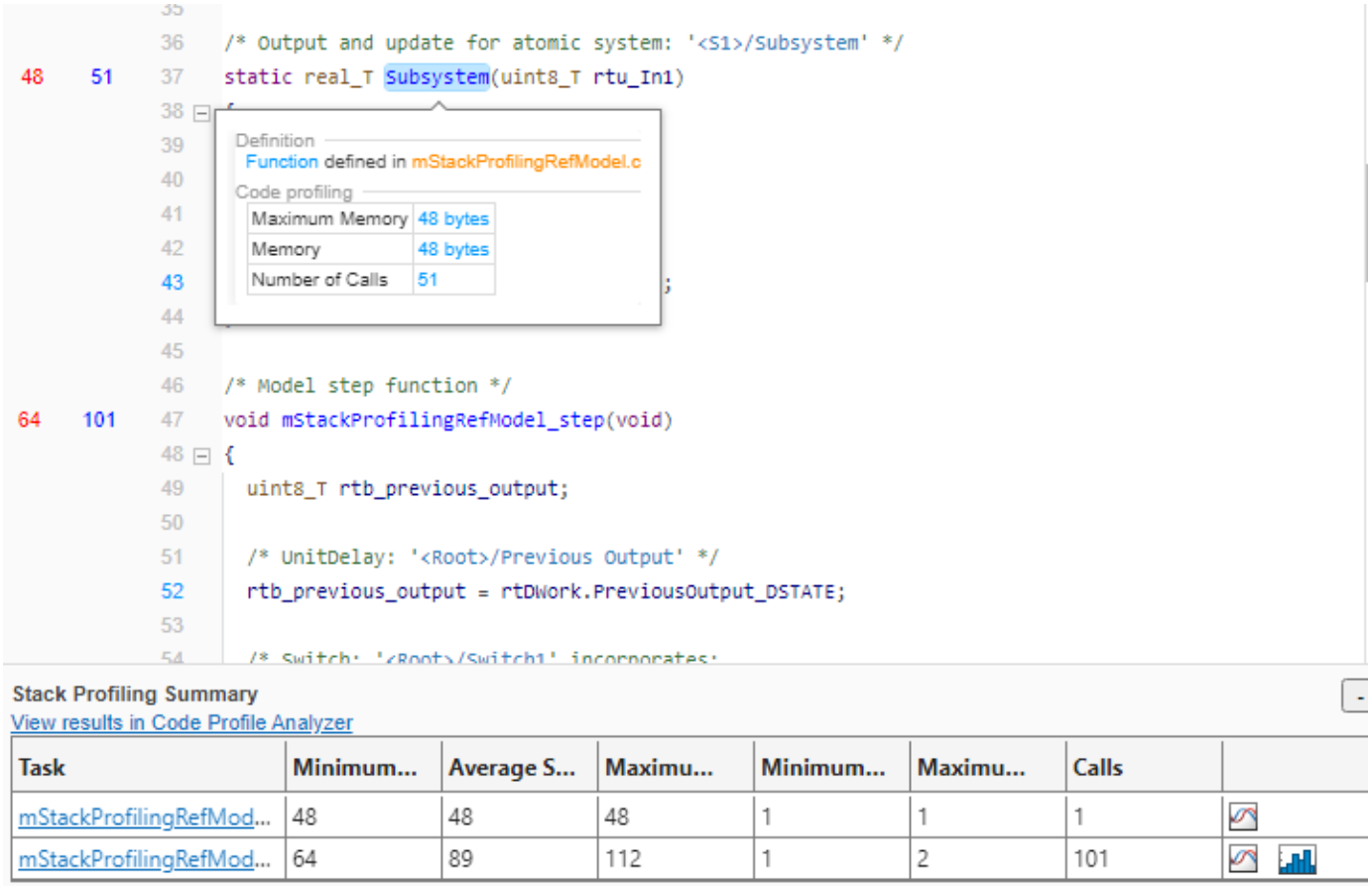
Replace code for min, max, sqrt, and sum functions with vector and matrix inputs

In R2026a, you can use custom implementation code to replace generated `min`, `max`, `sqrt`, and `sum` functions with vector or matrix inputs. Code replacement for the `sqrt` function now also supports complex inputs. Previously, code replacement for these functions supported only scalar inputs. Replacing the default generated functions can help you to optimize the code for your target hardware. Replacement of the `sum` function is supported only for Simulink Sum blocks that collapse the elements of the input. For more information, see “Code You Can Replace from MATLAB Code” and “Code You Can Replace From Simulink Models”.

Stack profiling in Code view

Starting in R2026a, you can view stack profiling information directly in the Code view to analyze memory usage in the generated code. When you run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations with stack profiling enabled, the Code view displays:

- A summary table of stack profiling results below the code
- Annotations next to function definitions showing self memory usage and number of calls to the function
- A tooltip with detailed stack metrics such as self memory usage, maximum memory usage, and number of calls to the function



For more information, see “View and Compare Stack Usage Metrics”.

Apply C restrict qualifier to function arguments

In R2026a, when you generate C code using the C99 language standard, you can apply the restrict qualifier to function arguments that are not aliased. Applying the restrict qualifier can enable compiler optimizations that improve the efficiency of the generated code.

To use the C restrict qualifier, select the new model configuration parameter, **Use C restrict qualifier**.

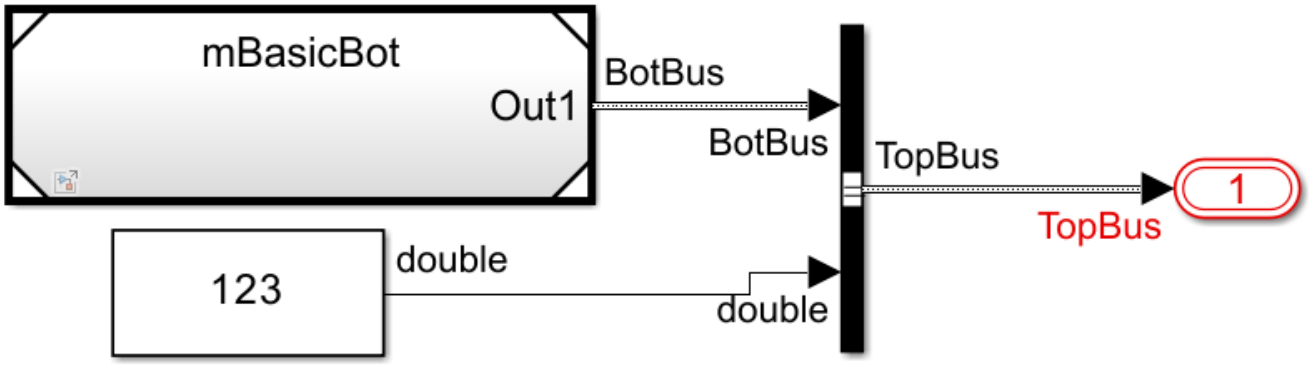
Data copy reduction for Bus Creator blocks from referenced models

Previously, for some modeling patterns containing a referenced model and Bus Creator blocks before and after the model reference boundary, the generated code contained an unnecessary data copy of the bus of the referenced model. In R2026a, the code generator optimizes the code for these modeling patterns by localizing the output and attempting to eliminate the unnecessary copy of the bus, which improves RAM consumption. To enable the optimization, set these model configuration parameters on the **Optimization** pane:

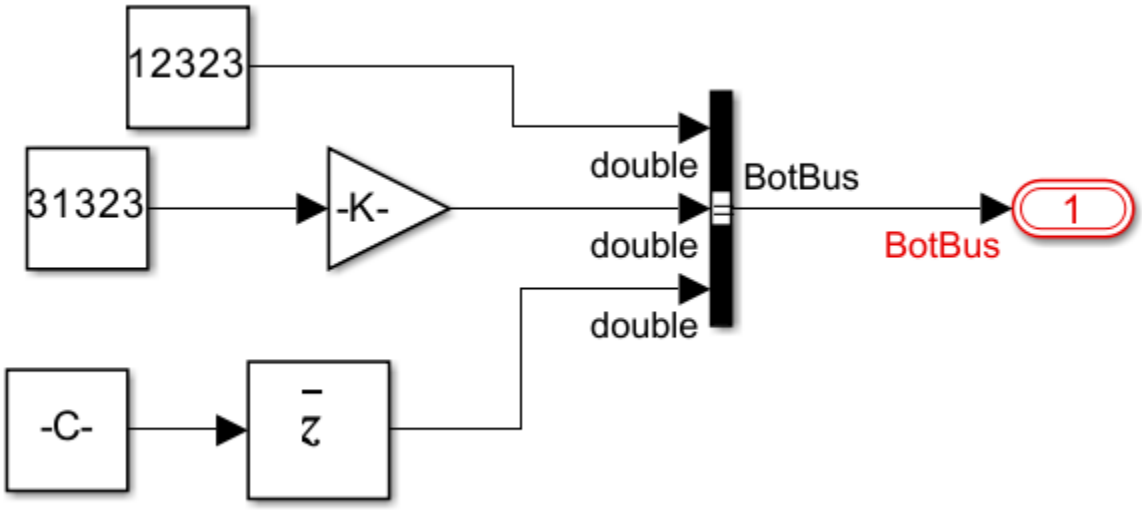
- **Level** — `Maximum`
- **Enable local block outputs** — select

- **Reuse local block outputs** — select

For example, consider the `mBasicTop` model.



The model contains a Model reference block that leads to a Bus Creator block. The output of the referenced model `mBasicBot` comes from another Bus Creator block.



Previously, the code generator produced this code for the top model:

```
void mBasicTop_step(void)
{
  mBasicBot(&rtDWork.Model, &(rtDWork.Model_InstanceData.rtdw));
  rtY.Out1.a = rtDWork.Model;
  rtY.Out1.a1 = 123.0;
}
```

The generated code unnecessarily copied the input bus `rtDWork.Model` to the output bus element `rtY.Out1.a`.

In R2026a, the code generator produces this code for the top model:

```
void mBasicTop_step(void)
{
  mBasicBot(&rtY.Out1.a, &(rtDWork.Model_InstanceData.rtdw));
  rtY.Out1.a1 = 123.0;
}
```

The code reuses the output bus element `rtY_Out1.a` to hold the input bus, eliminating the extra copy of the bus. For more information, see “Enable and Reuse Local Block Outputs in Generated Code”.

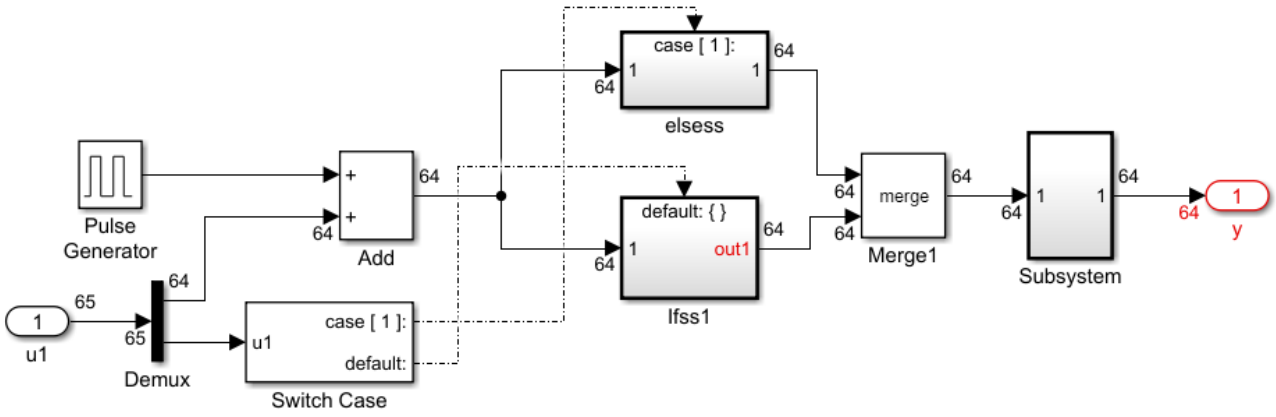
Reuse Merge block output buffers more effectively

In R2026a, the code generator more effectively reuses output buffers of Merge blocks that combine signals from:

- Each of the If Action Subsystem blocks that are controlled by an If block
- Each of the Switch Case Action Subsystem blocks that are controlled by a Switch Case block

Reusing the output buffer reduces RAM consumption and data copies. To enable this optimization, select the model configuration parameter **Signal storage reuse**.

For example, consider the `SimpleSwitchCase` model.



Previously, the code generator produced this code for the model:

```
/* Block signals (default storage) */
B_SimpleSwitchCase_T SimpleSwitchCase_B;

/* External outputs (root outports fed by signals with default storage) */
ExtY_SimpleSwitchCase_T SimpleSwitchCase_Y;

...

  for (i = 0; i < 64; i++) {
    SimpleSwitchCase_Y.y[i] = 22.0 * SimpleSwitchCase_B.Add[i] * -3.0;
  }

...

  for (i = 0; i < 64; i++) {
```

```
    SimpleSwitchCase_Y.y[i] = -3.0 * SimpleSwitchCase_B.Add[i];
  }
```

The code unnecessarily creates the buffer `SimpleSwitchCase_B.Add` for the output of the Add block.

In R2026a, the code generator produces this code for the model:

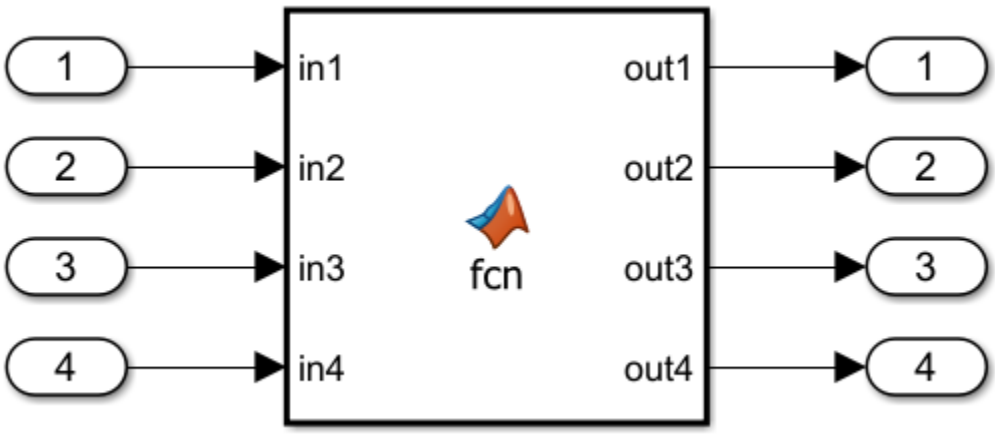
```
/* External outputs (root outports fed by signals with default storage) */
ExtY_SimpleSwitchCase_T SimpleSwitchCase_Y;

  for (i = 0; i < 64; i++) {
    SimpleSwitchCase_Y.y[i] = 22.0 * SimpleSwitchCase_Y.y[i] * -3.0;
  }

...

  for (i = 0; i < 64; i++) {
    SimpleSwitchCase_Y.y[i] *= -3.0;
  }
```

The code reuses the output `SimpleSwitchCase_Y.y` with `SimpleSwitchCase_B.Add` and does not generate the buffer `SimpleSwitchCase_B.Add`. For more information, see **Eliminate superfluous local variables (Expression folding)**.

Fold expressions efficiently when reusing buffers

In R2026a, when you enable buffer reuse and expression folding, if the optimizations overlap such that buffer reuse would prevent expression folding, the code generator analyzes both optimizations and uses expression folding instead of buffer reuse when it improves the efficiency of the generated code.

For example, this model contains a MATLAB Function block that performs several computations.



`fcn.m`

```
function [out1,out2,out3,out4] = fcn(in1,in2,in3,in4)

toRename = in1*in2*in3+in4;

a = toRename;
```

```
doesnotFold = toRename*in2+in4+in3;

toRename = in1*in2+in4*in3;

out1 = doesnotFold*doesnotFold;

useOfA = a * in1 * in2 + in4;

out2 = toRename*toRename*in1+in2*in2+toRename;
out3 = in1*toRename+in2*toRename+in3*toRename;
out4 = useOfA*in1+useOfA*in2+useOfA;

end
```

In R2025b, when the model uses buffer reuse and expression folding, Embedded Coder generates the following code from the MATLAB function block. The code contains local variable `a` and an expression that copies `toRename` to `a`.

```
/* Output and update for atomic system: '<Root>/MATLAB Function1' */
void ExpFoldingWithB_MATLABFunction1(real_T rtu_in1, real_T rtu_in2, real_T
  rtu_in3, real_T rtu_in4, real_T *rty_out1, real_T *rty_out2, real_T *rty_out3,
  real_T *rty_out4)
{
  real_T a;
  real_T doesnotFold;
  real_T toRename;
  real_T toRename_tmp;
  toRename_tmp = rtu_in1 * rtu_in2;
  toRename = toRename_tmp * rtu_in3 + rtu_in4;
  a = toRename;
  doesnotFold = (toRename * rtu_in2 + rtu_in4) + rtu_in3;
  toRename = rtu_in4 * rtu_in3 + toRename_tmp;
  *rty_out1 = doesnotFold * doesnotFold;
  a = a * rtu_in1 * rtu_in2 + rtu_in4;
  *rty_out2 = (toRename * toRename * rtu_in1 + rtu_in2 * rtu_in2) + toRename;
  *rty_out3 = (rtu_in1 * toRename + rtu_in2 * toRename) + rtu_in3 * toRename;
  *rty_out4 = (a * rtu_in1 + a * rtu_in2) + a;
}
```

In R2026a, Embedded Coder generates the following code from the MATLAB function block. The code eliminates the local variable `a` and the line that copied `toRename` to `a`.

```
/* Output and update for atomic system: '<Root>/MATLAB Function1' */
void ExpFoldingWithB_MATLABFunction1(real_T rtu_in1, real_T rtu_in2, real_T
  rtu_in3, real_T rtu_in4, real_T *rty_out1, real_T *rty_out2, real_T *rty_out3,
  real_T *rty_out4)
{
  real_T doesnotFold;
  real_T toRename;
  real_T toRename_0;
  toRename_0 = rtu_in1 * rtu_in2;
  toRename = toRename_0 * rtu_in3 + rtu_in4;
  doesnotFold = (toRename * rtu_in2 + rtu_in4) + rtu_in3;
  toRename_0 += rtu_in4 * rtu_in3;
  *rty_out1 = doesnotFold * doesnotFold;
  toRename = toRename * rtu_in1 * rtu_in2 + rtu_in4;
  *rty_out2 = (toRename_0 * toRename_0 * rtu_in1 + rtu_in2 * rtu_in2) +
    toRename_0;
  *rty_out3 = (rtu_in1 * toRename_0 + rtu_in2 * toRename_0) + rtu_in3 *
    toRename_0;
  *rty_out4 = (toRename * rtu_in1 + toRename * rtu_in2) + toRename;
}
```

For more information, see “Fold Expressions”.

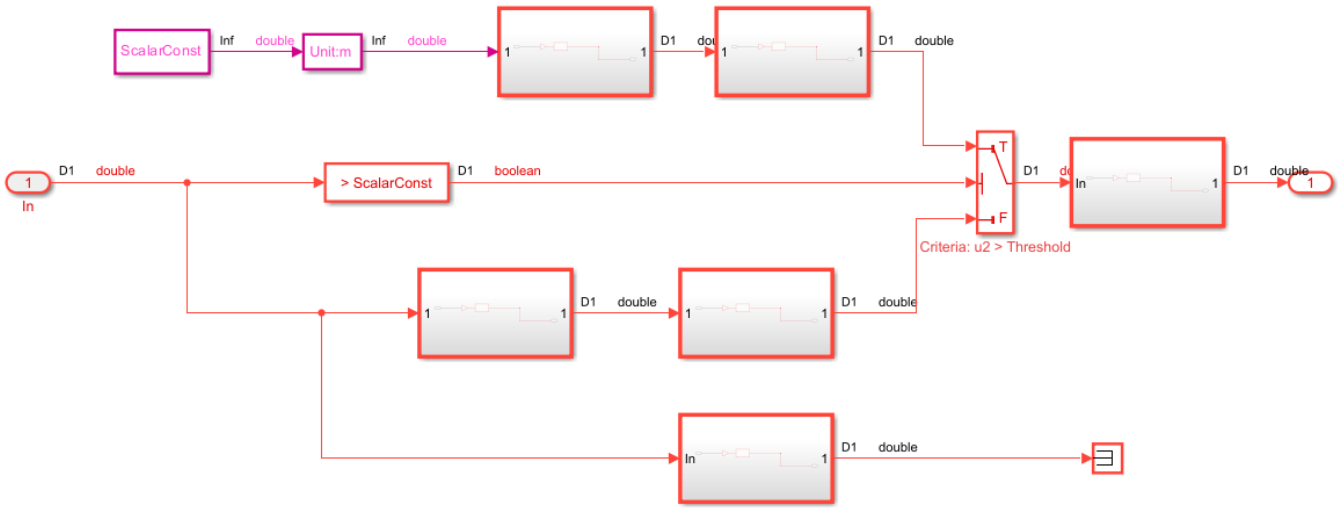
Extend memory for code profiling in XCP external mode simulations on Linux target hardware

To minimize the potential for data loss while profiling generated code during XCP-based external mode simulations on Linux target hardware, enable dynamic memory allocation using the Simulink parameter `CodeProfilingXCPDynamicMemory`. For more information, see “Specify Memory Allocation for Code Execution Profiling”.

Minimize computations for conditional input branches

In R2026a, the code generator minimizes the computation of intermediate results for conditional input branches by folding the computations into a single expression. Expression folding improves the efficiency of the generated code. To enable this optimization, select the model configuration parameter **Eliminate superfluous local variables (Expression folding)**.

For example, consider the `BranchFolding` model.



Previously, the code generator produced this code for the model:

```
void BranchFolding_step(void)
{
  real_T rtb_UnitConversion_h;
  real_T rtb_UnitConversion_m3;
  rtb_UnitConversion_m3 = Subsystem(36000.0);
  rtb_UnitConversion_m3 = Subsystem(rtb_UnitConversion_m3);
  rtb_UnitConversion_h = Subsystem(rtU.In);
  rtb_UnitConversion_h = Subsystem(rtb_UnitConversion_h);
  if (rtU.In > 36000.0) {
    rtb_UnitConversion_h = rtb_UnitConversion_m3;
  }

  rtY.Out1 = Subsystem(rtb_UnitConversion_h);
  Subsystem(rtU.In);
}
```

The code unnecessarily calls the generated `Subsystem` function for each instance of the subsystem that occurs before the Switch block for a total of six calls to the function.

In R2026a, the code generator produces this code for the model:

```
void BranchFolding_step(void)
{
  real_T tmp;
  if (rtU.In > 36000.0) {
    tmp = Subsystem( Subsystem(36000.0));
  } else {
    tmp = Subsystem( Subsystem(rtU.In));
  }

  rtY.Out1 = Subsystem(tmp);
}
```

The code folds the calls to the `Subsystem` function into the conditional paths so that each execution path calls the function only twice. The code also removes the dead `Subsystem` call from the end of the step function. For more information, see “Fold Expressions”.

Unify analysis for buffer reuse of different sizes and dimensions

In R2026a, you can generate efficient code that reuses more buffers of different sizes by enabling the model configuration parameters **Reuse buffers of different sizes and dimensions** and **Unify buffer reuse candidates**, which enables the code generator to perform unified analysis to identify more buffer reuse candidates. Previously, you could not enable both optimizations at the same time. For more information, see “Generate Efficient Code Using Unified Analysis”.

Verification

★Simplified PIL connectivity for Linux-based target hardware

For Linux-based target hardware, you can now enable processor-in-the-loop (PIL) connectivity by using `target.OperatingSystem` and `target.Support` objects. This approach simplifies the setup process when you have multiple boards that run the Linux operating system. For more information, see “Set Up PIL Connectivity for Linux-Based Target Hardware”.

Prior to R2026a, to set up PIL connectivity, you specified PIL dependencies (for example, the communication protocol, communication interface, and profiling timer) as properties of a `target.Board` object.

Enhanced SIL/PIL data logging for referenced models

Previously, for referenced models in a top model or Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, data logging was supported only if the **Total number of instances allowed per top model** configuration parameter for each model in the hierarchy was set to `Multiple`. Now, if the value of the configuration parameter is `One` or `Multiple`, you can log signals and discrete states and stream the signal and state data to the Simulation Data Inspector. The simulation stores logged data in `logsout` and `xout`. For more information, see “SIL/PIL Manager Verification Workflow”.

SIL/PIL code generation assumption checks for data types from external code

You can use data types from external code in your model (see “Control File Placement of Custom Data Types”). R2026a provides code generation assumption checks to verify that external code data type definitions are consistent with Simulink data type definitions for enumerations, aliases, and buses.

For more information, see “Check Code Generation Assumptions” and “Assumption Checks for Data Types from External Code”.

SIL/PIL support for AUTOSAR per-instance memory that accesses NVRAM

For an AUTOSAR model that contains per-instance memory data stores mapped to NVRAM, you can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation if you use tunable model workspace parameters to set the initial values of the data stores. For more information, see “SIL/PIL Requirement for NVRAM-Mapped Per-Instance Memory”.

⚠Functionality being removed or changed

SIL/PIL simulations no longer support BullseyeCoverage

Errors

Software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations no longer support the third-party tool BullseyeCoverage. For information about analyzing code coverage during a SIL or PIL simulation, see “Code Coverage”.

Use of SIL and PIL blocks for testing subsystem code is not recommended (May 2026)
Still runs

Starting in R2026a, the SIL and PIL block approach for testing subsystem code is not recommended. Test code generated for subsystems in the context of the parent model or convert the subsystem to a Model block and apply the reference model SIL/PIL approach. For more information, see “Choose a SIL or PIL Approach”.

Hardware Support

★ Embedded Coder Support Package for Renesas RA Microcontrollers: Generate, build, and deploy Simulink models on Renesas RA series microcontrollers

The Embedded Coder Support Package for Renesas® RA Microcontrollers is available starting in R2026a. This support package enables you to design real-time applications for the Renesas 32-bit RA family of microcontrollers using Simulink and generate processor optimized code that you can compile and execute on RA based boards. The support package also supports integration with third-party software such as RA Smart Configurator for advanced configuration and development.

The first release of this support package includes:

- Simulink block support — Use Analog To Digital Converter, Digital Port Pins Read, Digital Port Pins Write, Hardware Interrupt, PWM Output, and Three Phase PWM Simulink blocks to model and implement peripheral interactions.
- Code generation and deployment — Generate, build, and deploy code directly from Simulink to Renesas RA based board for rapid prototyping and testing.
- Peripheral configuration — Configure peripherals through RA smart configurator for flexible application development.
- Processor-in-the-loop (PIL) simulation, code verification, and profiling — Test generated code with PIL simulation to verify functionality and behavior on the target hardware. Perform code verification and profiling to assess execution performance and correctness.
- Monitor & Tune mode support — Monitor signals in real time and tune parameters while the algorithm runs on the hardware for efficient validation and optimization.
- Examples that demonstrate the capabilities of the support package:
  - “Getting Started with Renesas RA Microcontroller Boards”
  - “Model-Based BLDC Motor Control on Renesas RA6T2 Microcontroller Using Simulink”
  - “Using Analog to Digital Converter Block with Renesas RA Microcontrollers”
  - “Control Onboard LED Brightness Using PWM Blocks on Renesas RA Microcontrollers”
  - “Test Generated Code with PIL Simulations on Renesas RA Microcontrollers”
  - “Execution Time Profiling of Field-Oriented Control (FOC) Algorithm Using PIL on Renesas RA Microcontrollers”

For more information, see “Renesas RA Microcontrollers”.

★ Embedded Coder Support Package for Renesas RH850 Microcontrollers: Generate, build, and deploy Simulink models on Renesas RH850 U2A microcontrollers

The Embedded Coder Support Package for Renesas RH850 Microcontrollers is available starting in R2026a. This support package enables you to design real-time applications for the Renesas 32-bit RH850 U2A family of microcontrollers using Simulink and generate processor optimized code that you can compile and execute on RH850 U2A based boards. The support package also supports

integration with third-party software such as C Compiler for RH850 (CC-RH) for development workflows.

The first release of this support package includes:

- Simulink block support — Use MCAL ADC, MCAL DIO Channel Group Read, MCAL DIO Channel Group Write, MCAL DIO Channel Read, MCAL DIO Channel Write, MCAL DIO Port Pins Read, MCAL DIO Port Pins Write, Hardware Interrupt, and TSG3 Output Simulink blocks to model and implement peripheral interactions.
- Code generation and deployment — Generate, build, and deploy code directly from Simulink to RH850 U2A based board for rapid prototyping and testing.
- Peripheral configuration — Configure MCAL modules such as ADC, DIO, Mcu, and Port using third-party AUTOSAR MCAL configuration tools such as DaVinci Configurator. Import the Configuration Description File (CDF) through Simulink model configuration parameters to enable flexible application development.
- Processor-in-the-loop (PIL) simulation, code verification, and profiling — Test generated code with PIL simulation to verify functionality and behavior on the target hardware. Perform code verification and profiling to assess execution performance and correctness.
- Monitor & Tune mode support — Monitor signals in real time and tune parameters while the algorithm runs on the hardware for efficient validation and optimization.
- Examples that demonstrate the capabilities of the support package:
  - “Getting Started with MCAL DIO Blocks on Renesas RH850 Microcontrollers”
  - “Model-Based BLDC Motor Control on Renesas RH850 U2A Microcontroller Using Simulink”
  - “Using MCAL ADC and Hardware Interrupt Blocks with Renesas RH850 Microcontrollers”
  - “Control Onboard LED7 Brightness Using TSG3 Blocks on Renesas RH850 Microcontrollers”
  - “Test Generated Code with PIL Simulations on Renesas RH850 Microcontrollers”
  - “Code Verification and Profiling Using PIL Testing on Renesas RH850 Microcontrollers”

For more information, see “Renesas RH850 Microcontrollers”.

ARM Cortex Hardware: Support package external mode simulation with TCP/IP protocol is being removed

Support for the TCP/IP protocol will be removed for “External Mode Simulation with TCP/IP” in a future release. For the Embedded Coder Support Package for ARM® Cortex®-A Processors, Embedded Coder Support Package for ARM Cortex-M Processors, and Embedded Coder Support Package for ARM Cortex-R Processors, use the XCP protocol for simulation as described in “External Mode Simulation by Using XCP Communication”.

Infineon AURIX TC3x Microcontrollers: Support for iLLD 1.20.1

You can now use the Infineon low-level driver (iLLD) 1.20.1 with Embedded Coder Support Package for Infineon® AURIX™ TC3x Microcontrollers.

During the hardware setup process, you can choose to automatically or manually download and install the iLLD software. For details, see “Hardware Setup for Infineon AURIX Microcontrollers”.

Infineon AURIX TC3x Microcontrollers: Support for GCC for AURIX TriCore based on AURIX Development Studio 11.3.1

You can now compile and run code using the GCC for AURIX TriCore® based on AURIX Development Studio 11.3.1 with the Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC3x Microcontrollers: Support for Infineon TAS 8.3.0

You can now connect host computer to Infineon AURIX hardware boards using Infineon tool access socket (TAS) 8.3.0 with Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Infineon TAS is the successor of the previously used device access server (DAS). For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC3x Microcontrollers: Support for Infineon AURIX TC33x and TC36x hardware boards

Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers now supports TC33x and TC36x **Device Series** of Infineon AURIX hardware boards.

You can now:

- Use Digital Port Read, Digital Port Write, PWM, EVADC, SENT, QSPI Controller, QSPI Peripheral, Encoder, MCAN Transmit, MCAN Receive, and Hardware Interrupt blocks to design single-core application models by using the TriCore 0 processing unit of the TC33x series hardware boards (Infineon AURIX TC334 Triboard, Infineon AURIX TC337 Triboard, and Custom board). For more information, see “Monolithic Modeling for Infineon AURIX Microcontrollers”.
- Use Digital Port Read, Digital Port Write, PWM, EDSADC, EVADC, SENT, QSPI Controller, QSPI Peripheral, Encoder, MCAN Transmit, MCAN Receive, and Hardware Interrupt blocks to design single-core and multicore application models by using TriCore 0 and TriCore 1 processing units of the TC36x series hardware boards (Infineon AURIX TC367 Triboard and Custom board). For more information on these modeling workflows, see “Monolithic Modeling for Infineon AURIX Microcontrollers” and “SoC-Based Multicore Modeling Workflow for Infineon AURIX Microcontrollers”.
- Run processor-in-the-loop (PIL) simulation in the serial communication mode. For more information, see “Processor-in-the-Loop (PIL) Simulation”.
- Monitor signals and tune parameters in the external mode simulation. For more information, see “External Mode Simulation for Signal Monitoring and Parameter Tuning”.

Infineon AURIX TC3x Microcontrollers: Support for custom start-up options

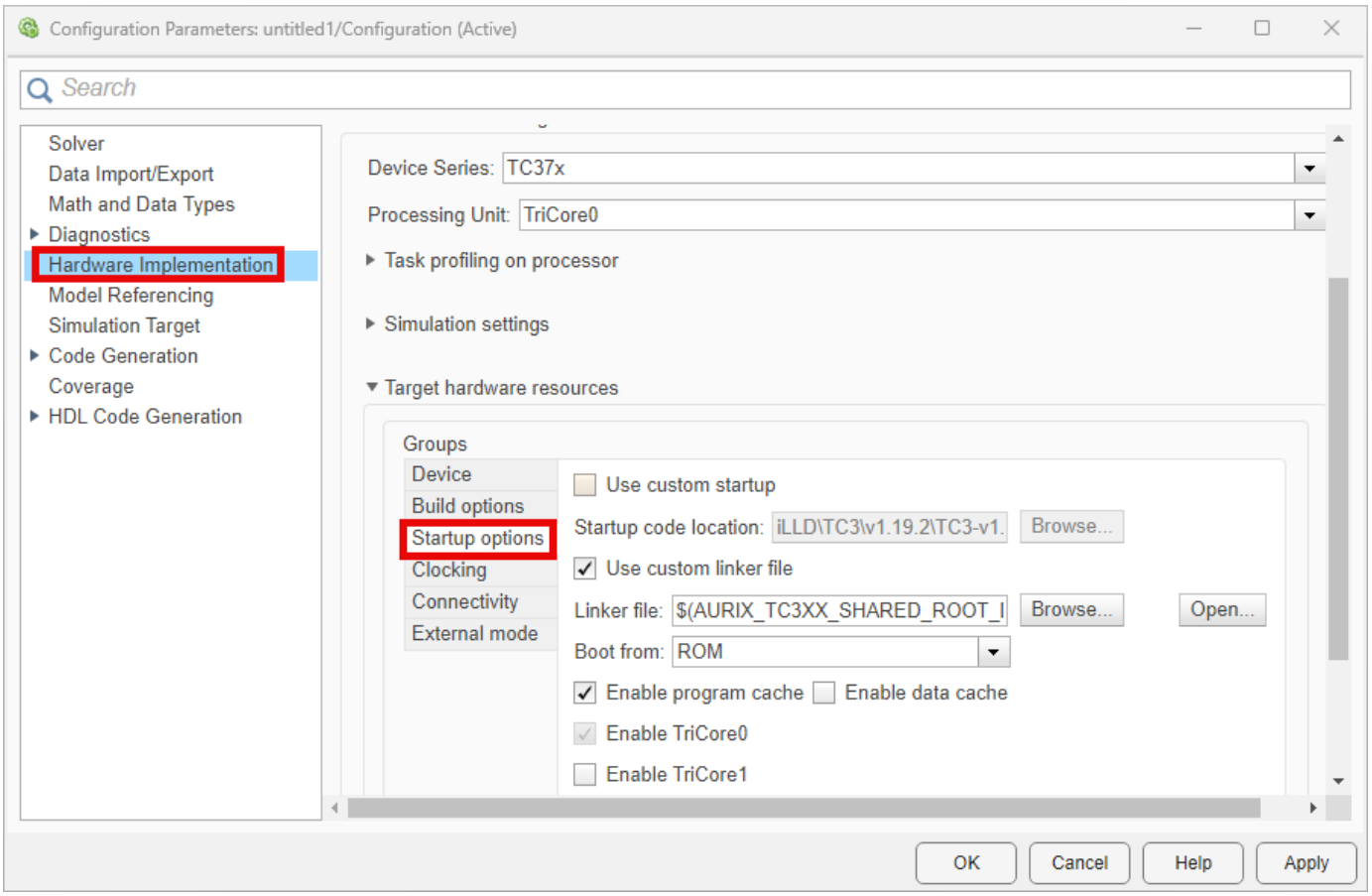
Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers now supports custom start-up options, which you can configure in the Configuration Parameters dialog box by navigating to **Hardware Implementation > Target hardware resources > Startup options**.

- **Use custom startup** — Select this option if you have your own custom startup code, which you can specify in the **Startup code location** parameter.
- **Startup code location** — For each family of the Infineon AURIX processors selected under **Target hardware resources**, a startup code file is selected automatically. You can provide your own custom startup code by clicking the **Browse** button.

In addition, several existing parameters have been moved to the **Startup options** tab:

- **Use custom linker file**, **Linker file**, and **Boot From** parameters were previously in the **Build options** tab.
- **Enable program cache**, **Enable data cache**, and **Enable TriCore#** parameters were previously in the **Device** tab.

For more information on these parameters, see “Model Configuration Parameters”.



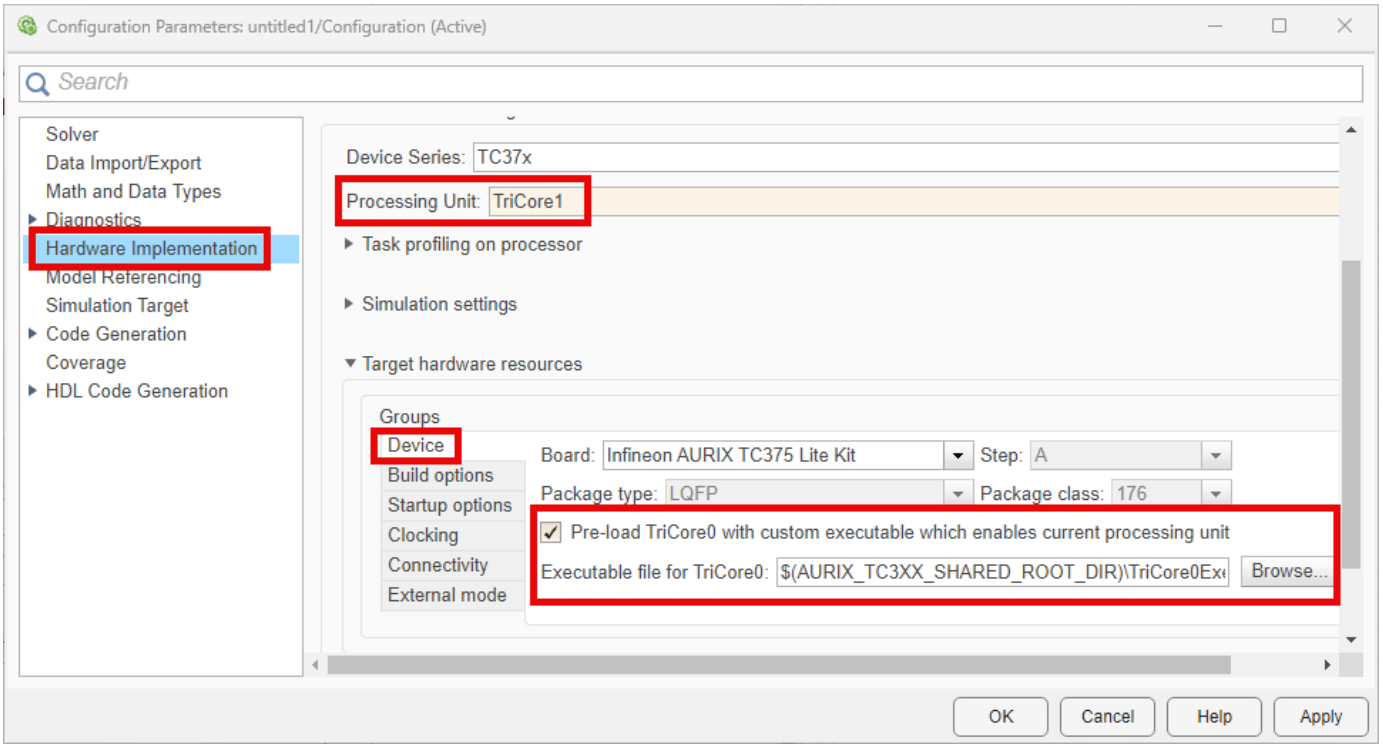
Infineon AURIX TC3x Microcontrollers: Support for custom executable file options for TriCore 0

Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers now supports custom executable file options for the TriCore 0 processing unit, which you can configure in the Configuration Parameters dialog box by navigating to **Hardware Implementation** > **Target hardware resources** > **Device**.

- **Pre-load TriCore0 with custom executable which enables current processing unit** — Select this option to pre-load the TriCore 0 processing unit with a custom executable file to enable the participating processing unit.

  TriCore 0 is the principal processing unit that handles system initialization, boot processes, and critical control tasks in the AURIX microcontrollers. If you design application model to run on the processing units other than TriCore 0, then you must initialize these processing units by deploying an executable code of the TriCore 0 processing unit.
- **Executable file for TriCore0** — Depending on the **Processing Unit** parameter value, a default executable code file is selected automatically. You can provide your own custom executable file by clicking the **Browse** button.

This figure shows custom executable option enabled for the TriCore 1 processing unit.



Infineon AURIX TC3x Microcontrollers: Enhancements to serial communication parameters

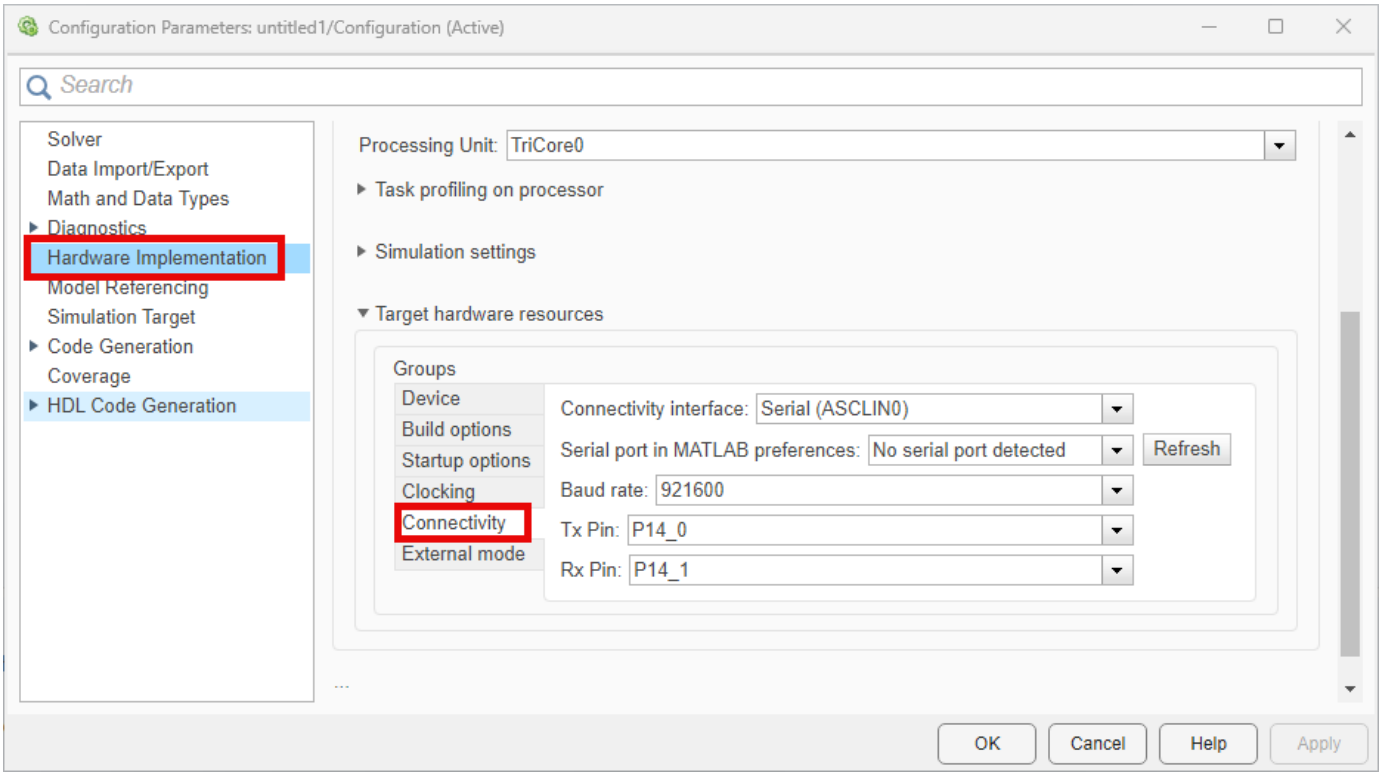
Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers now includes these enhancements to the serial communication parameters:

- You can configure the asynchronous synchronous interface (ASCLIN) pins for serial communication in processor-in the-loop and external mode simulation workflows. The available options range from `Serial (ASCLIN0)` to `Serial (ASCLIN23)` depending on the **Device Series** parameter value.

  Previously, `Serial (ASCLIN0)` and `DAS` were the only available options for the **Connectivity interface** parameter.

- The **Port** parameter has been renamed to **Serial port in MATLAB preferences**.
- You can specify the pin number for data transmission in the **Tx Pin** parameter when using the serial communication port for processor-in-the-loop (PIL) and external mode simulations.
- You can specify the pin number for data reception in the **Rx Pin** parameter when using the serial communication port for PIL and external mode simulations.

For more information on these connectivity parameters, see “Model Configuration Parameters”.



Infineon AURIX TC3x Microcontrollers: New examples that demonstrate the capabilities of TC3x Microcontrollers

Use the following examples to explore capabilities of Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

- “Code Verification Using PIL Simulation on Infineon AURIX TC3x Microcontrollers for ISO 26262 Certification” — This example shows how to configure the IEC Certification Kit example model for Infineon AURIX TC3x hardware board and use Simulink Test Manager for source code verification using PIL simulations.
- “AUTOSAR-Compliant Field-Oriented Control for PMSM on Infineon AURIX TC3x Microcontrollers” — This example shows how to integrate the AUTOSAR Microcontroller abstraction layer (MCAL) components and complex device drivers (CDD) with Simulink to generate production-ready code. You deploy this code on the Infineon AURIX TC3x microcontroller to implement an AUTOSAR-compliant field-oriented control (FOC) algorithm for a PMSM.

Infineon AURIX TC4x Microcontrollers: Support for iLLD 2.3.0

You can now use Infineon low-level driver (iLLD) 2.3.0 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC4x Microcontrollers: Support for Green Hills MULTI 2024.1.4

You can now use Green Hills® MULTI® 2024.1.4 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC4x Microcontrollers: Support for HighTec LLVM 9.1.2

You can now use HighTec LLVM 9.1.2 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC4x Microcontrollers: Support for GCC for AURIX TriCore based on AURIX Development Studio 11.3.1

You can now compile and run code on the TriCore processing units using the GCC for AURIX TriCore based on AURIX Development Studio 11.3.1 with the Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC4x Microcontrollers: Support for Synopsys MetaWare 2.1(2024.06)

You can now use the Synopsys® MetaWare for Infineon AURIX TC4x 2.1(2024.06) with the Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

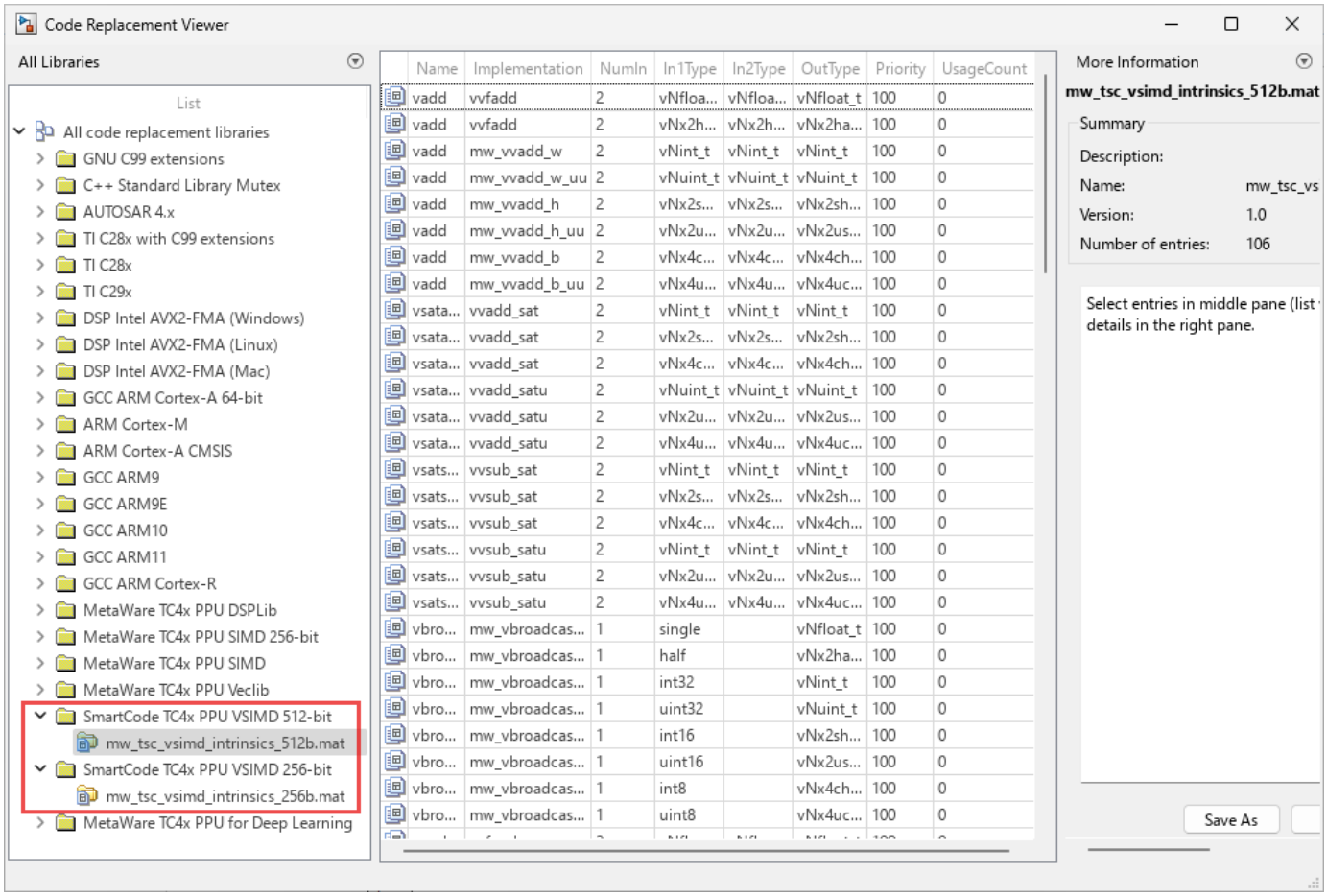
Infineon AURIX TC4x Microcontrollers: Support for TASKING SmartCode 10.3r1

You can now use TASKING® Smartcode 10.3r1 to compile and run code for the parallel processing unit (PPU) and TriCore processing units. Previously, this compiler has been supported for the TriCore

processing units. For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

You can now:

- Select `TASKING SmartCode for PPU` compiler option for the PPU. In the Configuration Parameters dialog box, select **Code Generation**. Under **Build process**, set **Toolchain** to `TASKING SmartCode for PPU`.
- Select these new code replacement libraries (CRL) to generate optimized code for the PPU based application models using the TASKING Smartcode compiler. For more information, see “Parallel Processing Unit for Optimized Code Generation”.



Infineon AURIX TC4x Microcontrollers: Support for Infineon TAS 8.3.0

You can now connect host computer to InfineonAURIX hardware boards using Infineon tool access socket (TAS) 8.3.0 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

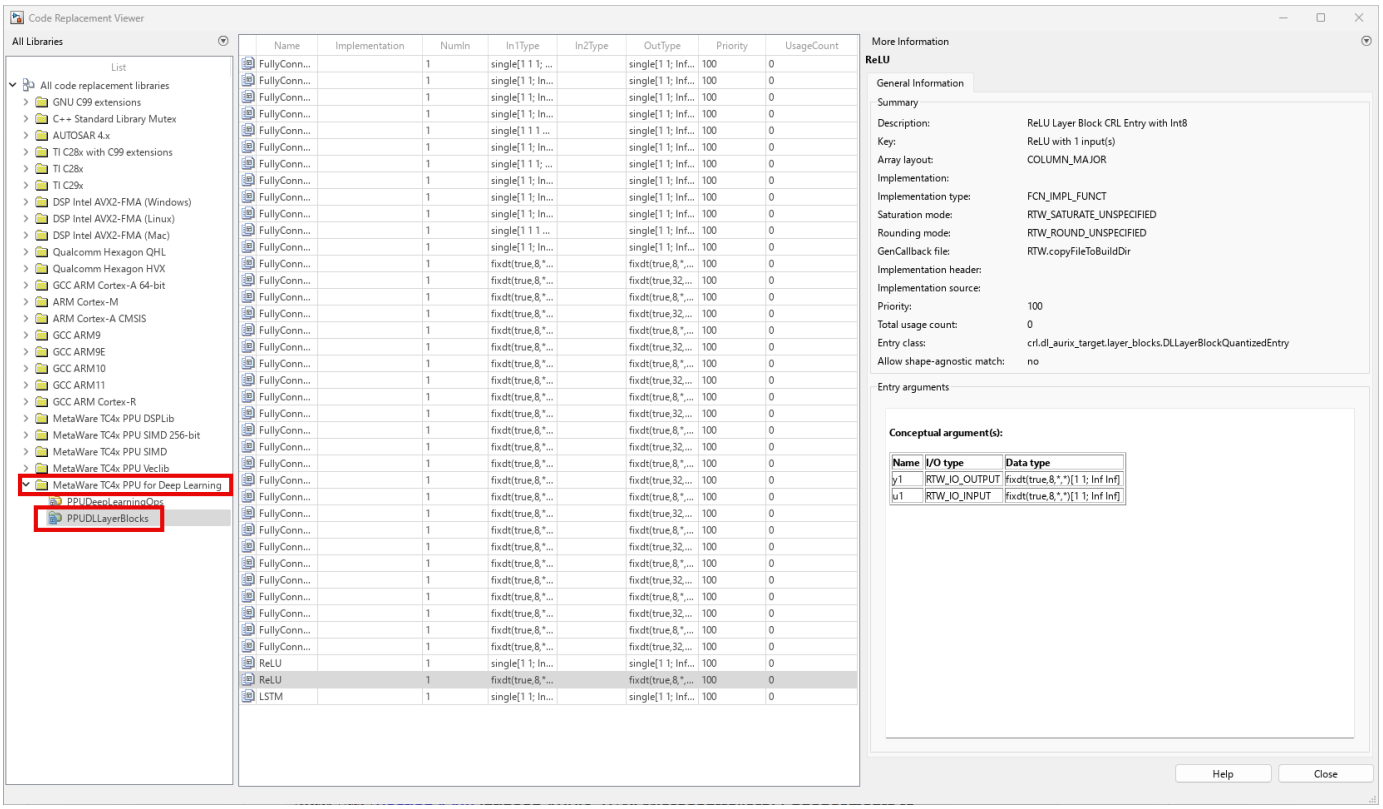
Infineon TAS is the successor of the previously used device access server (DAS). For more information on the supported third-party tools, see “Supported Hardware and Required Software”.

Infineon AURIX TC4x Microcontrollers: Enhancements to deep learning code replacement library

You can now generate optimized code in these deep learning layer Simulink blocks using the newly added `PPUDLLayerBlocks` code replacement library (CRL) table entry.

- Fully Connected Layer, LSTM Layer, and ReLU Layer blocks in the deep learning networks.
- Fully Connected Layer and ReLU Layer blocks in the quantized neural networks.

For more information on CRLs of PPU, see “Parallel Processing Unit for Optimized Code Generation”.



Infineon AURIX TC4x Microcontrollers: Enhancements to serial communication parameters

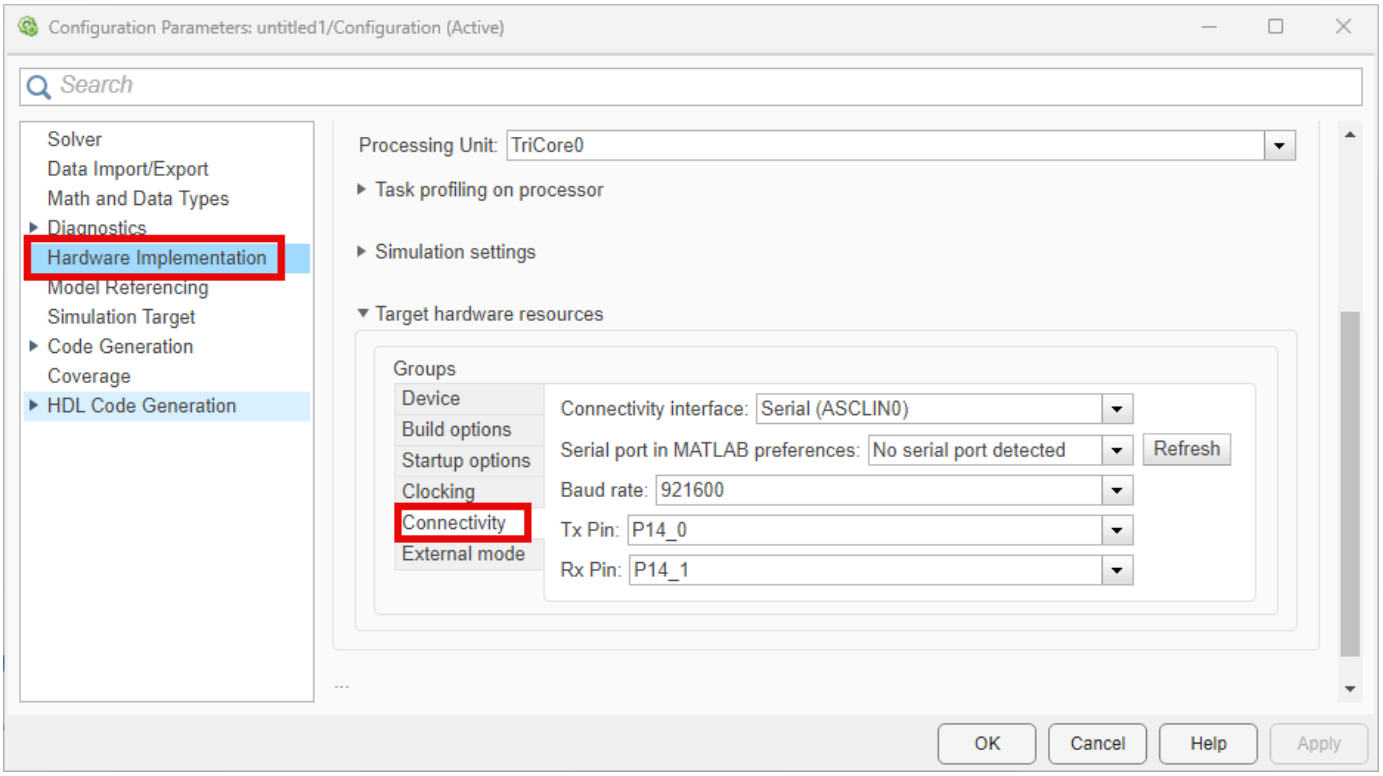
Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now includes these enhancements to the serial communication parameters:

- You can configure asynchronous synchronous interface (ASCLIN) pins for serial communication in processor-in the-loop and external mode simulation workflows. The available options range from `Serial (ASCLIN0)` to `Serial (ASCLIN27)` depending on the **Device Series** parameter value.

  Previously, `Serial (ASCLIN0)` and `DAS` were the only available options for the **Connectivity interface** parameter.
- The **Port** parameter has been renamed to **Serial port in MATLAB preferences**.

- You can specify the pin number for data transmission in the **Tx Pin** parameter when using the serial communication port for processor-in-the-loop (PIL) and external mode simulations.
- You can specify the pin number for data reception in the **Rx Pin** parameter when using the serial communication port for PIL and external mode simulations.

For more information on these connectivity parameters, see “Model Configuration Parameters”.



Infineon AURIX TC4x Microcontrollers: Support for custom start-up options

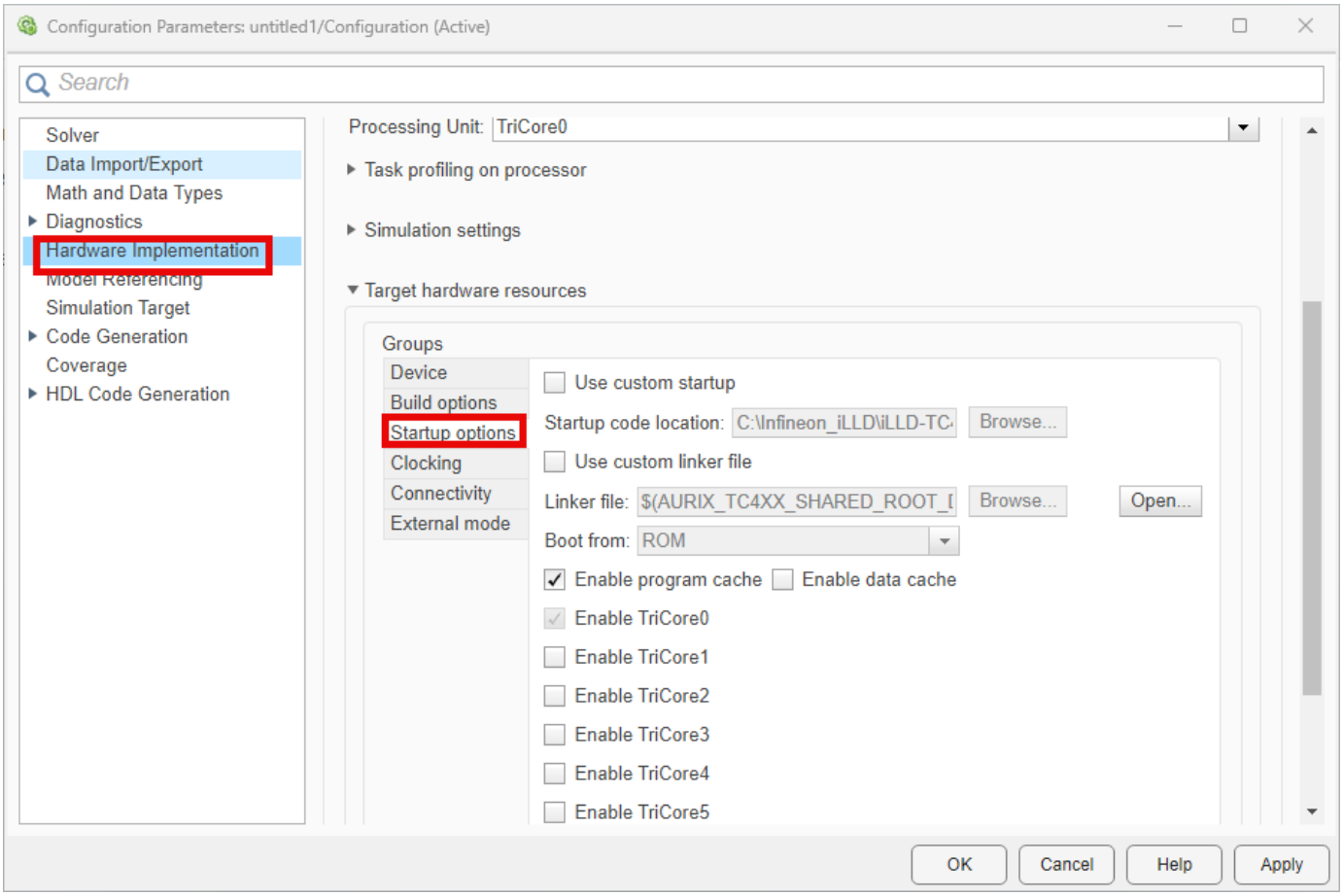
Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now supports custom start-up options, which you can configure in the Configuration Parameters dialog box by navigating to **Hardware Implementation** > **Target hardware resources** > **Startup options**.

- **Use custom startup** — Select this option if you have your own custom startup code, which you can specify in the **Startup code location** parameter.
- **Startup code location** — For each family of Infineon AURIX processors selected under **Target hardware resources**, a startup code file is selected automatically. You can provide your own custom startup code by clicking the **Browse** button.

In addition, several existing parameters have been moved to the **Startup options** tab:

- **Use custom linker file**, **Linker file**, and **Boot From** parameters were previously in the **Build options** tab.
- **Enable program cache**, **Enable data cache**, **Enable TriCore#**, and **Enable PPU** parameters were previously in the **Device** tab.

For more information on these parameters, see “Model Configuration Parameters”.



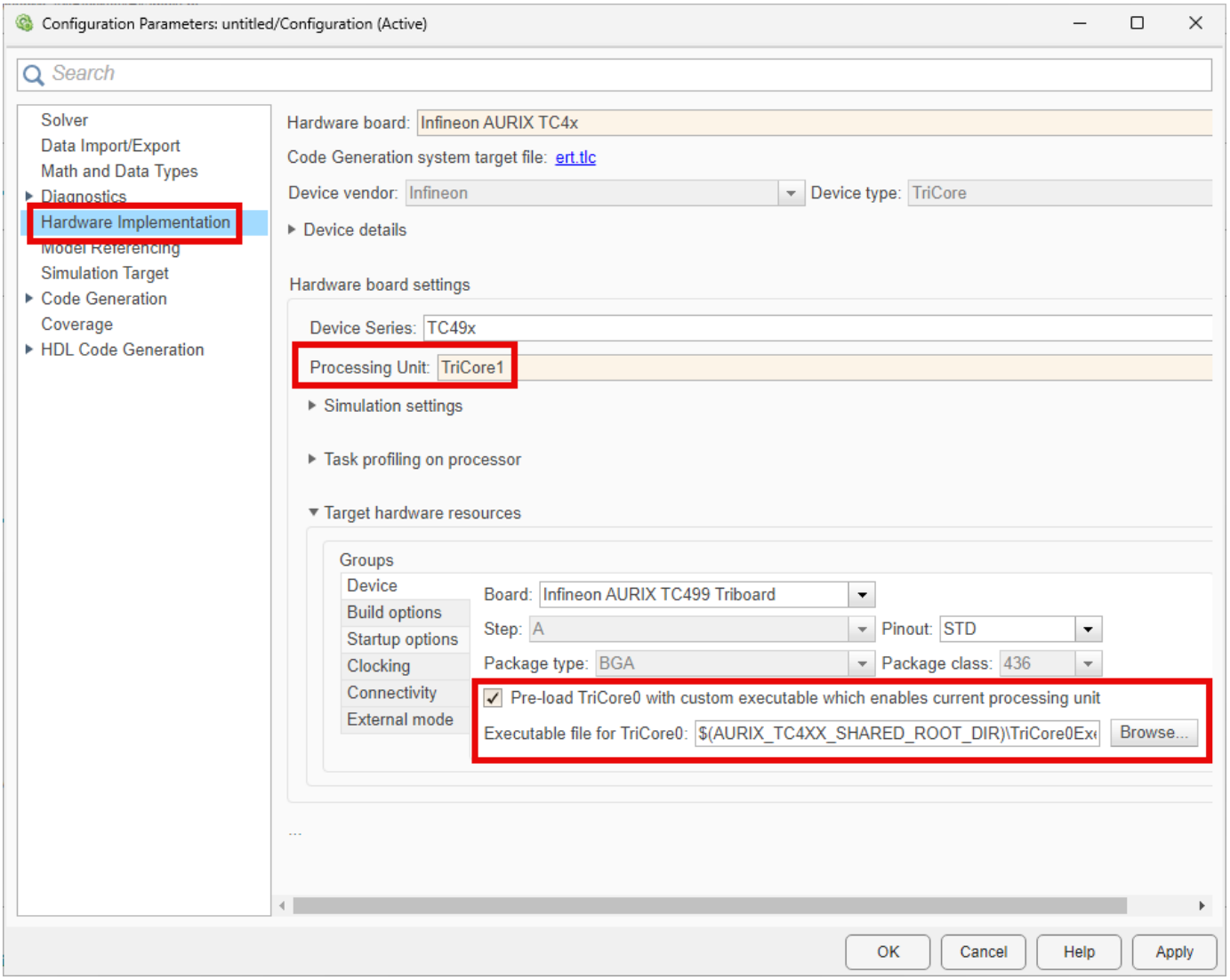
Infineon AURIX TC4x Microcontrollers: Support for custom executable file options for TriCore 0

Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now supports custom executable file options for the TriCore 0 processing unit, which you can configure in the Configuration Parameters dialog box by navigating to **Hardware Implementation > Target hardware resources > Device**.

- **Pre-load TriCore0 with custom executable which enables current processing unit** — Select this option to pre-load the TriCore 0 processing unit with a custom executable file to enable the participating processing unit.

  TriCore 0 is the principal processing unit that handles system initialization, boot processes, and critical control tasks in the AURIX microcontrollers. If you design application model to run on the processing units other than the TriCore 0 processing unit, then you must initialize these processing units by deploying an executable code of the TriCore 0 processing unit.
- **Executable file for TriCore0** — Depending on the choice of **Processing Unit** parameter, a default executable code file is selected automatically. You can provide your own custom executable file by clicking the **Browse** button.

This figure shows custom executable option enabled for the TriCore 1 processing unit.



Infineon AURIX TC4x Microcontrollers: Support for Infineon AURIX TC49xN, TC48x, and TC46x hardware boards

Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now supports TC49xN, TC48x, and TC46x **Device Series** of Infineon AURIX hardware boards.

You can now:

- Use Digital Port Read, Digital Port Write, PWM, TMADC, QSPI, Encoder, Hardware Interrupt, CDSP, DSADC, and FCC blocks to design single-core and multicore application models by using the TriCore 0, TriCore 1, TriCore 2, TriCore 3, TriCore 4, and parallel processing unit (PPU) of the TC49xN series hardware boards (Infineon AURIX TC4X9 Triboard and Custom board).

  You can use the `Basic data accumulation TM (FC7)`, `Basic average TM (FC8)`, and `Basic median TM (FC9)` “DSP Filter chain” options with the TC49xN series hardware boards.
- Use Digital Port Read, Digital Port Write, PWM, TMADC, QSPI, Encoder, Hardware Interrupt, DSADC, CDSP, and Resolver blocks to design single-core and multicore application models by

using the TriCore 0, TriCore 1, TriCore 2, and TriCore 3 processing units of TC48x series hardware boards (Infineon AURIX TC4X9 Triboard, Infineon AURIX TC4X7 Triboard, and Custom board).

- Use Digital Port Read, Digital Port Write, PWM, TMADC, QSPI, Encoder, Hardware Interrupt, FCC, and Resolver blocks to design single-core and multicore application models by using the TriCore 0, TriCore 1, TriCore 2, TriCore 3, and parallel processing unit (PPU) of TC46x series hardware boards (Infineon AURIX TC4X9 Triboard, Infineon AURIX TC4X7 Triboard, Infineon AURIX TC4X6 Triboard, and Custom board).

  For more information on these modeling workflows, see “SoC-Based Multicore Modeling Workflow for Infineon AURIX Microcontrollers” and “Monolithic Modeling for Infineon AURIX Microcontrollers”.
- Run processor-in-the-loop (PIL) simulation in the serial communication mode.
- Monitor signals and tune parameters in the external mode simulation.

Infineon AURIX TC4x Microcontrollers: New and updated examples that demonstrate the capabilities of TC4x Microcontrollers

Use the following examples to explore capabilities of Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

- “Code Verification Using PIL Simulation on Infineon AURIX TC4x Microcontrollers for ISO 26262 Certification” — This example shows how to configure the IEC Certification Kit example model for Infineon AURIX TC4x hardware board and use Simulink Test Manager for source code verification using PIL simulations.
- “Neural Network Based Position Estimation for Field-Oriented Control of PMSM on Infineon AURIX TC4x Microcontrollers” — This example shows how to implement field-oriented control (FOC) of a permanent magnet synchronous motor (PMSM) using a rotor position estimated by an autoregressive neural network (ARNN) trained with Deep Learning Toolbox™.
- “Getting Started with PPU Accelerator for Infineon AURIX TC4x Microcontrollers” — This example now shows how to configure the custom storage class and memory sections to control the placement of code and data in the memory of PPU.

ARM Cortex-A Processors: CMSIS CRL support for FFT and IFFT System object and block

Starting in R2026a, you can use the Embedded Coder Support Package for ARM Cortex-A Processors to generate code using the ARM Cortex-A CMSIS code replacement library (CRL).

- `dsp.FFT` System object™
- `dsp.IFFT` System object
- FFT block
- IFFT block

For more information, see “Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex-A Processors” and “Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex-A Processors”.

ARM Cortex-A Processors: CMSIS CRL support for FIR interpolator System object and block

Starting in R2026a, you can use the Embedded Coder Support Package for ARM Cortex-A Processors to generate code for `dsp.FIRInterpolator` System object and FIR Interpolator block using the ARM Cortex-A CMSIS CRL.

For more information, see “Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex-A Processors” and “Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex-A Processors”.

ARM Cortex-A Processors: CMSIS CRL support for matrix multiplication and matrix transpose

Starting in R2026a, you can use the Embedded Coder Support Package for ARM Cortex-A Processors to generate code for matrix multiplication and matrix transpose using the ARM Cortex-A CMSIS CRL.

For more information, see “Supported MATLAB Functions with CMSIS Library for ARM Cortex-A Processors” and “Supported Simulink Blocks with CMSIS Library for ARM Cortex-A Processors”.

ARM Cortex-M Processors: CRL support for Convolution 2D Layer and Fully Connected Layer blocks

Starting in R2026a, you can use the Embedded Coder Support Package for ARM Cortex-M Processors to generate optimized code for Convolution 2D Layer and Fully Connected Layer blocks added in the `ARM Cortex-M` code replacement library (CRL) on ARM Cortex-M processors. For more information, see “Supported Deep Learning Toolbox Blocks with CMSIS Library for ARM Cortex-M Processors”.

ARM Cortex-M Processors: Helium support for digital signal operations and math operations using ARM Cortex-M CRL

Starting in R2026a, you can use the Embedded Coder Support Package for ARM Cortex-M Processors to generate code for digital signal operations and math operations using the Helium instruction set and ARM Cortex-M CRL. For more information, see “Generate SIMD Code using Helium Instruction Set for ARM Cortex-M Processors”.

ARM Cortex-M Processors: Quantize and deploy speech command recognition for STM32 boards example

The “Quantize and Deploy Speech Command Recognition for STM32 Boards” example shows how to quantize and deploy a convolutional neural network (CNN)-based speech command recognition model to an STM32F769I-Discovery board using STM32CubeMX and the ARM Cortex-M code replacement library (CRL).

Qualcomm Hexagon Processors: Support for code-interface packaging and static code metrics

The Embedded Coder Support Package for Qualcomm® Hexagon® Processors now packages the generated C or C++ interface code as a reusable function. The code generator in the support

package also performs static analysis of the generated code from Simulink and includes these metrics in the HTML Code Generation Report.

Qualcomm Hexagon Processors: Use ARM Cortex-A CMSIS CRL to generate optimized code for Qualcomm Android Board and Qualcomm Linux Board

You can now use the ARM Cortex-A CMSIS code replacement library (CRL) to generate calls to the CMSIS-DSP library optimized for these CPU-based target boards: Qualcomm Android Board and Qualcomm Linux Board.

Qualcomm Hexagon Processors: Generate SIMD Code for 16-bit and 32-bit integer operations using Hexagon and HVX instruction set

You can now generate SIMD code for 16-bit and 32-bit integer operations when you use the Hexagon and HVX Instruction Set Extension (ISE) support for code optimization. For more information, see “Generate SIMD Code using Hexagon and HVX Instruction Set for Qualcomm Hexagon Processors”.

Qualcomm Hexagon Processors: Support for GPU and DSP QNN backends

You can now design embedded applications based on the Qualcomm AI Engine Direct SDK and perform inference using your deep learning networks on the GPU and DSP Qualcomm Neural Network (QNN) backends.

You can use the new blocks and System objects that correspond to the GPU and DSP backends:

- QNN GPU Predict block or `qnn.GPU` System object
- QNN DSP Predict block or `qnn.DSP` System object

Qualcomm Hexagon Processors: Specify backend configuration file QNN HTP and QNN LPAI System objects and blocks

You can now specify the backend configuration file when using these blocks and System objects:

- QNN HTP Predict block and `qnn.HTP` System object
- QNN LPAI Predict block and `qnn.LPAI` System object

You can use the backend configuration file to define backend-specific settings for model execution.

Qualcomm Hexagon Processors: Support for IPCV code replacement using HVX

Use Embedded Coder Support Package for Qualcomm Hexagon Processors to simulate and generate optimized code for image processing and computer vision (IPCV) functions. You can use these new MATLAB® functions and Simulink blocks to simulate the behavior of the Hexagon Vector eXtension (HVX) IPCV functions.

- `QHVXIPCV.conv2d`
- `QHVXIPCV.erode3x3`
- `QHVXIPCV.erode5x5`
- `QHVXIPCV.erode7x7`
- `QHVXIPCV.dilate3x3`
- `QHVXIPCV.dilate5x5`
- `QHVXIPCV.dilate7x7`
- HVX Conv2D block
- HVX Erode block
- HVX Dilate block

Qualcomm Hexagon Processors: Fixed-point input support for abs QHL code replacement

Use Embedded Coder Support Package for Qualcomm Hexagon Processors to generate optimized code for `abs` function with fixed-point Q15 inputs by using the Qualcomm Hexagon library (QHL) for scalar processors.

For more information, see “Conditions for Code Replacement of Math Functions with QHL”.

Qualcomm Hexagon Processors: Variable fractional length input support for QHL code replacements

Use Embedded Coder Support Package for Qualcomm Hexagon Processors to generate optimized code for math operators with variable fractional length input for fixed-point Q15 inputs by using the Qualcomm Hexagon library (QHL) for scalar processors.

You can generate optimized code for these operations with fixed-point Q15 inputs with variable fractional lengths:

- Addition
- Subtraction
- Multiplication

For more information, see “Conditions for Code Replacement of Math Operators with QHL”.

Qualcomm Hexagon Processors: Support for new QHL and HVX Optimized FFT implementations

You can now use the Embedded Coder Support Package for Qualcomm Hexagon Processors to generate optimized code that leverages the latest Qualcomm Hexagon QHL and Hexagon HVX library implementations for FFT .

For more information, see

- “Conditions for Code Replacement of DSP Blocks with QHL”
- “Conditions for Code Replacement of DSP System Objects with QHL”

- “Conditions for Code Replacement of DSP Blocks with HVX”
- “Conditions for Code Replacement of DSP System Objects with HVX”

Qualcomm Hexagon Processors: Deploy YOLOX object detection and tracking system on Qualcomm Hexagon NPU example

The “Deploy YOLOX Object Detection and Tracking System on Qualcomm Hexagon NPU” example shows how to deploy a YOLOX-based real-time object detection and tracking system on Qualcomm Hexagon NPU using the Qualcomm AI Engine Direct (QNN) SDK.

Qualcomm Hexagon Processors: Deploy speech enhancer model on Qualcomm Hexagon DSP example

The “Deploy Speech Enhancer Model on Qualcomm Hexagon DSP” example shows how to deploy a Simulink model designed to enhance a noisy speech audio on Qualcomm Hexagon DSP.

Check bug reports for issues and fixes

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

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

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

R2025b

Version: 25.2

New Features

Bug Fixes

Quality and stability improvements

R2025b delivers quality and stability improvements, building on the new features introduced in R2025a.

Hardware Support

Qualcomm Hexagon Processors: Support for Hexagon SDK 6.2.0.1

The Embedded Coder Support Package for Qualcomm Hexagon Processors now supports integration with Hexagon SDK 6.2.0.1. The Hardware Setup window for the support package provides the link to download this SDK version and validate the installation.

Qualcomm Hexagon Processors: Deploy code to Qualcomm Android CPU, Qualcomm Linux CPU, and Hexagon Linux, along with PIL support

The Simulink configuration parameter Hardware Board supports three new boards:

- Qualcomm Android® board – Deploy code to Qualcomm Android CPU.
- Qualcomm Linux board – Deploy code to Qualcomm Linux CPU.
- Qualcomm Hexagon Linux board – Deploy code to Qualcomm Hexagon DSP for Linux based boards.

These boards also support Processor-in-the-Loop (PIL) simulation to verify the code generation results.

Qualcomm Hexagon Processors: Accelerate AI network simulation by using hardware

You can now improve simulation speed for the eNPU module of a Hexagon processor V73. To do so, enable the Simulink configuration parameter **Use hardware for AI network simulation** (available under **Hardware Implementation > Target Hardware Resources > Device**). This option speeds up simulation of the eNPU Predict block by using the target eNPU.

Qualcomm Hexagon Processors: Automatic allocation of quantization value range for eNPU block and System object

If you use the latest supported version for LPAI Addon SDK (2.5.0.0), the software now automatically allocates the quantization ranges in eNPU Predict block and `hexagon.ENPU` System object. You need to specify the quantization range only if the installed LPAI SDK version is not the latest.

Qualcomm Hexagon Processors: Support for Qualcomm AI Engine Direct SDK and QNN backends

You can now design embedded applications based on the Qualcomm AI Engine Direct SDK and perform inference using your deep learning networks on the corresponding Qualcomm Neural Network (QNN) backends. The Hardware Setup window for the support package provides the link to download the latest Qualcomm AI Engine Direct SDK version and validate the installation. You can use these new blocks and System object that correspond to each of the QNN backends (HTP, CPU, or LPAI):

- QNN HTP Predict block or `qnn.HTP` System object

- QNN CPU Predict block or `qnn.CPU` System object
- QNN LPAI Predict block or `qnn.LPAI` System object

These blocks and System objects allow you to specify QNN models or context binary files specific to the host and target (QNN backend) separately. You can also specify whether dequantization is needed. You can then predict responses based on input data that matches the input-layer properties of the deep learning network.

Check bug reports for issues and fixes

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

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

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

R2025a

Code Generation from MATLAB Code

Automate profiling of generated MATLAB function code from command line

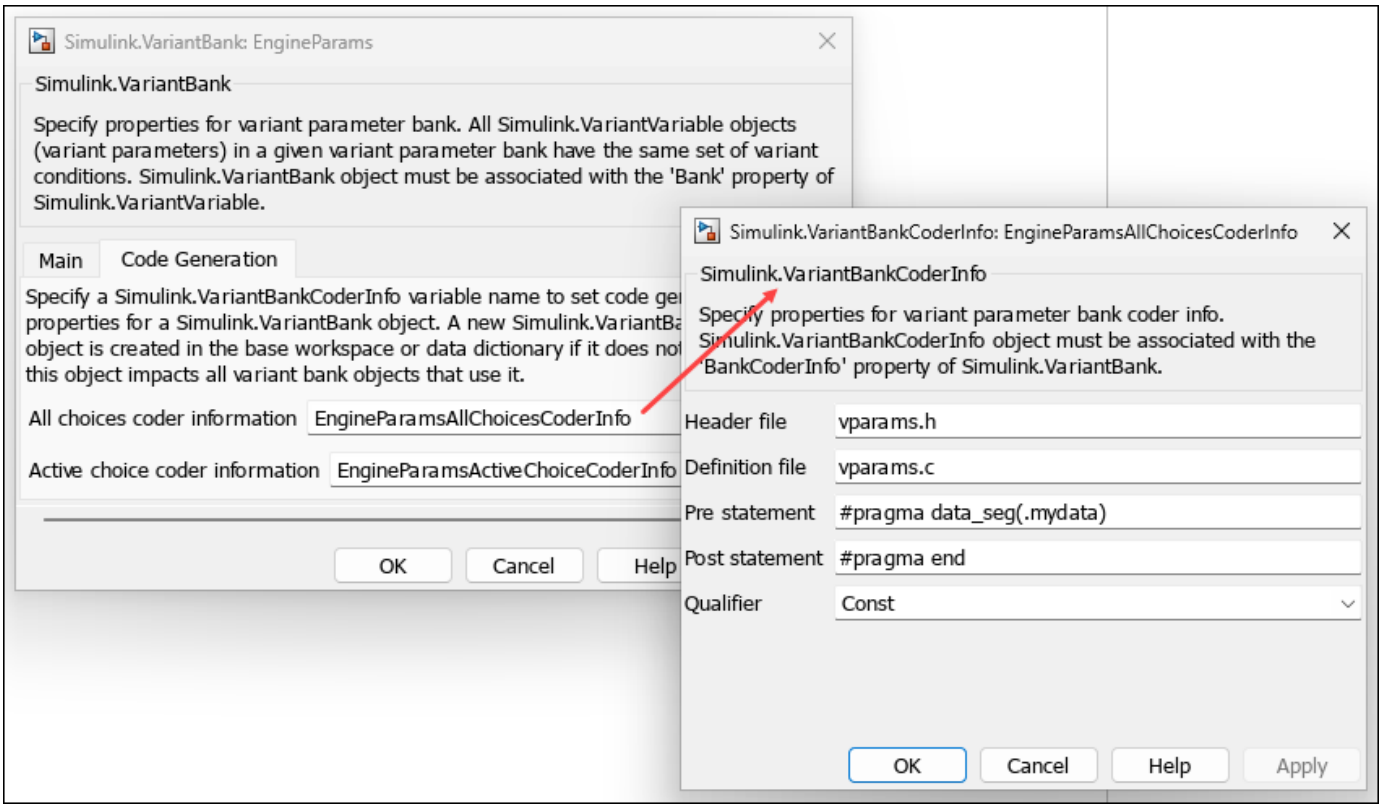
Use the `coder.profile.test.runTests` function to perform execution-time and stack usage profiling of code generated from MATLAB entry-point functions. From the command line, use the function to run multiple software-in-the-loop (SIL) or processor-in-the-loop (PIL) tests with execution-time or stack usage profiling enabled. View and analyze results by using the Code Profile Analyzer app. For more information, see Automate Execution-Time and Stack Usage Profiling of MATLAB Function.

Model Architecture and Design

Specify separate memory sections, header, and definition files for the parameter structure array and pointer used by a variant parameter bank

Before R2025a, when you grouped variant parameters in a model using variant parameter banks (`Simulink.VariantBank` objects), you could specify only a single memory section to place the structure array that groups the parameter values and the pointer that selects the active set of values from the array. The code generator placed the declarations and definitions of the array and the pointer in the same header and definition files that you specified.

Starting in R2025a, you can specify separate memory sections, header, and definition files to place the structure array and pointer in the generated code. You can also specify a type qualifier that is applied to the declaration and definition of the array and pointer variables in the code.



To support these enhancements, the `Simulink.VariantBank` class has two new properties, `AllChoicesCoderInfo` and `ActiveChoiceCoderInfo`. The `Simulink.VariantBankCoderInfo` class has a new property named `Qualifier`.

For an example on variant parameter banks, see Group Variant Parameter Values and Conditionally Switch Active Value Sets in Generated Code.

⚠Compatibility Considerations

The `BankCoderInfo` property of the `Simulink.VariantBank` class will be removed. Use the `AllChoicesCoderInfo` and `ActiveChoiceCoderInfo` properties instead. If your existing code sets the `BankCoderInfo` property, Simulink uses this value to automatically set the value for the `AllChoicesCoderInfo` and `ActiveChoiceCoderInfo` properties.

Generate code for variant parameter banks in model reference hierarchy

You can simulate and generate code for a model reference hierarchy that uses a variant parameter bank (`Simulink.VariantBank`) to group variant parameters (`Simulink.VariantVariable` objects) used by models across the hierarchy. Previously, when you used Embedded Coder to generate code that switches parameter banks using a pointer variable, model reference hierarchies were not supported. Starting in R2025a, the code generator identifies variant parameters in a variant parameter bank within the base workspace or data dictionaries visible to the model hierarchy and includes them in the parameter bank definition. The code generator places the header file that contains the type definition of the variant parameter bank in a shared utilities folder `slprj/target/_sharedutils`, where `target` is the name of the system target file for code generation. This mechanism enables sharing the variant parameter bank across multiple models in the hierarchy.

For an example, see Generate Code for Variant Parameter Banks in Model Reference Hierarchy.

Generate code for Simulink.VariantControl and Simulink.VariantVariable objects that contain expression values

You can set the value of a `Simulink.VariantControl` object and the values of a `Simulink.VariantVariable` object to mathematical expressions and then use Embedded Coder to generate code that preserves these expressions.

1. Create a `Simulink.Parameter` object or an object of a class that inherits from `Simulink.Parameter`.
2. Set the `Value` property of the object to the expression by using the `slexpr` function.
3. Use the object to set the value of the variant control or the values of the choices of a variant parameter.

For applicable code generation techniques and considerations, see Code Generation of Parameter Objects With Expression Values. For an example, see Use Mathematical Expressions as Values of `Simulink.VariantControl` Objects and `Simulink.VariantVariable` Choices.

Remove redundant preprocessor conditions to prevent unnecessary variant condition evaluations

When an overarching preprocessor condition already implies a block preprocessor condition, the code generator removes the redundant condition from the generated code. However, if the overarching condition does not imply a block condition, the code generator guards the code generated for the block with its own preprocessor condition to maintain correct execution. Previously, the code generator included block conditions without verifying their coverage by overarching conditions, resulting in redundant condition checks.

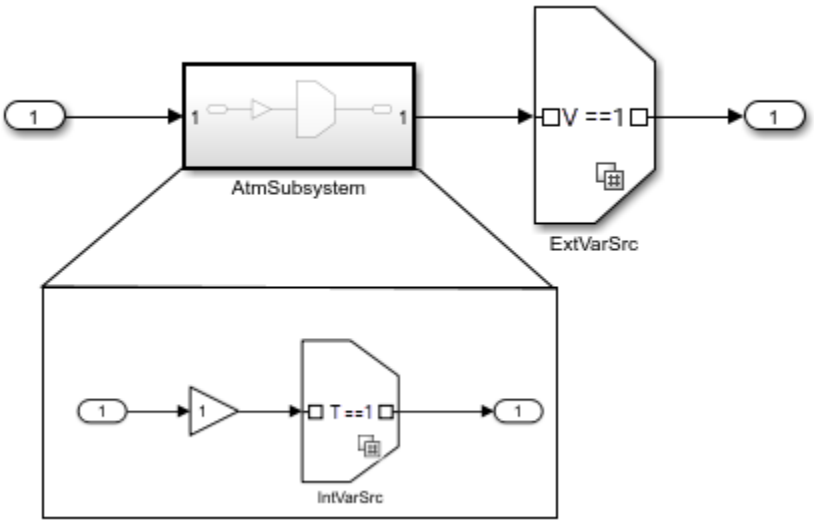
This redundancy often occurred in models when:

- The model contained an atomic subsystem.
- The **Function packaging** parameter of the atomic subsystem was set to `Auto`, `Reusable function`, or `Nonreusable function`, which creates separate functions for atomic subsystems in the generated code.
- Variant blocks had **Variant activation time** set to `code compile`.

The code generator does not remove redundant checks from the code it generates for exported functions and Stateflow® charts.

Consider the atomic subsystem `AtmSubsystem` in the model below. The **Function packaging** parameter is set to `Reusable` and the **Variant activation time** parameters of the underlying Variant Source block `IntVarSrc` and the connected Variant Source block `ExtVarSrc` are set to `code compile`. The `AtmSubsystem` block has a variant control expression `T == 1 && V == 1`. The variant control expression `T == 1` propagates from the `IntVarSrc` block, and `V == 1` propagates from the `ExtVarSrc` block.

When the condition `T == 1 && V == 1` evaluates to `true`, it inherently verifies that `T == 1` and `V == 1` are `true`. Therefore, having a separate check for `V == 1` within this block is redundant.



This table compares the code generated in R2024b and R2025a for the model. In R2025a, the code generator removes redundant condition checks to reduce unnecessary evaluations.

R2024b Generated Code	R2025a Generated Code

/* Output and update for atomic system: '<Root>/AtmSubsystem' */
#if T == 1 && V == 1
 	
static real_T AtmSubsystem(real_T rtu_In1)
{
  real_T rty_Out1_0;
 	
  /* Gain: '<S1>/Gain' */
#if T == 1
 	
  rty_Out1_0 = rtu_In1;
 	
#endif
 	
  /* End of Gain: '<S1>/Gain' */
  return rty_Out1_0;
}
 	
#endif
	/* Output and update for atomic system: '<Root>/AtmSubsystem' */
	#if T == 1 && V == 1
	
	static real_T AtmSubsystem(real_T rtu_In1)
	{
	  /* Gain: '<S1>/Gain' */
	  return rtu_In1;
	}
	
	#endif

Code Interface Configuration and Integration

★Configure subcomponent-level code interface packaging and root-level I/O

Starting in R2025a, you can configure code interface packaging for subcomponents (model references) using the **Code interface packaging (subcomponent)** model configuration parameter. New models use a reusable subcomponent code interface by default. Models from previous versions use a setting that is consistent with the setting for **Total number of instances allowed per top model**.

If you use an ERT or AUTOSAR Classic system target file, configuring the subcomponent code interface as nonreusable enables the **Implement root-level I/O as** model configuration parameter. Use this parameter to control how root-level I/O data are passed to subcomponent entry-point functions. For example, passing root-level I/O data as global variables allows you to eliminate function arguments and reduce stack memory usage.

For more information, see Configure Code Interface Packaging and Root-Level I/O Data for Model References.

★Persistency service interface support for target platform nonvolatile memory

Starting in R2025a, you can specify a measurement service interface for persistent data. The service interface for persistent data supports storing state and data store data in nonvolatile memory that is managed externally by a nonvolatile memory management service in the target environment.

To specify persistent data for state and data store elements, use the Code Mappings editor to map the service interface `PermanentRAM` to the variables. `PermanentRAM` is a newly introduced service interface for persistent data, defined in the Measurement Service Interfaces section of coder dictionaries created after R2024b.

To configure a new measurement service interface for persistent data, create a service interface in the Measurement Service Interfaces section of the Embedded Coder Dictionary and select **Persistent Data**. The storage class that you select for the service interface must have **Data Initialization** set to `None`. Alternatively, use the new storage class `PersistentMemory`, which is provided in the Storage Class section of the coder dictionary.

To obtain the properties of persistent data in measurement service interfaces, such as the Simulink identifier and initial conditions, use the `getPersistentDataInterfaces` code descriptor method for the `coder.descriptor.MeasurementServiceInterface` class.

For more information, see Generate C Interface Code for Measurement Service Using Persistent Data for Component Deployment.

Limitations removed for service code interface configurations

Starting in R2025a, for component models that are configured to use a service code interface, you can select the Subsystem block parameter **Function with separate data**, which instructs the code generator to place internal data for the subsystem in a data structure that is separate from the parent

model data structures. Previously, you had to clear this parameter because service code interface configurations did not support placement of subsystem internal data in a separate data structure.

You can also use these model configuration parameter settings. Previously, service code interface configurations did not support alternative settings for these parameters.

- Set **Default parameter behavior** to `Tunable`. With this setting, the code generator represents numeric block parameters and variables that use storage class `Auto` as tunable fields of a global parameters data structure. Previously, you had to set this parameter to `Inline`.
- Select **Support non-inlined S-functions**. With this setting, the code generator produces code for noninlined S-functions. Previously, you had to clear this parameter.
- Clear **Single output/update function**. With this setting, the code generator produces separate entry-point functions (*`model`*`_output` and *`model`*`_update`) for output and update functions. Previously, you had to select this parameter.
- Clear **Combine signal/state structures**. With this setting, the code generator produces separate data structures for block global signals and global states. Previously, you had to select this parameter.
- Clear **Remove error status field in real-time model data structure**. With this setting, the code generator includes an error status field in the generated real-time model data structure `rtModel`. Use macros to monitor the field for errors or to set the field with error status data. Previously, you had to select this parameter.
- Select **Classic call interface**, which enables you to generate calls to model functions that are compatible with the `main` program module (`grt_main.c` or `grt_main.cpp`) produced for models created before R2012a and configured with a GRT system target file. Previously, you have to clear this parameter.
- Set **File packaging format** to `Compact` or `Compact (with separate data file)`. With these parameter settings, the code generator packages component code in fewer files. Previously, you had to set this parameter to `Modular`.
- Set **Invalid input data access in chart initialization** to `none` or `warning`. Previously, you had to set this parameter to `error`.

For lists of remaining constraints and limitations, see Service Interface Constraints and Limitations.

Generate service code interface report for model reference

Starting in R2025a, when you generate a code generation report for model reference, it includes a code interface report. The report documents the generated code interface, including model entry-point functions and auxiliary services. The code interface information helps you to review the generated code interfaces and integrate them with other code.

For more information, see Analyze Generated Service Code Interface Report.

⚠Functionality being removed or changed

Configuring C/C++ function prototypes for subsystems no longer supported

Errors

The capability to configure C/C++ function prototypes for subsystems is no longer supported. The `RTW.configSubsystemBuild` function, previously used for this purpose, and its associated user interface (UI) have been removed.

If you need to configure a customized function prototype for a subsystem, convert the subsystem to a Model block and configure function prototypes for the converted model by using the code mappings. For models configured for C code generation, select option **Copy code mappings** when converting, which specifies that the conversion copies the existing code mapping configuration from the parent model to the new referenced model. See Copy Code Mappings When Converting Subsystems to Referenced Models. For models configured for C++ code generation, you must manually configure the code mapping settings on the converted model.

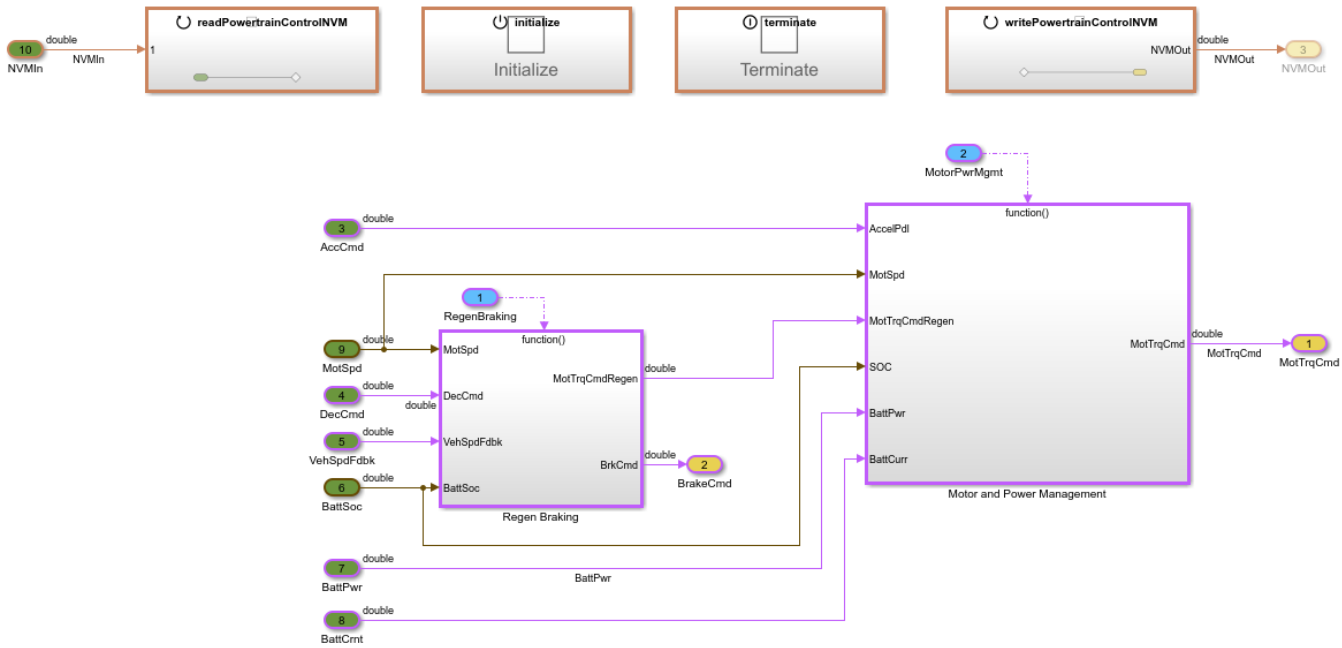
To convert a subsystem to a Model block, use the UI workflow described in Convert Subsystem to Referenced Model and Generate Code or, at the command line, use function Simulink.SubSystem.convertToModelReference.

Code Generation

★New examples of code generation and deployment workflow

A new set of examples shows how to generate, test, optimize, and deploy code for an electric vehicle powertrain controller. Each step of the workflow highlights tools and practices that you can use to scale your software development workflow to meet your needs. The workflow shows you how to:

- Follow model-based design practices to support your code generation requirements.
- Configure models to generate code according to your deployment and optimization goals.
- Incorporate existing code in the model and generated code.
- Test the generated code for numerical behavior and performance.
- Build and deploy an executable by integrating the generated code with middleware.



For more information, see Develop and Deploy Embedded Software Components to Platform Services by Using Embedded Coder.

★Generate code for models that use enumerations in MATLAB namespaces

Starting in R2025a, you can simulate a Simulink model and generate code for a model that uses an enumerated type that is derived from `Simulink.IntEnumType` and is defined inside a MATLAB namespace. By default, the code generator generates these enumerations in namespaces for C++ code generation. For C code generation, the code generator prefixes the enumerations with the MATLAB namespace. To disable the generation of C++ namespaces and C prefixes in the generated code, clear the new model configuration parameter **Preserve MATLAB namespaces in generated code**.

This table displays the generated code for enumerated type `Colors` in the MATLAB namespace `MyColors` with **Preserve MATLAB namespace in generated code** selected and cleared.

Language	Selected	Cleared

C	`typedef enum {`
	`  black,`
	`  white,`
	`} MyColors_Colors;`
		`typedef enum {`
		`  black,`
		`  white`
		`}  Colors;`

C++	`namespace MyColors {`
	`  enum class Colors : int32_t {`
	`  black,`
	`  white`
	`  };`
	`}`
		`enum class Colors : int32_t {`
		`  black,`
		`  white`
		`  };`

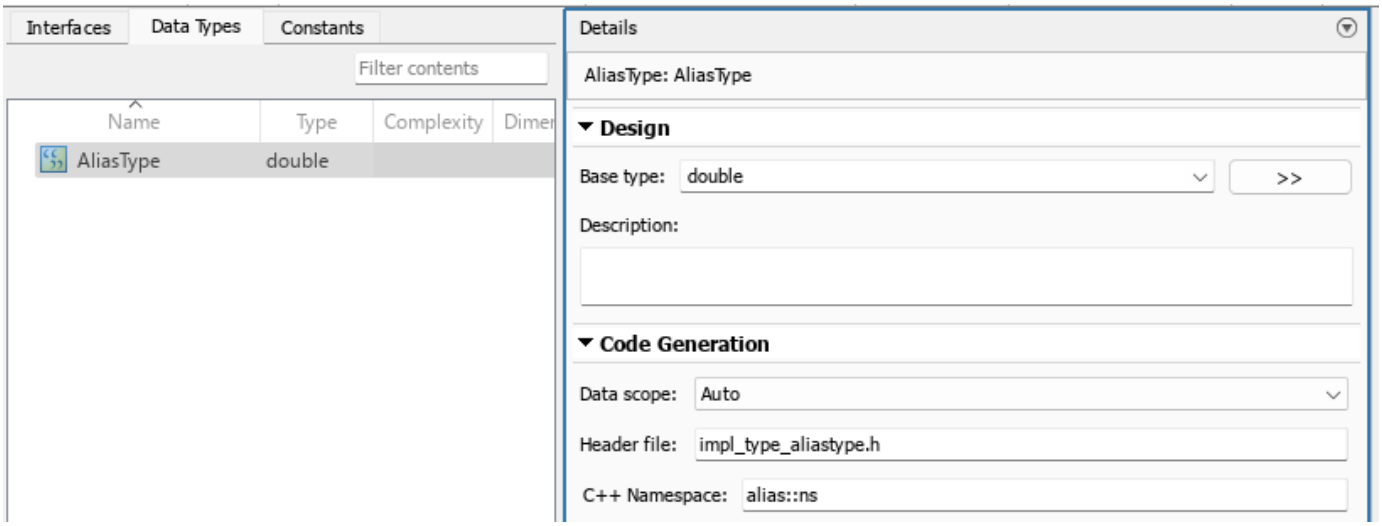
For more information, see Use Enumerated Data in Generated Code.

Manage data type replacement in workflows that use support packages

Starting in R2025a, when the configuration parameter **Data type replacement** is set to `Use C data types with fixed-width integers`, you can generate code for third-party hardware workflows that use Embedded Coder support packages. The code generator replaces Simulink data types with C99 data types. For more information, see Manage Replacement of Simulink Data Types in Generated Code.

Specify individual namespaces of some architectural data in generated C++ code

In R2025a, specify C++ namespaces in generated C++ code for data types consumed from the Architectural Data section of a data dictionary. Set the C++ namespace for a data type in the Architectural Data Editor by selecting the architectural data object and expanding the **Code Generation** section in the Details pane.



When generating C++ code for a model that consumes a data type owned by a data dictionary, the generated C++ code contains namespaces as defined in the data dictionary.

```
#ifndef ALIAS_NS_IMPL_TYPE_ALIASTYPE_H_
#define ALIAS_NS_IMPL_TYPE_ALIASTYPE_H_
#include <cstdint>

namespace alias
{
  namespace ns
  {
    using AliasType = double;
  }                                     /* namespace ns */
}                                       /* namespace alias */

#endif                                  //ALIAS_NS_IMPL_TYPE_ALIASTYPE_H_
```

The `CppNamespace` property is configurable only on these architectural data objects.

- `Simulink.dictionary.archdata.DataInterface`
- `Simulink.dictionary.archdata.ServiceInterface`
- `Simulink.dictionary.archdata.StructType`
- `Simulink.dictionary.archdata.AliasType`
- `Simulink.dictionary.archdata.EnumType`
- `Simulink.dictionary.archdata.NumericType`

For `Simulink.dictionary.archdata.NumericType` objects, the C++ namespace appears in the generated code only when the `Simulink.dictionary.archdata.NumericType` object is configured as an alias.

For more information, see Generate Individual C++ Namespaces for Architectural Data Types.

Generate code for Create Diagonal Matrix, Discrete FIR Filter, and Second-Order Section Filter blocks with symbolic dimension inputs

Starting in R2025a, you can generate code for the Create Diagonal Matrix, Discrete FIR Filter, and Second-Order Section Filter (DSP System Toolbox) blocks that have symbolic dimensions as inputs.

For more information, see https://www.mathworks.com/help/releases/R2025a/ecoder/ug/implement-dimension-variants-for-array-sizes-in-generated-code.html.

Functionality being removed or changed

Code generator no longer uses specified or derived range information for reusable subsystems or reusable library subsystems

Behavior change

Starting in R2025a, regardless of whether you select the model configuration parameter **Optimize using the specified minimum and maximum values**, the code generator does not use specified or derived range information inside a reusable library subsystem or inside a reusable subsystem with more than one instance in the model.

Deployment

Customize and export variant parameters to ASAP2 file

Starting in R2025a, Embedded Coder adds support to include the variant coding section and export variant characteristics in an ASAP2 file. You can:

- Generate the variant coding section in an ASAP2 file.
- Add custom data to the variant coding section.
- Add or delete a variant characteristic or variant criterion record in an ASAP2 file.
- Find existing variant characteristics or variant criterion records.
- Modify or find the existing records inside the variant coding section in an ASAP2 file.

For more information about generating variant coding section from a Simulink model, see Generate Variant Coding Section in ASAP2 File.

For more information about customization, see Create and Add Custom Variant Coding.

Deploy DDS Blockset applications on ARM 64 targets

Starting in R2025a, you can deploy DDS Blockset applications on ARM 64 target machines by using the Embedded Coder Support Package for Linux Applications.

For more information, see Build Simulink Model and Deploy Application.

⚠Functionality being removed or changed

GenerateASAP2 configuration parameter cannot be set to 'on'.

Starting in R2025a, you cannot set the `GenerateASAP2` configuration parameter to `'on'`. Instead, use Generate Calibration Files tool or `coder.asap2.export` function to generate an ASAP2 file after building the model.

For more information about generating an ASAP2 file, see Generate ASAP2 and CDF Calibration Files.

Performance

Generate SIMD code for Logical Operator AND/OR blocks

In R2025a, you can generate SIMD code for Intel® platforms from Logical Operator blocks for `AND` and `OR` operators. For more information, see Generate SIMD Code from Simulink Blocks for Intel Platforms.

Generate SIMD code for Data Type Conversion blocks and cast functions

In R2025a, you can generate SIMD code for Intel and ARM platforms Data Type Conversion blocks. When generating code from MATLAB code, you can generate SIMD code for calls to the `cast` function. For more information, see Generate SIMD Code from Simulink Blocks for Intel Platforms.

Unified analysis for efficient buffer reuse

Starting in R2025a, you can generate efficient code by enabling unified analysis using the new model configuration parameter **Unify buffer reuse candidates**. When you enable the parameter, the code generator performs a unified analysis, which first analyzes all potential buffer reuse candidates and then identifies and implements the efficient reuses in the generated code. Analyzing the potential buffer reuse candidates and identifying the efficient reuses upfront reduces the possibility of missing reuse opportunities. This can significantly reduce RAM consumption and data copies. Here, buffer reuse candidates refer to the buffers that can potentially be reused. For more information, see Generate Efficient Code Using Unified Analysis.

Display task allocation across CPUs for XCP external mode simulations

If you run XCP-based external mode simulations with execution-time profiling enabled, you can use the Code Profile Analyzer app to display the allocation of tasks across CPUs. On the **CPU Usage** tab, analyze how tasks are scheduled. You can:

- View and assess the workload of each CPU.
- Investigate how the scheduler performs load balancing.

For more information, see Validate Task-to-CPU Mapping.

Enhanced automated code execution-time and stack usage profiling

When you use the `coder.profile.test.runTests` function to perform execution-time and stack usage profiling, you can use:

- Key-value mapping to loop over pairs of configuration parameter options
- Hooks to customize automated code profiling

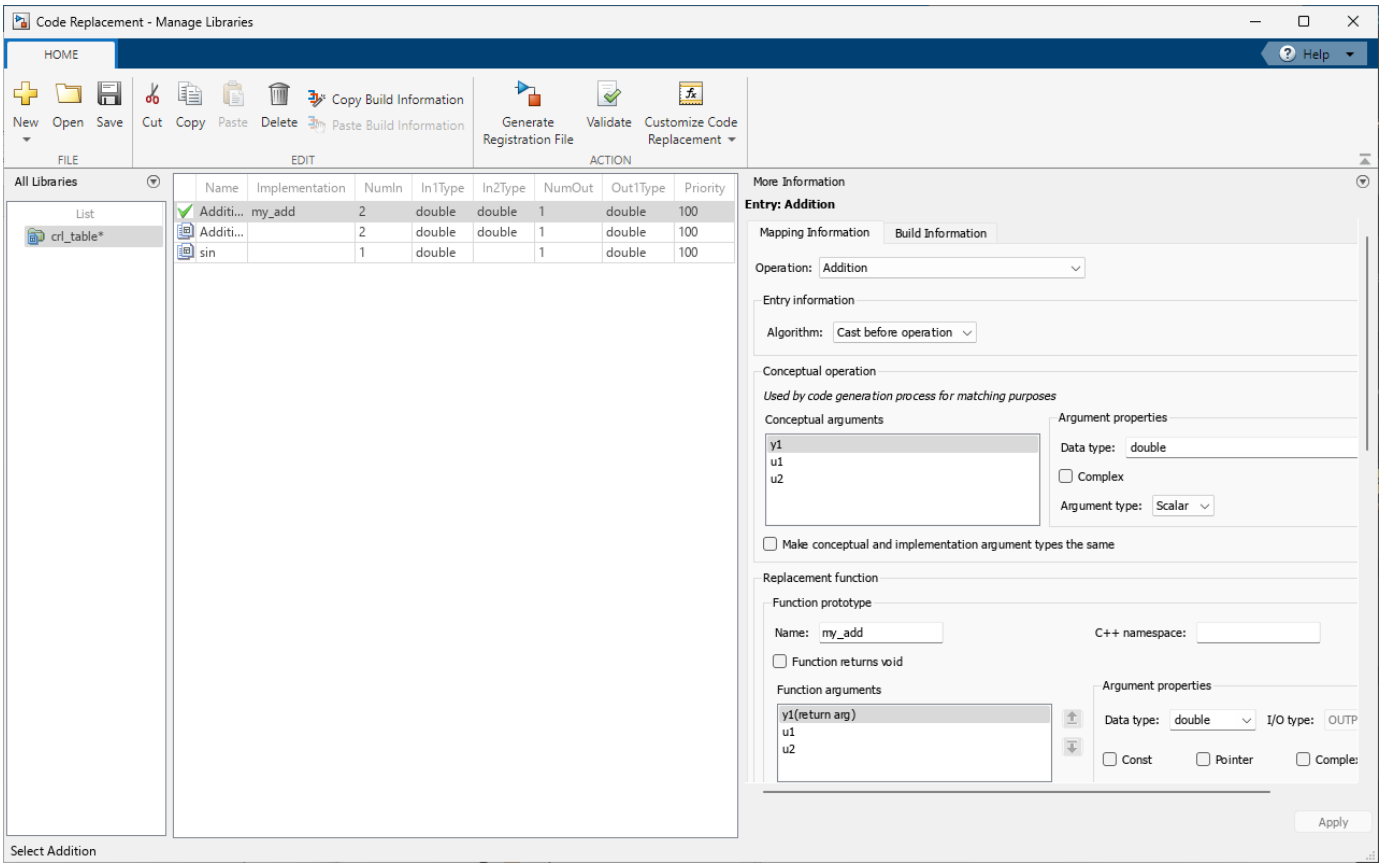
For more information, see Loop Over Configuration Options During Code Profiling and Customize Automated Code Profiling Using Hooks.

Define code replacement libraries in new Code Replacement Tool

In R2025a, you can create and edit code replacement libraries by using the new Code Replacement Tool. Use the options in the toolstrip to:

- Add code replacement tables and entries.
- Define custom code replacement entry classes.
- Validate and register your code replacement library.

For more information, see Code Replacement Tool.



Improved code efficiency by preserving parameters and eliminating redundant checks

In R2025a, Embedded Coder improves the handling of Simulink.Parameter objects during code generation by leveraging range analysis (Fixed-Point Designer) with specified minimum and maximum values. When you select the parameter **Optimize using the specified minimum and maximum values**, Embedded Coder preserves tunable parameters as variables, retaining them for flexibility after code generation rather than converting them to constants in the generated code. Embedded Coder also eliminates redundant checks for parameter conditions, enhancing the efficiency of the generated code.

While this optimization has existed in earlier releases, Embedded Coder extends its capability to preserve parameters with identical minimum and maximum values under these conditions:

- Parameters that use a non-`Auto` storage class, such as `Model Default`, `ExportedGlobal`, `ImportedExtern, ImportedExternPointer`, or `Custom`.
- Parameters set to use the `Auto` storage class, with **Default parameter behavior** set to `Tunable`.
- Block parameters that have identical minimum and maximum values, and with **Default parameter behavior** set to `Tunable`.

The generated code effectively preserves such parameters under these conditions, providing flexibility for post-code-generation modifications, and improving code readability and simplifying debugging.

By specifying minimum and maximum values for parameters, Embedded Coder further optimizes the generated code by evaluating parameter constraints and eliminating redundant conditional statements. For instance, if a parameter named `threshold` is constrained to values between `1` and `3`, conditions that are always true or false based on these range values are optimized out. Consider these `if`-statements:

```
void foo(int threshold) {
    // Range of threshold: 1 to 3

    // Original Code
    if (threshold < 1) {
        // Code that will never execute
        int result = threshold * 2; // Example computation
    }
    if (threshold > 0) {
        // Code that will always execute
        int result = threshold + 10; // Example computation
    }
}
```

The first `if` statement in the code is unreachable, so Embedded Coder deletes the `if` statement and its associated code. The second `if` statement always executes, so Embedded Coder moves the code outside the `if` statement and deletes the `if` statement. The code then runs without an unnecessary conditional check.

```
void foo(int threshold) {
    // Range of threshold: 1 to 3

    // Embedded Coder removes the first if-block entirely and
    // hoists the second block's code out as the condition is always true.

    // Code that will always execute:
    int result = threshold + 10; // Example computation
}
```

This optimization is especially pertinent to:

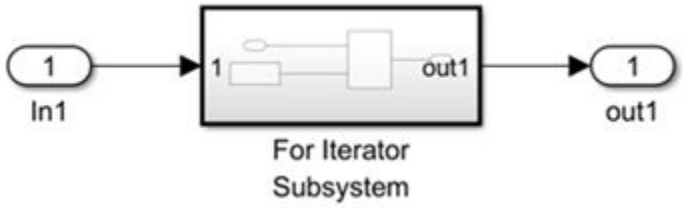
- Parameters with non-`Auto` storage classes that have defined range constraints.
- Parameters that use the `Auto` storage class when **Default parameter behavior** is set to `Tunable`.

To manage data representation and optimize code generation using storage classes, see Choose Storage Class for Controlling Data Representation in Generated Code. To understand the constraints of tunable parameters in Simulink, see Limitations for Block Parameter Tunability in Generated Code.

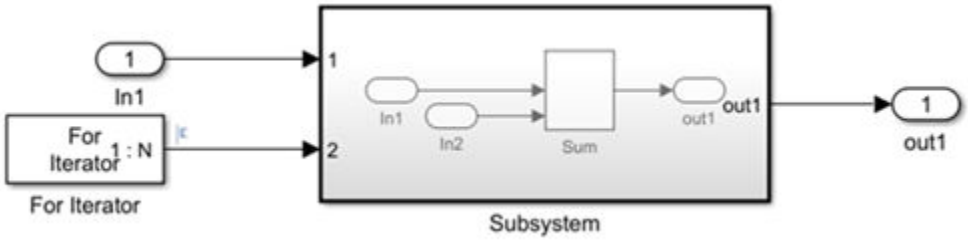
Enhanced subsystem functions return in generated code

Staring in R2025a, for modeling patterns containing subsystems, the code generator generates subsystem functions that return scalar output values whenever possible. Returning scalar output values can reduce stack size, ROM consumption, and data copies, as well as global RAM usage. Additionally, it enables compiler optimizations by eliminating pointer-type function arguments and supports compliance with MISRA C:2012 Rule 17.8 (Polyspace Bug Finder).

Consider the model `CombineInputs`, which contains a For Iterator subsystem.



The For Iterator subsystem contains a reusable subsystem that combines the input signals.



In R2024b, the code generator generated this code.

```
void CombineInputs_Subsystem(real_T rtu_In1, real_T rtu_In2, real_T *rty_out1)
{
  *rty_out1 = rtu_In1 + rtu_In2;
}

void CombineInp_ForIteratorSubsystem(real_T rtu_In1, real_T *rty_out1)
{
  int32_T s1_iter;
  for (s1_iter = 0; s1_iter < 10; s1_iter++) {
    CombineInputs_Subsystem(rtu_In1, (real_T)s1_iter + 1.0, rty_out1);
  }
}
```

The subsystem function `CombineInputs_Subsystem` was generated for the reusable subsystem, which passed the scalar output value through the pointer `*rty_out1` to the function `CombineInp_ForIteratorSubsystem`.

In R2025a, the code generator generates this code.

```
real_T CombineInputs_Subsystem(real_T rtu_In1, real_T rtu_In2)
{
  return rtu_In1 + rtu_In2;
}

void CombineInp_ForIteratorSubsystem(real_T rtu_In1, real_T *rty_out1)
```

```
{
  int32_T s1_iter;
  for (s1_iter = 0; s1_iter < 10; s1_iter++) {
    *rty_out1 = CombineInputs_Subsystem(rtu_In1, (real_T)s1_iter + 1.0);
  }
}
```

The subsystem function `CombineInputs_Subsystem` returns a scalar output value, which eliminates the pointer dereferencing overhead and consequently reduces RAM consumptions and data copies.

Replacement of relay operation

In R2025a, the C/C++ code generator supports code replacement of the `relay` function for scalar, vector, and matrix input and for output arguments for code generated from a Simulink model. For more information, see Math Functions – Simulink Support.

Verification

★Tune model workspace parameters during atomic subsystem SIL/PIL simulations

Starting in R2025a, you can tune model workspace parameters during atomic subsystem software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations. Tune the parameters by modifying their values in the test harness workspace. You can also tune model workspace parameters that use AUTOSAR storage classes.

For information about running an atomic subsystem SIL simulation, see Unit Test Subsystem Code with SIL/PIL Manager.

Hardware Support

★Qualcomm Hexagon Processors: Predict response of eAI network and deploy to eNPU

Starting in R2025a, Embedded Coder Support Package for Qualcomm Hexagon Processors provides a new Simulink block, eNPU Predict, and a new System object, `hexagon.ENPU`, to predict response of a pre-compiled eAI model (eAI network) that is designed using the Qualcomm LPAI SDK. The block and System object complements the EAI Runtime framework of LPAI SDK and allows you to verify the performance and accuracy by selecting either `Qualcomm Hexagon Android Board` or `Qualcomm Hexagon Simulator` as the target hardware for deployment.

The pre-compiled eAI model can be based on multiple-input multiple-output tensor layout. For floating-point input, the eNPU Predict block and the `hexagon.ENPU` System object also allow you to specify quantization parameters to convert the input to fixed-point before obtaining the output.

★ARM Cortex-M Processors: Support for SIMD code generation with Helium on ARM Cortex-M55 board

Starting in R2025a, you can generate optimized single instruction multiple data (SIMD) code using the Helium instruction set extension for Simulink and MATLAB on ARM Cortex-M55 board. For more information, see Generate SIMD Code using Helium Instruction Set for ARM Cortex -M Processors.

★STMicroelectronics STM32 Processors: Support for boards based on STM32F1xx and STM32G0xx processors

Embedded Coder Support Package for STMicroelectronics® STM32 Processors now supports the new boards based on STM32F1xx and STM32G0xx processors. Starting in R2025a, you can:

- Generate and build code using an STM32CubeMX project file.
- Use the blocks listed in this table to implement Model-Based Design workflows for boards based on STM32F1xx processors.

Analog to Digital Converter	CAN Read	CAN Write	Digital to Analog Converter
Encoder	I2C Controller Read	I2C Controller Write	PWM Output
Digital Port Read	Digital Port Write	Hardware Interrupt	Timer
Timer Capture	SPI Receive	SPI Transmit	SPI Controller Transfer
UART/USART Write	UART/USART Read		

- Use the blocks listed in this table to implement Model-Based Design workflows for boards based on STM32G0xx processors.

Analog to Digital Converter	FDCAN Read	FDCAN Write	Digital to Analog Converter
Encoder	I2C Controller Read	I2C Controller Write	PWM Output

Digital Port Read	Digital Port Write	Hardware Interrupt	Timer
Timer Capture	SPI Receive	SPI Transmit	SPI Controller Transfer
UART/USART Write	UART/USART Read	Comparator	

- Monitor signals and tune parameters in external mode. For more information, see Monitoring and Tuning Using STMicroelectronics STM32 Processor Based Boards.
- Run processor-in-the-loop (PIL) simulation in the serial communication mode.

STMicroelectronics STM32 Processors: Support for Modbus, Execution Profiler, and MQTT blocks

Starting in R2025a, you can use these blocks with STM32 processor based boards:

- Use client- and server-based Modbus Read and Modbus Write blocks to access server device registers to read and write data using boards based on STM32F7xx processors.
- Use the Execution Profiler block to profile the execution time of functions using the Data Watchpoint Trace (DWT) Timer, Digital Port Write (GPIO), or Timer measurement modes.
- Use the updated MQTT Publish and MQTT Subscribe blocks to exchange messages with boards based on STM32F4xx processors using the MQTT protocol, which is ideal for power- and bandwidth-constrained devices on wireless networks.

STMicroelectronics STM32 Processors: Support for I2S Mic In and I2S Audio Out base rate trigger for scheduler interrupt

You can now schedule Simulink models containing I2S Mic In and I2S Audio Out blocks to execute based on direct memory access (DMA) interrupt rates. This enhancement allows models to align execution times with audio data rates and improves the real-time performance of audio applications.

STMicroelectronics STM32 Processors: FreeRTOS support for Ethernet-based blocks

Starting R2025a, the Embedded Coder Support Package for STMicroelectronics STM32 Processors introduces FreeRTOS support for Ethernet-based blocks, including the TCP/IP functionality. With this update, you can utilize the multitasking and scheduling capabilities of FreeRTOS in networked applications.

STMicroelectronics STM32 Processors: Enhancements to SPI Communication Protocol

The SPI Receive, SPI Transmit, and SPI Controller Transfer blocks of Embedded Coder Support Package for STMicroelectronics STM32 Processors now support multibyte data transfers over the SPI protocol using buffered interrupts and direct memory access (DMA) mode. This enhancement enables efficient handling of larger data packets, making it easier to manage high-volume data transfers over the SPI protocol.

STMicroelectronics STM32 Processors: Support for STM32CubeMX 6.12.0

You can now use Embedded Coder Support Package for STMicroelectronics STM32 Processors to configure STM32CubeMX 6.12.0 to generate and deploy code to boards based on STM32 processors.

STMicroelectronics STM32 Processors: Support for ARM Compiler Toolchain 6.14

Starting R2025a, Embedded Coder Support Package for STMicroelectronics STM32 Processors adds support for the ARM Compiler Toolchain 6.14.

STMicroelectronics STM32 Processors: Higher data acquisition rates using connected IO

Starting in R2025a, Embedded Coder Support Package for STMicroelectronics STM32 Processors supports higher data acquisition rates for sensor/peripheral blocks with STM32 processor-based boards using connected IO. For a higher data acquisition rate, in the connected IO mode, use the sensor/peripheral blocks in the model. Enable **Simulation Pacing**, and then set the **Simulation time per wall clock second** parameter to 1. For more information, see Communicate with Hardware Using Connected IO.

STMicroelectronics STM32 Processors: Support for environmental and IMU sensors

Starting this release, Embedded Coder Support Package for STMicroelectronics STM32 Processors supports using sensor blocks listed in this table with the STM32 processor-based boards.

Environmental Sensors

Usage	Sensor Block
Measure equivalent carbon dioxide concentration (eCO2) and equivalent total volatile organic compound concentration (eTVOC).	CCS811 Air Quality Sensor
Measure relative humidity and ambient temperature.	HTS221 Humidity Sensor
Measure the analog and digital values of linear acceleration, voltages at ADC inputs, and temperature.	LPS22HB Pressure Sensor

IMU Sensors

Usage	Sensor Block
Measure linear acceleration along the *X*, *Y* and *Z*-axes. The block also provides the option to enable the data ready interrupt.	ADXL34x Accelerometer
Measure acceleration, angular rate, and magnetic field. This block also supports advanced application features with parameters such as **Single tap**, **Double tap**, **High g detection**, **Any motion**, **Slow motion**, **Flat detection**, and **Data ready** interrupts.	BMI160 IMU Sensor
Measures linear acceleration, angular velocity, and magnetic field strength along the *X*, *Y* and *Z*-axes. The block also reads temperature.	ICM20948 IMU Sensor
Measure the analog and digital values of linear acceleration, voltages at ADC inputs, and temperature. This block also supports advanced application features such as **Click**, **Free-fall**, **Inertial wake-up**, **6D/4D position**, and **6D/4D movement** detections.	LIS3DH Accelerometer Sensor
Measure magnetic field and temperature.	LIS3MDL Magnetometer Sensor
Measure linear acceleration, magnetic field, and temperature.	LSM303C IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DS3 IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DS3H IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DSL IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DSM IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DSO IMU Sensor
Measure linear acceleration, angular velocity, and temperature.	LSM6DSR IMU Sensor
Measure the distance to a target object for a complete field of view. This block also provides the option to select one of four ranging modes.	VL53L0X Time of Flight Sensor

STMicroelectronics STM32 Processors: Connected IO support for I2C blocks

Embedded Coder Support Package for STMicroelectronics STM32 Processors now supports running a Simulink model containing I2C Controller Read and I2C Controller Write blocks in the connected IO mode on STM32 processor-based boards.

AMD Zynq Hardware: Support package external mode simulation with TCP/IP protocol is being removed

Support for the TCP/IP protocol will be removed for External Mode Simulation with TCP/IP in a future release. For the Embedded Coder Support Package for AMD SoC Devices, use the XCP protocol for simulation as described in External Mode Simulation by Using XCP Communication.

Infineon AURIX TC3x Microcontrollers: Support for Infineon low-level driver (iLLD) 1.0.1.18.0

Starting in R2025a, you can use the Infineon low-level driver (iLLD) 1.0.1.18.0 with Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Infineon AURIX TC3x Microcontrollers: Enhancements to EVADC block

The updated EVADC block now supports three different conversion modes. On the **G #** tab of global parameters in the EVADC Peripheral Configuration window of the Hardware Mapping tool, set the **Conversion type** parameter to one of these values:

- `One-shot` — Select this option for a one-time analog-to-digital conversion after a software or hardware trigger event. In case of a hardware trigger event, you must add another EVADC block with the block parameter **Mode** set to `ADC Trigger` to fill the queue input register. You must map these EVADC trigger and read blocks using the **Select read block** parameter in the **Group select** tab in the Hardware Mapping tool.
- `Refill with trigger` — Select this option to wait for a software or hardware trigger for analog-to-digital conversion. This mode automatically refills the queue input register and a valid queue entry waits for a trigger event to initiate conversion.
- `Refill with one time trigger` — Select this option for perpetual analog-to-digital conversion of valid queue entries after a software trigger. This mode automatically refills the queue input register.

Infineon AURIX TC3x Microcontrollers: New MCAN blocks for CAN communication

Use the new MCAN Transmit and MCAN Receive blocks from Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers to write data to and read data from a CAN bus, respectively. After adding these blocks to the model, you can map the block parameters to the hardware using the MCAN Transmit Peripheral Configuration and MCAN Receive Peripheral Configuration tools.

If you configure the MCAN Transmit block to write data in the CAN message format, then depending on the frame format, you must use the CAN Pack or the CAN FD Pack block with the MCAN Transmit block.

If you configure the MCAN Receive block to receive packed CAN data, then depending on the frame format, you must use the CAN Unpack or CAN FD Unpack block with the MCAN Receive block.

Infineon AURIX TC3x Microcontrollers: Support for multicore external mode simulation

Starting in R2025a, you can use Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers to simulate a multicore model containing the homogenous TriCores of Infineon AURIX TC3x microcontrollers in external mode. Previously, only TriCore 0 was supported for external mode simulation.

Infineon AURIX TC3x Microcontrollers: Enhancements to interprocessor communication (IPC) blocks

Starting in R2025a, the enhanced IPC blocks support enumerated data types for communication between Simulink and Infineon AURIX TC3x microcontrollers, allowing transmission of state values using variable names.

- You can use enumerated types created in Simulink using the `Simulink.defineIntEnumType` function or by subclassing the `Simulink.IntEnumType` class with the Interprocess Data Read and Interprocess Data Write blocks.
- In the external mode simulations, Simulink displays enumerated types as their corresponding enumerated variable type names, enhancing clarity and consistency.

Infineon AURIX TC3x Microcontrollers: Create Simulink block for custom or third-party C/C++ files

Starting in R2025a, you can use the IO Device Builder app in Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers to create a System object for custom or third-party source files (C/C++). You can create a Simulink block from the generated System object by using the MATLAB System block, which can then be added to your Simulink model.

To use the IO Device Builder app, select `Infineon AURIX TC3x` as the hardware board. Then, on the **Hardware** tab of the Simulink toolstrip, navigate to the **Prepare** section, and under **Design**, select `IO Device Builder`. A series of screens then leads you through the process of creating a System object and a Simulink block. For more information, see Get Started with IO Device Builder.

Infineon AURIX TC3x Microcontrollers: New examples that demonstrate the capabilities of TC3x Microcontrollers

Use the following new examples to explore capabilities of Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

- Analyze Sensorless Observers for Field-Oriented Control Using Multiple Cores of Infineon AURIX TC3x — This example shows how to use the TriCore processing units to implement sensorless field-oriented control algorithms.
- Integrate Code Generated for Infineon TC3x with ADS Workflow — This example shows how to export code generated from Simulink and deploy it to hardware using AURIX Development Studio (ADS).
- PIL Simulation for AUTOSAR Software Component with Infineon AURIX TC3x Microcontroller — This example shows how to verify and validate the generated code of the AUTOSAR component models using PIL simulations.

- Custom Storage Class for Infineon AURIX TC3x Microcontrollers — This example shows how to define a custom storage class, which enables applications to control the placement of data and code elements in any TriCore core memory of the device.

Infineon AURIX TC4x Microcontrollers: Support for Infineon low-level driver (iLLD) 2.0.1.3.13

Starting in R2025a, you can use the Infineon low-level driver (iLLD) 2.0.1.3.13 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Update to TASKING Smartcode v10.2r1

Starting in R2025a, you can download and install the patch version TASKING Smartcode v10.2r1p1 to compile homogenous TriCores of Infineon AURIX microcontrollers. This patch replaces a few installation files of the previously installed TASKING Smartcode v10.2r1.

Infineon AURIX TC4x Microcontrollers: Enhancements to CDSP block

Starting this release, Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers includes these enhancements to the CDSP block:

- The updated block accepts input from the TMADC block and General purpose (GP) registers. For GP registers, the block accepts data from Simulink source blocks such as the Repeating Sequence Stair block.

  If you set the block parameter **DSP Filter chain** to `Demo FIR (FC0)`, `Advanced average 1 (FC1)`, `Advanced median (FC3)`, or `Advanced average 2 (FC4)`, the CDSP block expects a 16-bit primary input from the GP registers.

  If you set the block parameter **DSP Filter chain** to `Basic data accumulation (FC2)` or `Basic AURIX (FCM)`, the CDSP block expects a 32-bit primary input from the GP registers, where the upper 16 bits represent sideband signals of the DA filter block. These sideband signals control the integration operation in DA filter block, with the value 0x6000 starting and resetting the integrator and 0x4000 starting the integrator.
- You can specify the source of primary input and configure the corresponding parameters in the **Configuration** tab of the CDSP Peripheral Configuration window in the Hardware Mapping tool.

  This source must match the block or GP input registers connected to the **Primary input** port of the CDSP block in the Simulink model.
- You can specify the source of the secondary input and configure the corresponding parameters in the **Configuration** tab of the CDSP Peripheral Configuration window in the Hardware Mapping tool.

  This source must match the block or GP input registers connected to the **Secondary input** port of the CDSP block in the Simulink model.

Infineon AURIX TC4x Microcontrollers: Peripheral block support for TC4Dx Hardware boards

Starting in R2025a, you can use these blocks to implement Model-Based Design workflows for Infineon AURIX TC4Dx hardware boards:

- Digital Port Read
- Digital Port Write
- MCAN Transmit
- MCAN Receive
- QSPI
- Hardware Interrupt

Infineon AURIX TC4x Microcontrollers: Enhancements to deep learning code replacement library

Starting in R2025a, you can use Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to generate optimized code for these MATLAB functions by using the enhanced **MetaWare TC4x PPU for Deep Learning** code replacement library (CRL).

- Hyperbolic tangent (tanh) and sigmoid activation functions used within recurrent neural network layers such as long short-term memory (LSTM), bidirectional LSTM (BiLSTM), and gated recurrent unit (GRU) layers of deep learning networks.
- Tanh and sigmoid functions operating on dlarray (Deep Learning Toolbox) objects, which you can use for custom training loops in deep learning networks.
- Tanh function operating on numeric array objects, which you can use for custom layers in deep learning networks.

Infineon AURIX TC4x Microcontrollers: Enhancements to SIMD code replacement libraries

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers includes these enhancements to the MetaWare TC4x PPU SIMD 256-bit and MetaWare TC4x PPU SIMD code replacement libraries (CRLs).

- The arithmetic left-shift and right-shift operation entries have been renamed.

Operator Name	Supporting Data Type	Previous Implementation Function Name	New Implementation Function Name
vshiftleft	w (32-bit integer)	mw_vvsl_w	mw_vvasrm_w
	h (16-bit integer)	mw_vvsl_h	mw_vvasrm_h
	b (8-bit integer)	mw_vvsl_b	mw_vvasrm_b
vshiftright	w (32-bit integer)	vvcslm	mw_vvslm_w
	h (16-bit integer)	vvcslm	mw_vvslm_h
	b (8-bit integer)	vvcslm	mw_vvslm_b

- These CRL table entries have been added for saturating addition and subtraction operations:

Operator Name	Supporting Data Types	Implementation Function Name
vsatadd	32-bit signed integer, 16-bit signed integer, and 8-bit signed integer	vvadd_sat
	32-bit unsigned integer, 16-bit unsigned integer, and 8-bit unsigned integer	vvadd_satu
vsatsub	32-bit signed integer, 16-bit signed integer, and 8-bit signed integer	vvsub_sat
	32-bit unsigned integer, 16-bit unsigned integer, and 8-bit unsigned integer	vvsub_satu

Infineon AURIX TC4x Microcontrollers: Support for multicore external mode simulation

Starting in R2025a, you can use Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to simulate a multicore model containing the homogenous TriCores (TriCore 0 to TriCore 5) of Infineon AURIX TC4x microcontrollers in external mode. Previously, only TriCore 0 was supported for external mode simulation.

Infineon AURIX TC4x Microcontrollers: Enhancements to interprocessor communication (IPC) blocks

Starting in R2025a, the enhanced IPC blocks support enumerated data types for communication between Simulink and Infineon AURIX TC4x microcontrollers, allowing transmission of state values using variable names.

- You can use enumerated types created in Simulink using the `Simulink.defineIntEnumType` function or by subclassing the `Simulink.IntEnumType` class with the Interprocess Data Read and Interprocess Data Write blocks.
- In the external mode simulations, Simulink displays enumerated types as their corresponding enumerated variable type names, enhancing clarity and consistency.

Infineon AURIX TC4x Microcontrollers: Support for nSIM simulator

Starting in 2025a, you can use the nSIM instruction-set simulator in Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to run PIL simulations and test PPU-based application models without connecting to the hardware board. For more information, see PIL Simulation Using nSIM Simulator and Generate and Deploy Optimized Code for Digit Classification Deep Learning Network on Infineon PPU Target.

Infineon AURIX TC4x Microcontrollers: New examples that demonstrate the capabilities of TC4x Microcontrollers

Use the following new examples to explore capabilities of Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

- Custom Storage Class for Infineon AURIX TC4x Microcontrollers — This example shows how to define a custom storage class, which enables applications to control the placement of data and code elements in any TriCore core memory of the device.
- PIL Simulation for AUTOSAR Software Component with Infineon AURIX TC4x Microcontrollers — This example shows how to verify and validate the generated code of the AUTOSAR component models using PIL simulations.
- Generate and Deploy Optimized Code for Digit Classification Deep Learning Network on Infineon PPU Target — This example shows how to run PIL simulations and verify the numerical accuracy of the generated code against the simulation output by using nSIM simulator.

Qualcomm Hexagon Processors: Code optimization enhancements for QHL

Starting in R2025a, Embedded Coder Support Package for Qualcomm Hexagon Processors supports code replacements for additional math functions and math operators by using the Qualcomm Hexagon Library (QHL) for scalar processors.

- Math functions for single-precision floating-point inputs: `tan`, `ceil`, `floor`, and `hypot`
- Math operators for Q15 fixed-point inputs: `add`, `sub`, and element-wise multiplication

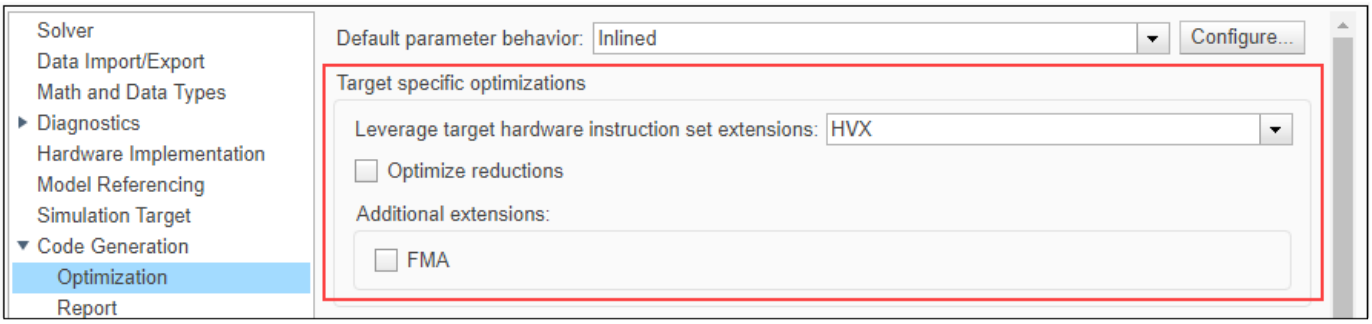
Qualcomm Hexagon Processors: Code optimization enhancements for HVX

Starting in R2025a, Embedded Coder Support Package for Qualcomm Hexagon Processors supports code replacements for math functions and additional math operators by using the Hexagon Vector eXtension (HVX) for vector processors.

- Math functions for single-precision floating-point inputs: `sqrt`, `rsqrt`, `log`, `log2`, `log10`, `pow`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, and `hypot`
- Math operators for Q15 fixed-point inputs: `dot product`

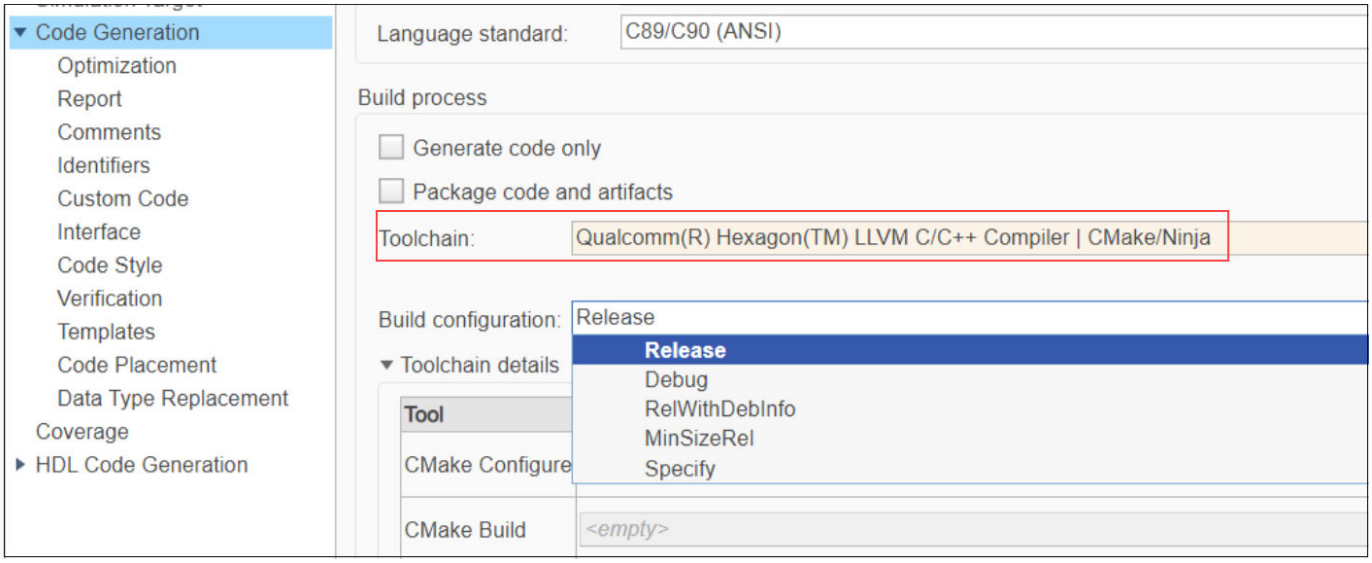
Qualcomm Hexagon Processors: Generate SIMD code, optimize reductions, and enable FMA for HVX

Starting in R2025a, the Hexagon Vector eXtension (HVX) support in Embedded Coder Support Package for Qualcomm Hexagon Processors allows SIMD code generation that helps in improving the overall performance. You can also enable the option to optimize reductions for HVX, along with enabling optional support for the fused multiply-add (FMA) extension.



Qualcomm Hexagon Processors: Support for Hexagon LLVM C/C++ compiler toolchain with CMake

Starting in R2025a, Embedded Coder Support Package for Qualcomm Hexagon Processors supports Hexagon LLVM C/C++ compiler toolchain with CMake. To complete the configuration of CMake-based build generation for Hexagon processor, in the Configuration Parameters dialog box, select `Qualcomm(R) Hexagon(TM) LLVM C/C++ Compiler | CMake/Ninja` in the **Toolchain** parameter and specify the build configuration.



ARM Cortex-A Processors: CMSIS code replacement library support for dsp.SOSFilter System object and Second-Order Section Filter block

Starting in R2025a, you can use Embedded Coder Support Package for ARM Cortex-A Processors to generate optimized code for the `dsp.SOSFilter` System object and the Second-Order Section Filter block using the `ARM Cortex-A CMSIS` code replacement library (CRL) and deploying on ARM Cortex-A target hardware. For more information, see Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex -A Processors and Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex -A Processors.

ARM Cortex-A Processors: CMSIS code replacement library support for dsp.FIRDecimator System object and FIR Decimator block

Starting in R2025a, you can use Embedded Coder Support Package for ARM Cortex-A Processors to generate optimized code for the `dsp.FIRDecimator` System object and the FIR Decimator block using the `ARM Cortex-A CMSIS` code replacement library (CRL) and deploying on ARM Cortex-A target hardware. For more information, see Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex -A Processors and Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex -A Processors.

ARM Cortex-A Processors: Fixed-point and floating-point input support for dsp.FIRFilter System object and Discrete FIR Filter block

Starting in R2025a, you can use Embedded Coder Support Package for ARM Cortex-A Processors to generate optimized code for the `dsp.FIRFilter` System object and the Discrete FIR Filter block with fixed-point and floating-point input using the `ARM Cortex-A CMSIS` code replacement library (CRL). You can deploy the generated optimized code on ARM Cortex-A target hardware. For more information, see Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex -A Processors and Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex -A Processors.

ARM Cortex-A Processors: Deploy parametric audio equalizer on ARM Cortex-A processors

The Deploy Parametric Audio Equalizer on ARM Cortex-A Processors example shows how to deploy a three-band parametric audio equalizer on the ARM Cortex-A Raspberry Pi 3 Model B board running on 32 bit OS. You can deploy this parametric equalizer on any supported ARM Cortex-A boards.

ARM Cortex-M Processors: Code replacement library support for tensor multiplication in deep learning network layers

Starting in R2025a, Embedded Coder Support Package for ARM Cortex-M Processors supports `ARM Cortex-M` CRL to generate optimized code for tensor multiplication in fully connected layer, gated recurrent unit (GRU) layer, long short-term memory (LSTM) layer, and bidirectional long short-term memory (BiLSTM) layer of deep learning networks on ARM Cortex-M target hardware in MATLAB and Simulink. For more information, see Code Generation for Matrix Multiplication in Deep Learning Layers with CMSIS Library for ARM Cortex -M Processors

ARM Cortex-M Processors: Code replacement library support for complex matrix multiplication

Starting in R2025a, you can use Embedded Coder Support Package for ARM Cortex-M Processors to generate optimized code for complex matrix multiplication using the `ARM Cortex-M` code replacement library (CRL) and deploying on ARM Cortex-M target hardware. For more information, see Supported MATLAB Functions with CMSIS Library for ARM Cortex -M Processors and Supported Simulink Blocks with CMSIS Library for ARM Cortex -M Processors.

ARM Cortex-M Processors: Fixed-point input support for dsp.SOSFilter and floating-point input support for Second-Order Section Filter block using ARM Cortex-M code replacement library

Starting in R2025a, you can use Embedded Coder Support Package for ARM Cortex-M Processors to generate optimized code for the `dsp.SOSFilter` with fixed-point input and the Second-Order Section Filter block with floating-point input using the `ARM Cortex-M` code replacement library (CRL). You can deploy the generated optimized code on ARM Cortex-M target hardware. For more information, see Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex -M Processors and Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex -M Processors.

ARM Cortex-M Processors: Generate and deploy optimized code for digit classification deep learning network on ARM Cortex-M target

The Generate and Deploy Optimized Code for Digit Classification Deep Learning Network on ARM Cortex-M Target example shows how to generate optimized code for digit classification deep learning network and deploy the generated code on ARM Cortex-M target.

Check bug reports for issues and fixes

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

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

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

R2024b

Code Generation from MATLAB Code

Static local variable size included in static code metrics report

Starting in R2024b, the static code metrics report includes the total size of static local variables, providing you with a more accurate measurement of memory usage. For more information see Generate Static Code Metrics Report for Simulink Model and Static Code Metrics.

Additional metrics for generated code profiling

When you run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution on Linux-based target hardware, the new MATLAB Coder™ app setting, **SIL/PIL profiling metrics** (`CodeProfilingCodeMetrics`), enables you to profile generated code using advanced metrics from the Performance Monitoring Unit (PMU) of the hardware.

MATLAB Coder App Setting (Configuration Parameter) Name	App Setting (Configuration Parameter) Value	Metric Extracted from PMU
SIL/PIL profiling metrics (`CodeProfilingCodeMetrics`)	`Time ('time')` default	N/A
	`Total Instructions ('totinstr')`	Total instruction count
	`Float Instructions ('floatinsts')`	FPU instruction count
	`Float Operations ('floatopts')`	FPU operation count
	`Integer Instructions ('intinstr')`	ALU instruction count
	`Load Instructions ('loadinstr')`	Load instruction count
	`Store Instructions ('storeinstr')`	Store instruction count
	`Total Cycles ('totcyc')`	Total cycle count
	`L1 Data Cache Misses ('l1dcm')`	L1 data cache misses
	`L1 Instruction Cache Misses ('l1icm')`	L1 instruction cache misses
	`L2 Data Cache Misses ('l2dcm')`	L2 data cache misses
	`L2 Instruction Cache Misses ('l2icm')`	L2 instruction cache misses
	`TLB Data Misses ('tlbdm')`	TLB data misses
	`TLB Instruction Misses ('tlbim')`	TLB instruction misses

MATLAB Coder App Setting (Configuration Parameter) Name	App Setting (Configuration Parameter) Value	Metric Extracted from PMU
	`Memory Stall Cycles ('memstall')`	Memory stall cycles
	`Total Stall Cycles ('totstall')`	Total stall cycles

Using the Code Profile Analyzer, you can view and analyze the profiles created. If your target hardware is not supported natively, you can register custom drivers to extract the required metrics from the PMU. For more information, see Investigate Execution-Time Issues Using PMU Metrics and Register Custom Driver to Extract PMU Metrics.

Code Interface Configuration and Integration

Improved highlighting of data transfer elements in model canvas when configuring service interfaces

When configuring service code interfaces for a model, model element highlighting can help you identify where elements that are listed in the Code Mappings editor appear in the model canvas. When you select a model element in the Code Mappings editor, the Embedded Coder app highlights the corresponding element in the model diagram.

Starting in R2024b, for models configured to use a service code interface, the app provides improved highlighting in the model canvas for data transfer elements. When you select a data transfer element in the **Data Transfer** tab of the Code Mappings editor, highlighting in the model canvas appears as listed in these tables.

Modeling Style	Modeling Pattern	What Is Highlighted
Export-function model	Data transfer signal connects blocks that exchange data and result in callable functions in generated code.	Source block of the data transfer and the data transfer signal
Rate-based model	Data transfer is represented as a Rate Transition block that Simulink inserts implicitly at the output port of a nonvirtual block.	Downstream branches of the signal exiting the output port of the nonvirtual block until the branches reach nonvirtual block input ports
	Data transfer is represented as a Rate Transition block that Simulink inserts implicitly at the input port of a nonvirtual block.	Branches of signals entering the nonvirtual block input port
	Data transfer is represented as a Rate Transition block that you insert.	Rate Transition block

This table lists highlighting improvements in the model canvas for models configured to use an AUTOSAR system target file.

Modeling Style	Modeling Pattern	What Is Highlighted
Export-function model	Data transfer signal connects blocks that exchange data and result in callable functions in generated code.	Source block of the data transfer and the data transfer signal
Rate-based model	Data transfer is represented as a Rate Transition block that you insert.	Rate Transition block

For a data transfer, the Code Mappings editor includes the path to the data transfer signal or Rate Transition block as a hyperlink. For model hierarchies, when you click the link, the input focus changes to the model canvas where the corresponding subsystem level of the model includes that data transfer element.

Highlighting improvements are limited for nonvirtual block input ports that receive virtual bus signals. When a data transfer takes place at a nonvirtual block input port that receives a virtual bus signal, the canvas highlights only the upstream branch back to a nonvirtual block, Bus Creator block, or Bus Selector block output port.

Prior to R2024b:

- For data transfers, the model canvas highlighted the source or destination block that the selected data transfer signal connects to instead of expected model elements listed in the preceding tables.
- For export-function models configured to use an AUTOSAR system target file, the model canvas did not highlight data transfers.

For the list of Simulink blocks that are virtual, see Nonvirtual and Virtual Blocks. For information about configuring data transfer service interfaces, see Configure Data Transfer Service Interfaces for Data Transfer Signals and Generate C Data Transfer Service Interface Code for Component Deployment.

⚠Naming convention changed for access methods in C++ generated code for Bus Element ports

Starting in R2024b, the naming convention for access methods generated for nonvirtual In Bus Element and Out Bus Element blocks uses the block label instead of the block name when generating C++ class code. Names of access methods are now in the form `set`*`PortName_ElementName`* for In Bus Element ports and `get`*`PortName_ElementName`* for Out Bus Element ports, based on the block label. This change occurs when the member access method option for inputs or outputs is configured as `Method` or `Inlined method`.

For example, the access method generated for this In Bus Element block is `setMyPort_MySignal` in R2024b but was `setMyBlock` in R2024a and earlier.

For more information about block names and labels, see Configure Model Element Names and Labels.

For more information about configuring member access method options, see Configure Model Data Elements as Class Members.

Code Generation

★Use std::vector containers to represent 1-D dynamic arrays

Starting in R2024b, the C++ code generator can generate `std::vector` containers to represent 1-D dynamic arrays. Previously, the C++ code generator could only generate `coder::array` containers to represent 1-D dynamic arrays. Generate `std::vector` containers to integrate generated code with external code that uses standard C++ libraries.

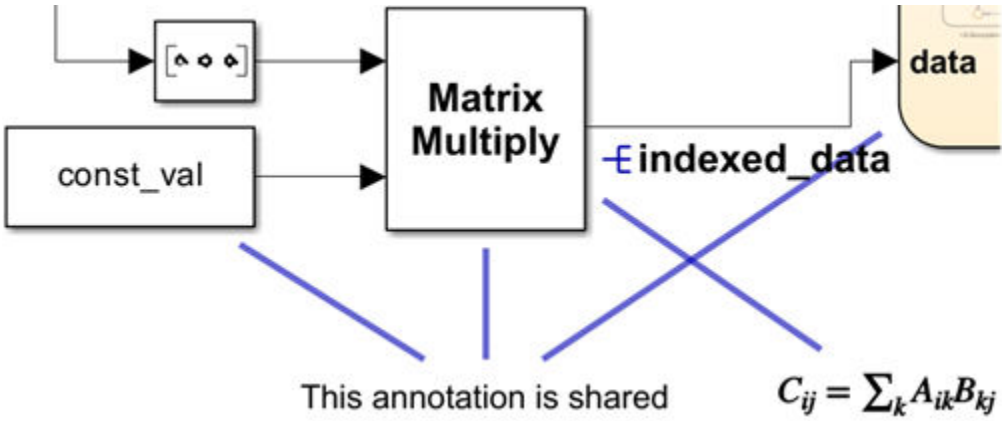
Use the new **Dynamic array container type** model configuration parameter to select between `coder::array` and `std::vector`. New models default to `std::vector`. Models from previous versions default to `coder::array` to preserve existing behavior.

To distinguish from the new parameter, the **Array container type** model configuration parameter is now named **Static array container type**. The programmatic name of that parameter is still `ArrayContainerType`.

Include connected annotations as block comments

Starting in R2024b, you can select the **Connected annotations in block comments** model configuration parameter to render annotations connected to blocks as comments in the generated code.

For instance, consider the following section of a model, where annotations are connected to Constant, Matrix Multiply, and Stateflow blocks.



When you enable this parameter, the code generator renders annotations connected to the three blocks as comments in the code. These comments appear alongside comments generated from block descriptions.

Generated code in R2024b

```
/* Product: '<Root>/Product' incorporates:
   *  Constant: '<Root>/Constant'
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   *
   * Block description for '<Root>/Product':
   *  This block determines the output index by taking the inner product of
   *  the vector of inputs and [1 2 3 4].
   *
   * Annotations for '<Root>/Product':
   *  $C_{ij} = \sum_k A_{ik} B_{kj}$
   *  This annotation is shared
   *
   *
   * Annotations for '<Root>/Constant':
   *  This annotation is shared
   */
  indexed_data = ((rtU.In1 * const_val[0] + rtU.In2 * const_val[1]) + rtU.In3 *
                  const_val[2]) + rtU.In4 * const_val[3];

  /* Chart: '<Root>/Stateflow'
   *
   * Block description for '<Root>/Stateflow':
   *  Stateflow block decides the index output by following logic
   *  1. If one and only one input signal is enabled, the index will
   *  refer to the enabled signal line.
   *  2. Otherwise, maintain the old value.
   *
   * Annotations for '<Root>/Stateflow':
   *  This annotation is shared
   */
```

For more information about this example, see Generate Custom Comments from Block Annotations.

Code generation support for fixed-point data types greater than 128 bits

You can now generate code for fixed-point data types with word lengths up to 65,535 bits. Prior to R2024b, the maximum word length was 128 bits.

For more information, see Supported Data Types (Fixed-Point Designer).

⚠Replace real-time model macros with member methods

Starting in R2024b, Embedded Coder uses member methods in place of macros to access real-time model (RTM) data structures in the generated C++ code, in compliance with MISRA Rule 16-0-4. For details on RTM data structures and a list of access functions, see Real-Time Model Data Structure.

Consider a model involving a simple S-function block `timestwo`:

The following table compares the C++ code generated in R2024a and R2024b. The model header file generated in R2024a defines function-like macros that are MISRA (Rule 16-0-4) noncompliant. In

R2024b, the code generator replaces these with RTM member methods defined in the model source file.

Generated code in R2024a	Generated code in R2024b

Model header file:
`// Macros for accessing real-time model data structure`
`#ifndef rtmGetErrorStatus`
`#define rtmGetErrorStatus(rtm)  ((rtm)->errorStatus)`
`#endif`
`#ifndef rtmSetErrorStatus`
`#define rtmSetErrorStatus(rtm, val)  ((rtm)->errorStatus = (val))`
`#endif`
`// Class declaration for model timesTwo`
`class timesTwo final`
`{`
`  // public data and function members`
` public:`
`  // External outputs (root outports fed by signals with default storage)`
`  struct ExtY_timesTwo_T {`
`    real_T Outport;  // '<Root>/Outport'`
`  };`
`  // Real-time Model Data Structure`
`  struct RT_MODEL_timesTwo_T {`
`    const char_T * volatile errorStatus;`
`  };`
	Model header file:
	`// Class declaration for model timesTwo`
	`class timesTwo final`
	`{`
	`  // public data and function members`
	` public:`
	`  // External outputs (root outports fed by signals with default s`
	`  struct ExtY_timesTwo_T {`
	`    real_T Outport;  // '<Root>/Outport'`
	`  };`
	`  // Real-time Model Data Structure`
	`  struct RT_MODEL_timesTwo_T {`
	`    const char_T * volatile errorStatus;`
	`    const char_T* getErrorStatus() const;`
	`    void setErrorStatus(const char_T* const volatile aErrorStatus)`
	`  };`

Model source file:	Model source file:
	`const char_T* timesTwo::RT_MODEL_timesTwo_T::getError`
	`{`
	`  return (errorStatus);`
	`}`
	`void timesTwo::RT_MODEL_timesTwo_T::setErrorStatus(co`
	`  aErrorStatus)`
	`{`
	`  (errorStatus = aErrorStatus);`
	`}`

Example main file:
`  while (rtmGetErrorStatus(timesTwo_Obj.getRTM()) == (`
`    //  Perform application tasks here`
`  }`
	Example main file:
	`  while (timesTwo_Obj.getRTM()->getErrorStatus() == (`
	`    //  Perform application tasks here`
	`  }`

Compatibility Considerations

As RTM macros are no longer defined in the model header file, calling these macros in the generated C++ code can cause compilation failure.

Currently, only `ert.tlc` and `autosar_adaptive.tlc` system target files support this functionality. Additionally, external mode code generation is currently not supported. Enabling the **External mode** configuration parameter can lead to uncompilable C++ code.

Halide code generation for Neighborhood Processing Subsystem

Starting in R2024b, the code generator supports generating Halide code for the Neighborhood Processing Subsystem for certain blocks inside the Neighborhood Processing Subsystem block. For list of blocks, see Supported Blocks for Halide Code Generation in Neighborhood Processing Subsystem.

For more information, see Speed Up Generated Code Execution with Halide Code.

Expanded C++ code generation support for referenced models that use port-scoped functions

To expand support for referenced models that use port-scoped functions, C++ code generation now uses pointers for the port classes in the constructor of a model and sets the pointers during model initialization. Previously code generation used references to the service port classes, which did not allow dependency cycles between model references. For more information about port-scoped functions, see Scoped, Global, and Port-Scoped Simulink Function Blocks Overview.

⚠Functionality being removed or changed

Dynamic array code generation produces an error if Support: variable-size signal model configuration parameter is cleared

Behavior change

Before R2024b, the code generator generated dynamically-allocated arrays regardless of the value of the **Support: variable-size signals** model configuration parameter. Starting in R2024b, the code generator throws an error if you generate code for a model that uses unbounded variable-size signals and has the **Support: variable-size signal** parameter cleared.

Deployment

★Deploy AUTOSAR adaptive applications on ARM 64 targets

Starting in R2024b, you can deploy AUTOSAR adaptive applications on ARM 64 target machines by using the Embedded Coder Support Package for Linux Applications.

For more information, see Build Simulink Model and Deploy Application.

Include or exclude referenced model elements in ASAP2 file

Starting in R2024b, the Generate Calibration Files tool allows you to exclude or include referenced model elements in the generated ASAP2 file by selecting tool option **Include referenced models**.

For more information, see Customize Generated ASAP2 File.

Generate application packages containing executables

Starting in R2024b, the Linux Runtime Manager allows you to generate application packages containing executables without generating source code by clicking the **Create & Deploy Application Package > Create** button.

⚠Functionality being removed or changed

RTE naming convention for ASAP2 elements for AUTOSAR classic models will change in a future release
Behavior change in future release

Starting in R2024b, the RTE elements in the generated ASAP2 file for AUTOSAR classic models will use RTE API service call names.

Performance

★Perform overhead-free execution-time profiling of generated code

If you set up your target hardware to generate a code execution trace, you can use the `coder.profile.trace` API to implement, register, and execute a parser that processes the trace. Running a PIL simulation on your target hardware produces the trace. When the simulation is complete, run the parser, which creates a `coder.profile.ExecutionTime` object that contains execution-time metrics. This profiling approach does not instrument generated code, so the execution-time metrics do not contain overheads associated with instrumentation. Use the Code Profile Analyzer to view the overhead-free execution-time metrics.

For more information, see Perform Instrumentation-Free Profiling Using Hardware Execution Tracers and Remove Instrumentation Overheads from Execution Time Measurements.

★Generate SIMD code for additional blocks, data types, and reduction operations

In R2024b, you can generate SIMD code for:

- Relational Operator blocks.
- Integer types for some AVX512F instructions.
- Complex data types with integer base types for some addition, subtraction, multiplication, and division operations. This new data type support applies to operations that do not generate `for`-loops without SIMD enabled, such as operations on vector inputs with sizes below the loop unrolling threshold.
- Reduction operations when the model configuration parameter **Support: non-finite numbers** is selected.

  Using SIMD instructions and enabling support for nonfinite numbers at the same time can produce numerical mismatches between the simulation and the generated code.

For more information, see Generate SIMD Code from Simulink Blocks for Intel Platforms, Generate SIMD Code from Simulink Blocks for ARM Platforms, and Optimize Code for Reduction Operations by Using SIMD.

★Perform task validation using Code Profile Analyzer

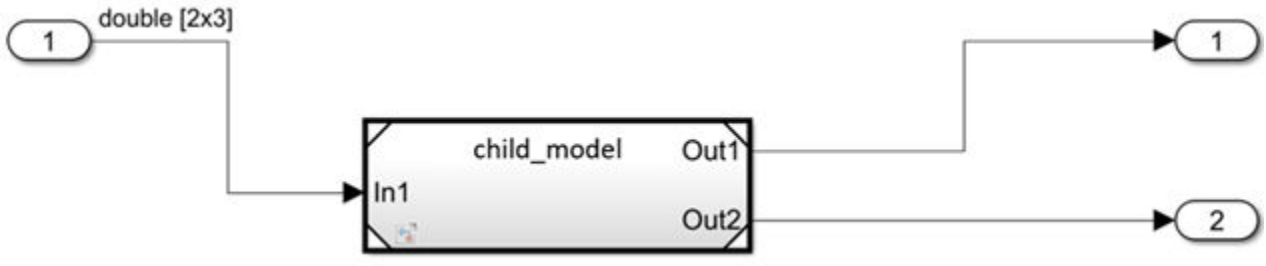
Using task execution times from a SIL, PIL, or XCP-based external mode simulation, the Code Profile Analyzer enables you to:

1. Map tasks to CPUs, applying well-known algorithms.
2. Validate CPU utilization for the task-to-CPU mapping.

The validation process helps you to detect CPU utilization issues for generated tasks at design time. For more information, see Validate Task-to-CPU Mapping.

Optimize generated code for models that include referenced models using Function Prototype Control

Starting in R2024b, the code generator optimizes code for Simulink models that include referenced models that use Function Prototype Control (FPC). The code generator produces more efficient code by reducing data copies and the stack size. Consider this `top_model`, which references the model `child_model`.



This code, generated in R2024a, contains an unnecessary data copy where the values from the model reference output `rtb_Model_o2` are copied to the local variable `arg_Out2`.

```
/* Model step function */
void top_model(real_T arg_inout1[6], real_T arg_Out2[6])
{
  /* local block i/o variables */
  real_T rtb_Model_o2[6];
  int32_T i;

  /* ModelReference: '<Root>/Model' */
  child_model_custom (&arg_inout1[0], &rtb_Model_o2[0]);

  /* Outport: '<Root>/Out2' incorporates:
   *  ModelReference: '<Root>/Model'
   */
  for (i = 0; i < 6; i++) {
    arg_Out2[i] = rtb_Model_o2[i];
  }
```

This code, generated in R2024b, uses the data from the `child_model_custom` function without creating a separate copy, reducing data duplication and stack size.

```
/* Model step function */
void top_model(real_T arg_inout1[6], real_T arg_Out2[6])
{
  /* Outport: '<Root>/Out2' incorporates:
   *  ModelReference: '<Root>/Model'
   */
  child_model_custom (&arg_inout1[0], &arg_Out2[0]);
}
```

For more information on model references, see Generate Code for Model Reference Hierarchy, Function Prototyping and Support C Function Prototype Control.

Additional metrics for generated code profiling

When you run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations on Linux-based target hardware, the new model parameter `CodeProfilingCodeMetrics` enables you to profile generated code using advanced metrics from the Performance Monitoring Unit (PMU) of the hardware.

Model Parameter	Value	Metric Extracted from PMU
`CodeProfilingCodeMetrics`	`'time'` (default)	N/A
	`'totinstr'`	Total instruction count
	`'floatinsts'`	FPU instruction count
	`'floatopts'`	FPU operation count
	`'intinstr'`	ALU instruction count
	`'loadinstr'`	Load instruction count
	`'storeinstr'`	Store instruction count
	`'totcyc'`	Total cycle count
	`'l1dcm'`	L1 data cache misses
	`'l1icm'`	L1 instruction cache misses
	`'l2dcm'`	L2 data cache misses
	`'l2icm'`	L2 instruction cache misses
	`'tlbdm'`	TLB data misses
	`'tlbim'`	TLB instruction misses
	`'memstall'`	Memory stall cycles
	`'totstall'`	Total stall cycles

Using the Code Profile Analyzer, you can view and analyze the profiles created. If your target hardware is not supported natively, you can register custom drivers to extract the required metrics from the PMU. For more information, see Generate Profiles Using PMU Metrics.

Use code replacements with data alignment for data that uses storage classes

In R2024b, when you use code replacement for Simulink code generation, you can use data alignment specifications for data that use most storage classes, including storage classes that you create. Previously, only a limited set of storage classes supported data alignment.

For more information, see Optimize Performance of Memory Access by Using Data Alignment and Data Alignment for Code Replacement.

Optimize generated code for deep learning network models

Starting in R2024b, you can use the **Generate parallel for-loops**, **Automatically schedule for-loops**, and **Generate Halide code** model configuration parameters to enable optimizations in generated code for deep learning network models that use the new `Deep Learning Layers` block library in Deep Learning Toolbox.

For more information about the `Deep Learning Layers` block library, see “Simulink: New Deep Learning Layers block library and exportNetworkToSimulink function” (Deep Learning Toolbox).

Generate SIMD code for ARM Cortex-A hardware boards by using configuration parameters

In R2024b, you can generate SIMD code for **Hardware board** targets that have the model configuration parameter **Device type** set to `ARM Cortex-A (32-bit)` or `ARM Cortex-A (64-bit)` by setting the parameter **Leverage target hardware instruction set extensions** to `Neon v7`. Before R2024b, to generate SIMD instructions for these targets, you had to use a GCC ARM Cortex-A code replacement library. For more information, see Generate SIMD Code from Simulink Blocks for ARM Platforms.

Compatibility Considerations

For more information about changes to ARM Cortex-A optimization options, see “ARM Cortex-A Processors: Code replacement library GCC ARM Cortex-A removed” on page 4-22.

Generate SIMD code for Apple platforms by using configuration parameters

In R2024b, you can generate SIMD code for Apple platforms by setting the model configuration parameter **Leverage target hardware instruction set extensions** to `Neon v7`. The parameter supports SIMD code generation for `maca64` host computers and hardware targets with parameters **Device vendor** set to `Apple` and **Device type** set to `ARM64`. Before R2024b, to generate SIMD instructions for these targets, you had to use a GCC ARM Cortex-A code replacement library. For more information, see Generate SIMD Code from Simulink Blocks for Apple silicon Platforms.

Compatibility Considerations

For more information about changes to Apple optimization options, see “ARM Cortex-A Processors: Code replacement library GCC ARM Cortex-A removed” on page 4-22.

Generate SIMD code for control flow constructs on ARM Cortex-A platforms

In R2024b, you can generate SIMD code for control flow constructs for ARM Cortex-A platforms. SIMD code generation is supported for loops with control flow constructs that the code generator can optimize by converting to loops without control flow constructs. For more information, see Generate SIMD Code from Simulink Blocks for ARM Platforms.

Functionality being removed or changed

`timeline` function has been removed
Errors

The `timeline` function has been removed. Run the `schedule` function to visualize task scheduling using the Simulation Data Inspector.

Hardware Support

★Embedded Coder Support Package for Qualcomm Hexagon Processors: Generate optimized code using Qualcomm Hexagon Library (QHL) and Hexagon Vector eXtension (HVX)

The Embedded Coder Support Package for Qualcomm Hexagon Processors is available from release R2024b onwards. You can use the support package to generate efficient code using Qualcomm Hexagon Library (QHL scalar processor) and Hexagon Vector eXtension (HVX vector processor) for the Hexagon Digital Signal Processor (DSP).

The first release of this support package includes:

- Support code replacement library (CRL) for DSP Simulink blocks, DSP System object, Math Operators, and Math functions. For more information, see Code Optimization for QHL and Code Optimization for HVX.
- Examples that demonstrate the capabilities of the support package:
  - Getting Started with Embedded Coder Support Package for Qualcomm Hexagon Processors
  - Code Verification and Validation with PIL
  - Parametric Audio Equalizer for Qualcomm Hexagon DSP

For more information on support package, see Qualcomm Hexagon Processors.

★STMicroelectronics STM32 Processors: Support for STM32F2xx and advanced peripherals for STM32F3xx- and STM32H7xx- (dual-core) based boards

Embedded Coder Support Package for STMicroelectronics STM32 Processors now supports the new STM32F2xx-based board and advanced peripherals for STM32F3xx-based boards and STM32H7xx-dual core based boards. Starting in R2024b, you can:

- Generate and build code using an STM32CubeMX project file.
- Use the Analog to Digital Converter, CAN Read, CAN Write, Digital to Analog Converter, Encoder, I2C Controller Read, I2C Controller Write, I2S Audio Out, I2S Mic In, PWM Output, Digital Port Read, Digital Port Write, Hardware Interrupt, Timer, SPI Receive, SPI Transmit, SPI Controller Transfer, UDP Receive, UDP Send, and UART/USART Read blocks to implement Model-Based Design workflows for STM32F2xx based boards.
- Use the advanced peripherals blocks like CAN Read, CAN Write, Comparator, I2C Controller Read, I2C Controller Write, I2S Audio Out, I2S Mic In, SPI Receive, SPI Transmit, SPI Controller Transfer, and Higher Resolution Timer blocks to implement Model-Based Design workflows for STM32F3xx based boards.
- Use the advanced peripherals blocks like FDCAN Read, FDCAN Write, Comparator, CORDIC co-processor, Digital to Analog Converter, I2C Controller Read, I2C Controller Write, I2S Audio Out, I2S Mic In, SPI Receive, SPI Transmit, SPI Controller Transfer, and Higher Resolution Timer blocks to implement Model-Based Design workflows for STM32H7xx (Dual core) based boards.
- Monitor signals and tune parameters in external mode. For more information, see Monitoring and Tuning Using STMicroelectronics STM32 Processor Based Boards.

- Run processor-in-the-loop (PIL) simulation in the serial communication mode.

STMicroelectronics STM32 Processors: External mode simulation using XCP on CAN interface

Starting in R2024b, you can configure a model for simulating in external mode to perform signal logging and parameter tuning using XCP on CAN in Embedded Coder Support Package for STMicroelectronics STM32 Processors. For more information, see Signal Monitoring and Parameter Tuning Over XCP-based CAN Interface Using STM32 Processors.

STMicroelectronics STM32 Processors: Data logging on SD card

You can use Embedded Coder Support Package for STMicroelectronics STM32 Processors to log data from Simulink blocks to an SD card mounted on STM32F2xx, STM32F4xx, STM32F7xx, STM32H7xx, and STM32L4xx based boards. You can save the data values in three formats: structure, structure with time, or array. The data is logged to a MAT file. For more information, see MAT-file Logging on SD Card for STMicroelectronics STM32 Processors.

STMicroelectronics STM32 Processors: Communicate with STM32 Processors in Normal mode simulation using Connected IO

Starting in R2024b, the Embedded Coder Support Package for STMicroelectronics STM32 Processors supports connected IO mode simulation with the STM32 processor-based boards. When you simulate a model in connected IO mode, the model communicates with the IO peripherals on the hardware, enabling you to verify the model design before deploying it on the hardware.

The STM32 processor supports connected IO during normal mode simulation for these peripherals blocks:

- PWM Output
- Analog to Digital Converter
- Digital Port Read
- Digital Port Write

STMicroelectronics STM32 Processors: Enhancements to Digital to Analog Converter block

Starting in R2024b, with the DAC block, now you can disable the DAC module after enabling it.

Additionally the block outputs the status indicating:

- 0: DAC has been enabled and is operating as expected
- 1: DAC has been disabled successfully
- 2: error occurred during command execution

Use the updated Digital to Analog Converter block to convert a digital value to its equivalent analog voltage on the specified channel.

STMicroelectronics STM32 Processors: Create Simulink block for custom or third-party C/C++ files

Starting in R2024b, you can use the IO Device Builder app in Embedded Coder Support Package for STMicroelectronics STM32 Processors to create a System object for custom or third-party source files (C/C++). You can create a Simulink block from the generated System object by using the MATLAB System block and add the Simulink block to your model.

To use the IO Device Builder app, on the **Hardware** tab of the Simulink toolstrip, navigate to the **Prepare** section, and under **Design**, select `IO Device Builder`. A series of screens then leads you through the process of creating a System object and a Simulink block. For more information, see Create Real-Time Clock Block to Display Time Using IO Device Builder App and Get Started with IO Device Builder.

STMicroelectronics STM32 Processors: Support for ARM GNU Toolchain 13.2.Rel1

Embedded Coder Support Package for STMicroelectronics STM32 Processors now supports the ARM GNU Toolchain 13.2.Rel1.

STMicroelectronics STM32 Processors: New examples that demonstrate the capabilities of STM32 Processors

Use the following new examples to explore capabilities of the Embedded Coder Support Package for STMicroelectronics STM32 Processors.

- Field-Oriented Control of PMSM Using Position Estimated by Neural Network on STM32 Processor Based Boards - This example shows how to implement field-oriented control (FOC) of a permanent magnet synchronous motor (PMSM) using rotor position estimated by a deep-learning system implemented using a simple non-linear auto regressive neural network (ARNN).
- Field-Oriented Control of PMSM with Hall Sensor Using STM32G4xx Based Processors - This example implements the field-oriented control (FOC) technique to control the speed of a three-phase permanent magnet synchronous motor (PMSM). The FOC algorithm requires rotor position feedback, which is obtained by a Hall sensor.
- Hall Offset Calibration for PMSM with STM32 Processors - This example calculates the offset between the rotor direct axis (d-axis) and position detected by the Hall sensor. The field-oriented control (FOC) algorithm needs this position offset to run the permanent magnet synchronous motor (PMSM) correctly.
- Getting Started with Hardware Profiling - This example demonstrates how to perform real-time execution profiling of algorithms using the Embedded Coder Support Package for STMicroelectronics STM32 Processors.
- Optimizing FreeRTOS Scheduling on STM32 Processors - This example demonstrates how to use the Embedded Coder Support Package for STMicroelectronics STM32 Processors to execute a Simulink model on the STMicroelectronics NUCLEO-F207ZG board. It guides you through implementing **FreeRTOS** scheduling within the STM32CubeMX workflow, enabling efficient task management and code execution.
- MAT-file Logging on SD Card for STMicroelectronics STM32 Processors - This example shows you how to perform MAT-file logging using Simulink model on a Micro SD card mounted on an STMicroelectronics STM32 processor based board.

- Processor-in-the-Loop Verification of MATLAB Functions Using STMicroelectronics STM32 Processors - This example shows you how to use the Embedded Coder Support Package for STMicroelectronics STM32 Processors for Processor-in-the-Loop (PIL) verification of MATLAB functions.

Infineon AURIX TC3x Microcontrollers: Support for Infineon low-level driver (iLLD) 1.0.1.17.0

Starting in R2024b, you can use the Infineon low-level driver (iLLD) 1.0.1.17.0 with Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers.

Infineon AURIX TC3x Microcontrollers: New EDSADC block to measure voltage using delta-sigma conversion

Use the new Delta-Sigma Analog-to-Digital Converter (EDSADC) block from Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers to measure the voltage of an analog input pin using delta-sigma conversion. After adding the block to the model, you can map the block parameters to the hardware using the EDSADC Peripheral Configuration tool.

Infineon AURIX TC3x Microcontrollers: New SENT block to read high-resolution sensor data over SENT protocol

Use the new Single Edge Nibble Transmission (SENT) block from Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers to read high-resolution sensor data transmitted over the SENT protocol. After adding the block to the model, you can map the block parameters to the hardware using the SENT Configuration tool.

Infineon AURIX TC3x Microcontrollers: Multicore support

The Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers now supports multicore workflow. Use the TriCore processing units of Infineon AURIX TC3x microcontrollers with Interprocess Data Read, Interprocess Data Write, Interprocess Data Channel, Task Manager, and peripheral blocks from the AURIX TC3x library for multicore modelling, PIL simulation, external mode of simulation, and deployment.

Infineon AURIX TC4x Microcontrollers: SoC Blockset Support Package for Infineon AURIX Microcontrollers moved to Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers

Starting in R2024b, the SoC Blockset™ Support Package for Infineon AURIX Microcontrollers is moved to the Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers. You can now perform multicore modeling and generate hardware-specific code for the PPU core using Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Support for Infineon low-level driver (iLLD) 2.0.1.3.5

Starting in R2024b, you can use the Infineon low-level driver (iLLD) 2.0.1.3.5 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Support for Green Hills MULTI v2023.5

Starting in R2024b, you can use the Green Hills MULTI v2023.5 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Support for TASKING Smartcode v10.2r1

Starting in R2024b, you can use the TASKING Smartcode v10.2r1 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: New MCAN blocks for CAN communication

Use the new MCAN Transmit block from Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to write data to a CAN bus. If you configure the block to write data in CAN message format, then depending on the frame format, you must use the CAN Pack or the CAN FD Pack block with the MCAN Transmit block. After adding the block to the model, you can map the block parameters to the hardware using the MCAN Transmit Peripheral Configuration tool.

Use the new MCAN Receive block from Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to read data from the CAN bus. If you configure the block to receive packed CAN data, then depending on the frame format, you must use the CAN Unpack or CAN FD Unpack block with the MCAN Receive block. After adding the block to the model, you can map the block parameters to the hardware using the MCAN Receive Peripheral Configuration tool.

Infineon AURIX TC4x Microcontrollers: Enhancements to CSDP block

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers includes these enhancements to the CDSP block:

- The **Filter chain** parameter is renamed to **DSP filter chain**. You can set the **DSP filter chain** parameter to one of these values:

DSP filter chain	Filter Blocks to Use in Predefined Sequence
`Demo FIR (FC0)`	Finite impulse response (FIR)
`Advanced average1 (FC1)`	FIR > (Infinite impulse response filter of order 6) IIR6 > Math filter (MAT) > Average filter (AVG)

DSP filter chain	Filter Blocks to Use in Predefined Sequence
`Basic data accumulation (FC2)`	FIR > IIR6 > MAT > Data accumulation filter (DA)
`Advanced median (FC3)`	FIR > IIR6 > Median filter (MDN)
`Advanced average 2 (FC4)`	IIR6 > FIR > MAT > AVG
`Basic AURIX (FCM)`	FIR > IIR1 > DA

- The block adds the **FIR, IIR6, IIR1, MAT, AVG**, and **DA** tabs in the block parameters window depending on the value of the **DSP filter chain** parameter. You can set the parameters of filter block in these tabs.
- You can bypass the filter blocks in their corresponding tabs in the block parameters window. For example, if you set the **DSP filter chain** parameter to `Advanced average 1 (FC1)`, then you can bypass the IIR6 filter block in the **IIR6** tab by enabling the **Bypass IIR6 block** parameter.
- For `Basic data accumulation (FC2)` and `Basic AURIX (FCM)` DSP filter chains, you can configure the DA filter block parameters in **Integration Control** tab of CDSP Peripheral Configuration.

Infineon AURIX TC4x Microcontrollers: Support for Infineon AURIX TC4Dx hardware boards

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now supports Infineon AURIX TC4Dx hardware boards. The parallel processing unit (PPU) core and peripheral blocks from the AURIX TC4x library are not supported for Infineon AURIX TC4Dx hardware boards. You can use the homogenous TriCores with Interprocess Data Read, Interprocess Data Write, Interprocess Data Channel, and Task Manager blocks for modeling, PIL simulation, external mode of simulation, and deployment.

Infineon AURIX TC4x Microcontrollers: CRL support for tensor multiplications in deep learning network layers

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers supports `MetaWare TC4x PPU for Deep Learning` code replacement library (CRL). Use this CRL to generated optimized code for tensor multiplications in Fully Connected, LSTM, BiLSTM , and GRU layers of deep learning networks implemented on PPU core.

Xilinx Zynq Hardware: Support package product name change

In R2024b, the Embedded Coder Support Package for Xilinx® Zynq®-7000 Platform product has been renamed as Embedded Coder Support Package for AMD SoC Devices. This name change conveys added support for AMD® products.

ARM Cortex-M Processors: Support for ARM GNU Toolchain 13.2.Rel1

Embedded Coder Support Package for ARM Cortex-M Processors now supports for ARM GNU Toolchain 13.2.Rel1.

ARM Cortex-A Processors: CMSIS CRL support for complex math functions, basic vector functions, and Discrete FIR Filter block

Starting in R2024b, you can use Embedded Coder Support Package for ARM Cortex-A Processors to generate code for these complex math functions and vector functions using the ARM Cortex-A CMSIS code replacement library (CRL).

- Complex Dot Product
- Complex Multiplication
- Vector Scale
- Vector Offset

Additionally, you can generate code for these Simulink blocks using the ARM Cortex-A CMSIS CRL.

- Discrete FIR Filter
- Saturate

For more information see:

- Supported CMSIS Library Functions for ARM Cortex -A Processors
- Supported DSP System Toolbox Blocks with CMSIS Library for ARM Cortex -A Processors
- Supported DSP System Toolbox System Objects with CMSIS Library for ARM Cortex -A Processors

ARM Cortex-A Processors: Generate and deploy optimized code for interpolated FIR filter on Raspberry Pi using ARM Cortex-A CMSIS CRL

The Generate and Deploy Optimized Code for Interpolated FIR Filter on Raspberry Pi using ARM Cortex-A CMSIS CRL example shows how to generate and deploy optimized code for an interpolated finite impulse response filter (IFIR) on a Raspberry Pi target using ARM Cortex-A CMSIS CRL.

ARM Cortex-M Processors: CMSIS CRL support for matrix multiplication and transpose

Starting in R2024b, you can use the Embedded Coder Support Package for ARM Cortex-M Processors to generate code for these Simulink blocks and MATLAB functions using the ARM Cortex-M CMSIS CRL.

- Matrix Multiplication
- Matrix Transpose

The ARM Cortex-M CMSIS CRL generates optimal code for the supported vector lengths in MATLAB. If the CRL outperforms other methods for certain vector lengths, then the ARM Cortex-M CMSIS CRL generates CRL code. In other cases, the ARM Cortex-M CMSIS CRL generates C code. For more information, see Supported Simulink Blocks with CMSIS Library for ARM Cortex -M Processors.

ARM Cortex-M Processors: Code generation for interpolated FIR filter on ARM Cortex-M target using CMSIS

The Code Generation for Interpolated FIR Filter on ARM Cortex-M Target using CMSIS example shows how to generate and run optimized code using ARM Cortex-M CRL for an interpolated finite impulse response (IFIR) filter on the STM32F746G-Discovery hardware.

ARM Cortex-M Processors: Deploy parametric audio equalizer on ARM Cortex-M processors

The Deploy Parametric Audio Equalizer on ARM Cortex-M Processors example shows how to deploy three-band parametric audio equalizer on the ARM Cortex-M™ STM32F746G-Discovery board. You can deploy this parametric equalizer on any supported ARM Cortex-M hardware.

ARM Cortex-M Processors: Generate code and deploy acoustic-based machine fault detection using deep learning on ARM Cortex-M hardware

The Code Generation and Deployment for Acoustic-Based Machine Fault Detection using Deep Learning on ARM Cortex-M Hardware example shows how to generate code for acoustics-based machine fault detection using long short term memory (LSTM) network and spectral descriptors. This example uses deep learning support to generate a processor in the loop (PIL) executable function which is then deployed on ARM Cortex-M hardware.

⚠Functionality being removed or changed

ARM Cortex-A Processors: Code replacement library GCC ARM Cortex-A removed

Behavior change

The code replacement library `GCC ARM Cortex-A` is no longer visible in the code replacement library selector. To use the optimizations that were in this code replacement library:

- For SIMD optimizations, use the model configuration parameter **Leverage target hardware instruction set extensions**.
- For other optimizations, set the parameter **Code replacement libraries** to `ARM Cortex-A CMSIS`.

Models that already used the code replacement library still generate code that contains the replacements from the library.

Check bug reports for issues and fixes

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

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

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

R2024a

Code Generation from MATLAB Code

★Generate SIMD instructions by default for Intel hardware

MATLAB Coder generates SIMD instructions by default when you generate non-MEX code for Intel hardware. The new default value of the **Leverage target hardware instruction set extensions** configuration parameter is `Auto`. MATLAB Coder resolves the value to `SSE2` for Intel hardware. For more information, see Generate SIMD Code from MATLAB Functions for Intel Platforms.

Code Interface Configuration and Integration

★Configure multirate, rate-based component models to use service code interfaces

You can configure nonpartitioned multirate rate-based component models to use service code interfaces. Before R2024a, you could configure only export-function and single-rate, rate-based component models to use service code interfaces. Service code interfaces do not support partitioning.

For an example, see Deploy Multirate Rate-Based Component Configured for C Service Interface Code Generation. For background and high-level workflow information, see Embedded Coder Fundamentals and Service Interfaces.

Timer service interface support for 64-bit time values

For models configured to use a service code interface:

- The code generator supports 64-bit time values, using an unsigned 64-bit integer data type.
- The default value for the configuration parameter **Application lifespan (days)** is `inf`. When the parameter is set to `inf`, the code generator returns a 64-bit integer for a timer interface function call, honoring the 64-bit integer data type requirement for continuous execution.
- If the target hardware does not support 64-bit integers:
  - The code generator represents a 64-bit timer as a multiword unsigned 64-bit integer data type.
  - You can generate a timer service for a 32-bit timer with an unsigned 32-bit integer data type by setting model configuration parameters **Application lifespan (days)** and **Clock resolution (seconds)** so that the product of the application lifespan setting and 86,400 divided by the clock resolution setting is less than or equal to `MAX_uint32_T`:
  ((*application-lifespan** 86400)/*clock-resolution*)≤MAX_uint32_T

Before R2024a, the data type of time values that could be exchanged between generated code and a target platform service was fixed at 32 bits (`uint32`). This fixed data type support for timers resulted in overflow conditions when the model configuration parameter **Application lifespan (days)** was set to `inf` to support continuous execution. The `inf` setting requires the new support for 64-bit time values.

For information on configuring a model for 64-bit integer support, see Optimize Memory Usage and Prevent Overflows for Time Counters.

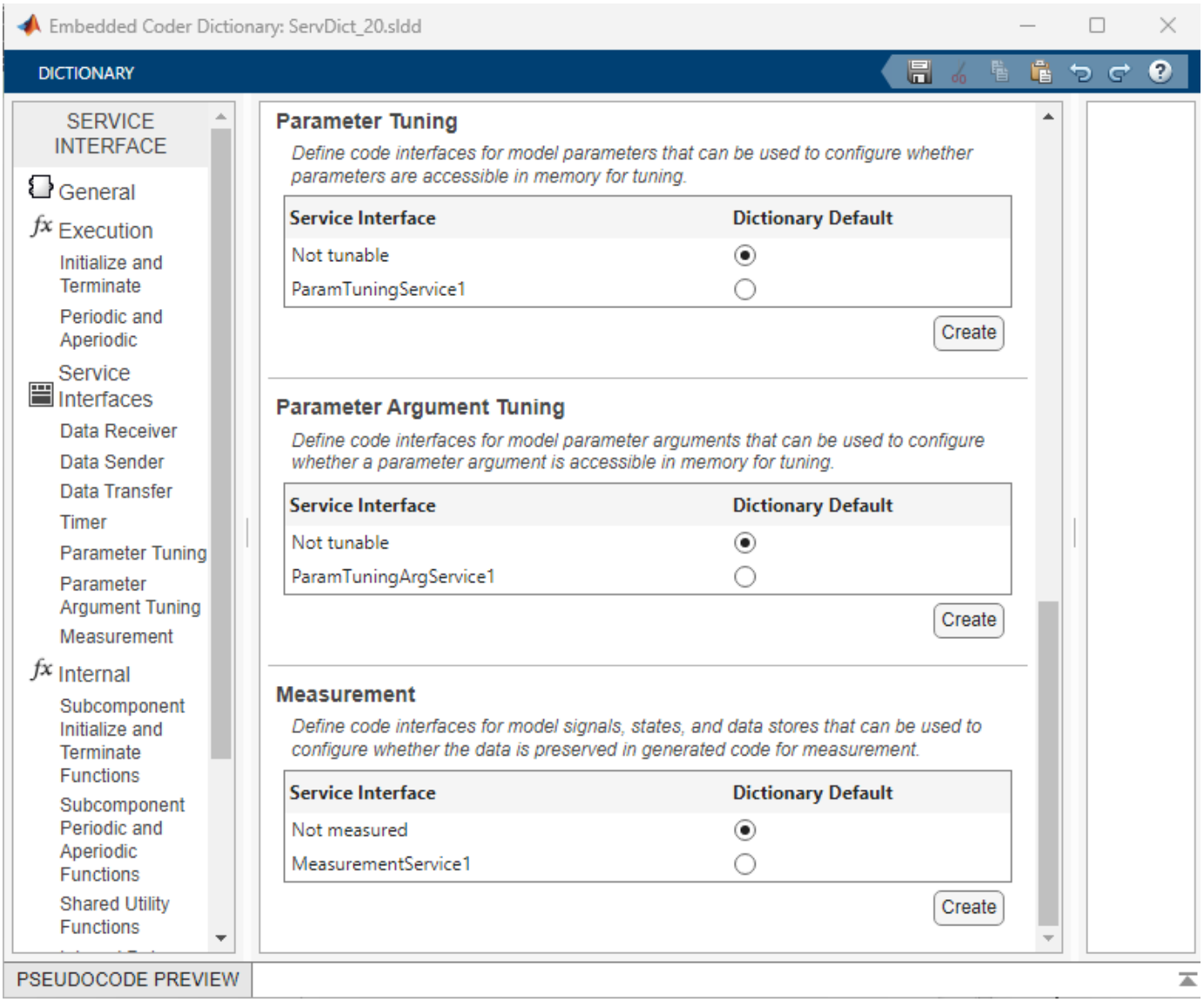
Optimize, by default, code generated from tunable and measurable elements of models configured with service interface configuration

In models configured to use service interfaces, when you configure model elements to be `Not tunable` or `Not measured`, the code generator attempts to optimize the generated code by using as few variables as possible to store the values of these elements.

Starting in R2024a, you can specify `Not measured` as the default value of the **Measurement** section in the coder dictionary to optimize, by default, measurable model elements you do not need to measure. This default value is applied to signals, states, and data stores that specify `Dictionary`

`Default` as their measurement service in the Code Mappings Editor – C (or the equivalent programmatic interface).

Similarly, you can now specify `Not tunable` as the default value of the **Parameter Tuning** and **Parameter Argument Tuning** sections in the coder dictionary to optimize, by default, tunable model elements you do not need to tune. This default value is applied to model parameters and model parameter arguments that specify `Dictionary Default` as their tuning service in the Code Mappings editor (or the equivalent programmatic interface).



The `Not measured` and `Not tunable` entries are included in the dictionary, and are preselected as default values in newly created dictionaries. You can select other entries as dictionary default for these sections, but you cannot delete these entries from the dictionary.

Starting in R2024a, the measurement and tuning service of newly added model elements is initially set to `Dictionary Default` in the Code Mappings editor.

Before R2024a:

- The measurement and tuning services of newly added model elements were initially set to `Not measured` and `Not tunable` respectively, in the Code Mappings editor.
- You could only configure individual model elements to be not tunable or not measured by specifying `Not measured` or `Not tunable` as their measurement or tuning service in the Code Mappings editor.

For more information about tuning model parameters and parameter arguments, see:

- Parameter Tuning Interfaces
- Parameter Argument Tuning Interfaces
- Configure Parameter and Parameter Argument Tuning Service Interfaces for Model Parameters and Model Parameter Arguments

For more information about measuring model elements, see:

- Measurement Service Interfaces
- Configure Measurement Service Interfaces for Signals, States, and Data Stores

Enhanced control of generated service code interfaces

Starting in R2024a, when using models that are configured with a service interface configuration, you can control the generated code interface of more element types. The Embedded Coder Dictionary contains new sections, and some of the existing dictionary sections now apply to more code elements than before.

This table summarizes code interface types that can now be managed with the new dictionary sections.

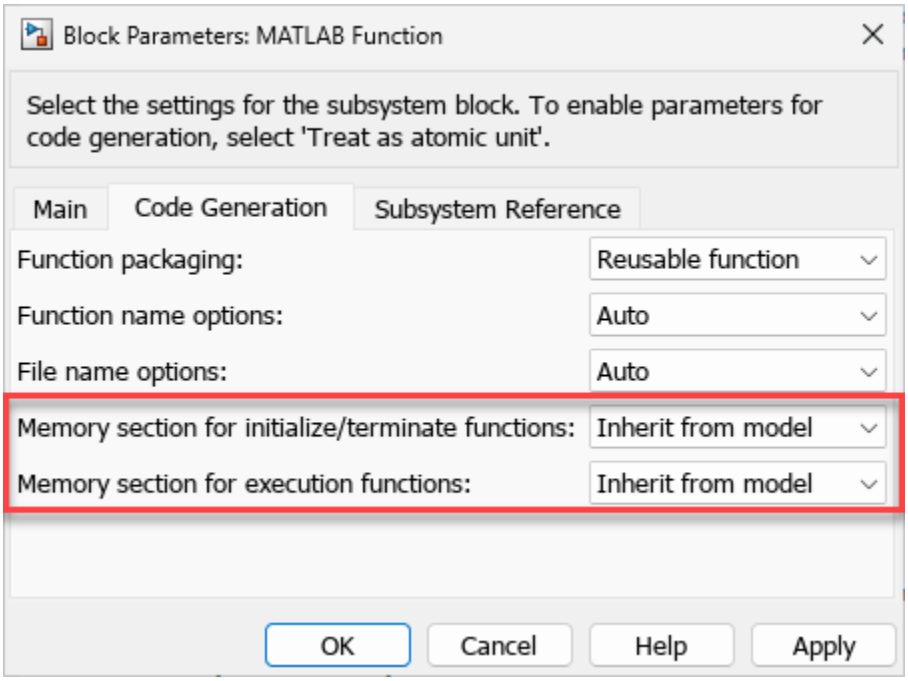
Generated Code Element Types	Applicable Interface Properties	Dictionary Section	Programmatic Interface Section Name
Constant variables that are not inlined	Memory section	Constants	`Constants`
Signals, states, and internal data	Memory section or storage class	Internal Data	`InternalData`
Initialize and terminate functions of subcomponent models	Function customization template	Subcomponent Initialize and Terminate Functions (formerly a subset of Subcomponent Functions)	`SubcomponentInitTermFunctions`
Periodic and aperiodic functions of subcomponent models	Function customization template	Subcomponent Periodic and Aperiodic Functions (formerly a subset of Subcomponent Functions)	`SubcomponentPeriodicAperiodicFunctions`

The default **Function Customization Template** specified in the Embedded Coder dictionary applies to entry-point functions of the function category that specifies `Dictionary default` as the

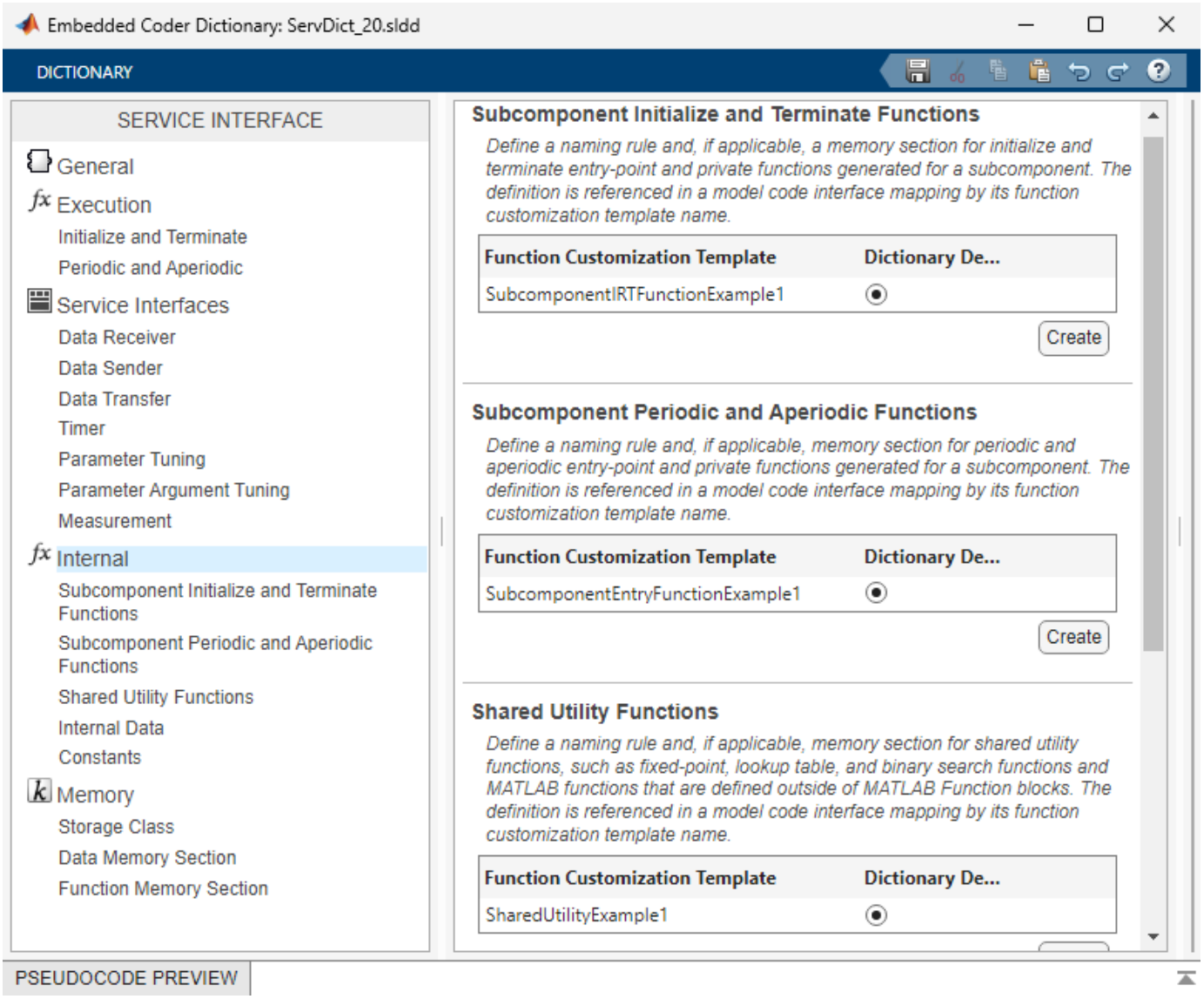
functions' customization template in the Code Mappings Editor – C (or the equivalent programmatic interface).

Starting in R2024a:

- By default, the specified template for entry-point functions in the Code Mappings editor is `Dictionary default`. You can use the Code Mappings editor (or the equivalent programmatic interface) to manually specify different customization templates for individual entry-point functions.
- The memory section of the default **Function Customization Template** specified in the dictionary applies to functions generated from atomic subsystem blocks that specify `Inherit From Model` as the memory section for the function category. This includes blocks that are handled in the same way as atomic subsystems during code generation, such as MATLAB Function blocks and Stateflow Chart blocks. To specify the memory sections in such blocks, select the **Code Generation** tab of the block parameters and specify `Reusable function` or `Reusable function` for the **Function packaging**.



To learn more about how to generate functions from subsystems, see Generate Subsystem Code as Separate Function and Files.



Functionality being removed or changed

Error reported when you load MPT objects last created or modified before R2012b
Behavior change

In R2024a, if you load a `.mat` file or a `.slx` file that contains `mpt.Signal` or `mpt.Parameter` objects that were last created or modified before R2012b, Simulink reports one of these error messages:

- `Class mpt.CustomRTWInfoSignal not supported in R2024a and removed in R2024b. Migrate instances of this class to the new supported class.`
- `Class mpt.CustomRTWInfoParameter not supported in R2024a and removed in R2024b. Migrate instances of this class to the new supported class.`

To resolve this error, load the `.mat` file or `.slx` file in a release later than R2012a and before R2024a. Then resave the file.

For more information on MPT objects, see MPT Data Object Properties.

Code Generation

★Automatically schedule generated for-loops for Neighborhood Processing Subsystem, Pixel Processing Subsystem, and Array Processing Subsystem blocks

Starting in R2024a, you can select the **Automatically schedule for-loops** model configuration parameter to generate optimized nested loop code for the Neighborhood Processing Subsystem, Pixel Processing Subsystem, and Array Processing Subsystem blocks. Use this parameter to generate code that efficiently processes large input data such as images and video.

For more information about this parameter, see Automatically schedule for-loops.

For an example involving a Neighborhood Processing Subsystem block, see Automatically Schedule `for`-Loops for Neighborhood Processing Subsystems.

Support for C99 and C++11 nonfinite data

Starting in R2024a, the code generator uses built-in C99 and C++11 language standard literals for nonfinite constants in the generated code instead of generated nonfinite constants. For C89/C90 and C++03, the code generator produces nonfinite constants that conform to the IEEE-754 standard. Examples of nonfinite constants are NaN and Inf.

Before R2024a, the code generator produced nonfinite constants per the IEEE-754 standard for supported C/C++ target language standards (C89/C90, C99, C++03, and C++11).

For more information about nonfinite number support, see Support: non-finite numbers.

Code generator uses std::atomic operations for rate-transition and task-transition modeling tasks

Starting in R2024a, the code generator uses `std::atomic` operations in the generated C++ code for managing the scheduling and execution of rate-transition and task-transition model tasks. C++11 atomic operations are supported for models configured with Language set to `C++` and Language standard set to `C++11 (ISO)`.

For more information about rate-transition modeling, see Rate Transition.

Specify .cpp and .hpp file extensions for exported storage class definitions

In R2024a, for built-in storage classes and storage classes defined in an Embedded Coder Dictionary, Simulink does not report edit-time or compile-time errors when you specify the storage class property `DefinitionFile` with a `.cpp` file extension. Also, you can specify the storage class property `HeaderFile` with a `.h` or `.hpp` file extension. The code generator generates the exported files as shown in the tables below. Previously, for a model configured for C++ code generation, when you used an exported storage class to generate a global variable definition or declaration to an external file, the specifications for `DefinitionFile` and `HeaderFile` required `.c` and `.h` file extensions, respectively. The code generator remapped the `.c` file extension to a `.cpp` file extension.

Results for `DefinitionFile`

Specified file extension	Previous result	Result in R2024a
`.cpp`	Edit-time error	`.cpp` file produced
`.c`	`.cpp` file produced	`.cpp` file produced

Results for `HeaderFile`

Specified file extension	Previous result	Result in R2024a
`.hpp`	Compile-time error	`.hpp` file produced
`.h`	`.h` file produced	`.h` file produced

If you export a model to a previous version of Simulink, the software maps a `DefinitionFile` property specified with a `.cpp` file extension to a `DefinitionFile` property with a `.c` file extension for successful code generation.

For more information on storage classes, see Choose Storage Class for Controlling Data Representation in Generated Code and Embedded Coder Dictionary.

Code generation support for function ports in single-instance referenced model

Starting in R2024a, the code generator supports single-instance referenced models that use function ports. Before R2024a, the code generator supported function ports only in referenced models that were configured as multi-instance.

For more information about modeling communication by using function ports and generating code, see Model Client-Server Communication Using Function Ports and Client-Server Communication Interfaces.

Generate code for Width block with symbolic dimension inputs

Starting in R2024a, you can generate code for the Width block that has symbolic dimensions as inputs. Before R2024a, code generation for Width blocks that had symbolic dimensions as inputs was not supported.

For more information, see Implement Dimension Variants for Array Sizes in Generated Code.

Enhancements to Halide code generation

Starting in R2024a, the code generator has several enhancements to support Halide code generation:

- The code generator supports generating Halide code for MATLAB Function blocks with vector inputs for arithmetic operations such as addition, subtraction, element-wise multiplication, and division. In R2023b, support was limited to matrix multiplication only.
- The code generator supports the tensor multiplication operation used in deep learning neural network models.
- The code generator supports generating Halide code with the target language set to C. In R2023b, it supported only C++ as the target language.

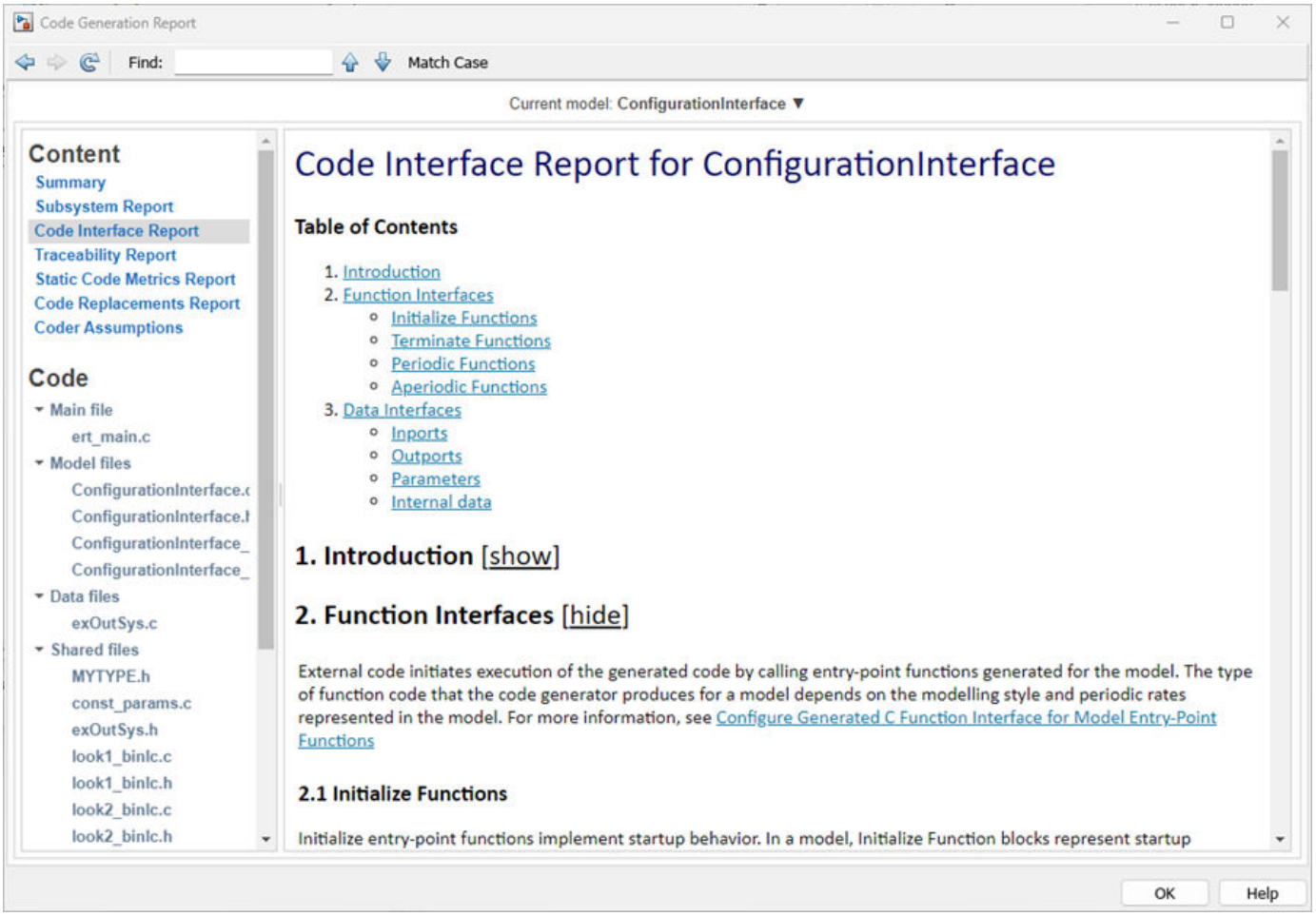
- The code generator supports generating Halide code for the following **Hardware Board** configurations, when target language is C++:
  - `Android Device`
  - `Android Device (64bit)`
  - `Raspberry Pi`
  - `Raspberry Pi (64bit)`
  - `Raspberry Pi - Robot Operating System (ROS)`

For more information, see Speed Up Generated Code Execution with Halide Code.

Code Interface Report improvements for data code interface

In R2024a, the Code Interface Report features improve formatting when you generate code for a model configured to use a data code interface configuration. The features make it easier to assess and navigate the generated code interfaces. The Code Interface Report includes these improvements:

- Table of contents with hyperlinks to report sections and subsections
- Model entry-point functions documented as function interfaces
- Data interface elements, such as root-level inports and outports, summarized under data interfaces
- Formatted documentation that includes function and variable declarations found in the code files with hyperlinks to the respective code file



For more information, see Analyze Generated Data Code Interface Report.

Set data visibility in generated code to protected by using C++ Code Mappings editor

Starting in R2024a, the C++ Code Mappings editor supports configuring model data element categories **Inports**, **Outports**, **Model parameters**, and **Signals, states and internal data** to appear as protected members of the model class when generating C++ class code.

Before R2024a, you could programmatically configure data class members as protected by using the setData function but not interactively with the Code Mappings editor.

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

Add type name prefixes for C++ bus data initialization code

When generating C++ code, if you set the **Code interface packaging** model configuration parameter to `C++ class` and the **Array container type** model configuration parameter to `std::array`, the code generator now includes type name prefixes in bus data initialization code. The prefixes improve bus data initialization code readability. For example, consider this model.

The variable `structInit` has this value.

```
structInit =

  struct with fields:

     a: [0 0 0 0 0]
    a1: [0 0 0 0 0]
    nB: [1×1 struct]
```

The `nB` field contains a struct with one field, `n`, set to `0`.

The model defines two bus types, `topBus` and `nestedBus`, such that `topBus` contains a `nestedBus` element.

Name	Type
nestedBus	
n	double
topBus	
a	double
a1	double
nB	**Bus: nestedBus**

Before R2024a, the code generator generated this bus data initialization code.

```
const topBus model_rtZtopBus{
  { {
      0.0, 0.0, 0.0, 0.0, 0.0 } }
  ,                                     // a

  { {
      0.0, 0.0, 0.0, 0.0, 0.0 } }
  ,                                     // a1

  {
    0.0                                 // n
  }                                     // nB
};                                      // topBus ground
```

Starting in R2024a, the code generator generates this code that includes the type name prefixes `topBus` and `nestedBus`.

```
const topBus model_rtZtopBus{ topBus{
    { {
```

```
        0.0, 0.0, 0.0, 0.0, 0.0 } }
    ,                                   // a

    { {
        0.0, 0.0, 0.0, 0.0, 0.0 } }
    ,                                   // a1
    nestedBus{
      0.0                               // n
    }                                   // nB
  } };                                  // topBus ground
```

List code definition packages loaded into an Embedded Coder Dictionary

List the code definition packages loaded into an Embedded Coder dictionary by using the `getLoadedPackages` method of the `coder.Dictionary` class.

Removed code generation checks

Starting in R2024a, code generation modeling guideline **cgsl_0101: Zero-based indexing** is removed. As a results, check **Identify blocks using one-based indexing** (ID: `mathworks.codegen.cgsl_0101`) is removed from the Model Advisor.

Deployment

ASAP2 file generation for C++ programming language

R2024a adds support for generating an ASAP2 file when generating C++ code from a model. You can set the model configuration parameter **Language** to `C++` and configure the model for ASAP2 file generation by using the Generate Calibration Files tool.

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

Support for ASCII data types in ASAP2 files

R2024a adds support for exporting the model elements configured with `string` data type to ASAP2 files. This feature enables you to:

- Generate an ASAP2 file containing the group of ASCII-based characteristics and measurements.
- Create a custom characteristic or measurement object with ASCII type.

For more information, see Export Characteristics and Measurements Objects in Groups.

Generate C++ concurrent main for XCP-based external mode simulation

For an XCP-based external mode simulation, the code generator produces a simplified `ert_main.cpp` file when you specify these configuration parameter settings:

- `ConcurrentTasks` — `'on'`
- `SystemTargetFile` — ERT-based, for example, `ert.tlc`
- `TargetLang` — `'C++'`
- `TargetLangStandard` — `'C++11 (ISO)'`
- `CodeInterfacePackaging` — `'C++ class'`
- `GenerateSampleERTMain` — `'on'`
- `TargetOS` — `'NativeThreadsExample'`

The generated code is portable because it uses the concurrency and multithreading capabilities of the C++11 (ISO) standard library. In previous releases, the generated code used a threading API that was specific to the operating system. For example, for Linux, the generated `main` used POSIX® threads.

For more information, see:

- External Mode Simulation by Using XCP Communication
- Model Multicore Concurrent Tasking Application
- Generate an example main program
- Deploy Applications to Target Hardware

Embedded Coder Support Package for Linux Applications Enhancements

In R2024a, the Embedded Coder Support Package for Linux Applications has these enhancements:

- `linuxTarget` function — Use the `linuxTarget` function to get an instance of the target object.
- `restartContainer` function — Use the `restartContainer` function to restart or recreate the docker container on a Linux target.

⚠Functionality being removed or changed

lcc-win64 compiler not supported

Errors

The `lcc-win64` compiler is no longer supported. For information about supported compilers, see Supported and Compatible Compilers - Windows.

Performance

★Identify critical paths in generated code

Use the `coder.profile.test.analyzePath` function to run a critical path analysis, which enables you to identify the code execution paths that produce the longest execution time for the generated tasks. Such paths are called critical paths. If you have a set of model inputs, you can use the function to run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation to identify the critical paths and produce execution-time metrics for them. If the model inputs do not have sufficient coverage and you have a Simulink Design Verifier™ license, use the `coder.profile.test.generateTests` function to generate test cases.

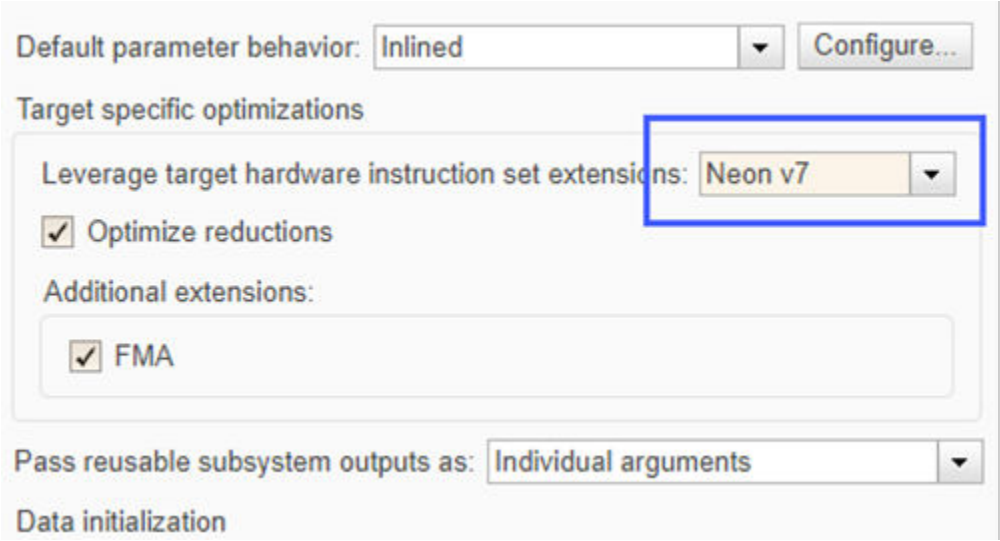
Then, using the Code Profile Analyzer, for each generated task, you can:

- Review the execution flow of the identified critical path.
- Identify the most expensive code sections of the path.
- Highlight the path in the model canvas.

Identifying the critical paths and expensive sections helps you focus your optimization effort on the paths in the expensive sections to achieve your performance requirements. For more information, see Analyze Execution Times of Critical Paths. You can also generate a set of new test cases for the identified paths by using the `coder.profile.test.generateCriticalPathTest` function. Using the test cases, you can automate profiling and identify the most demanding tests. For more information, see Identify Hotspots in Generated Code.

★Generate SIMD code for ARM Cortex-A by using configuration parameters

Generate optimized single instruction, multiple data (SIMD) code for ARM Cortex-A targets by using the model configuration parameter **Leverage target hardware instruction set extensions**. Before R2024a, to generate SIMD instructions for ARM Cortex-A targets, you had to use a GCC ARM Cortex-A code replacement library. For more information, see Generate SIMD Code from Simulink Blocks for ARM Platforms.



★Replace code generated from signal processing blocks

Use custom implementation code that is optimized for your target hardware to replace the default code generated from these blocks:

- Discrete FIR Filter
- Biquad Filter
- FFT
- IFFT
- FIR Decimation
- FIR Interpolation

To replace the generated code for these blocks with custom code, use a code replacement library. For more information, see Block Replacement and Replace Code Generated from Discrete FIR Filter Blocks.

Improved code efficiency with MATLAB System blocks and System objects for code generation targets

Starting in R2024a, when using MATLAB System blocks within a model or a block implemented by using a System object, the generated code is optimized for efficiency for rapid accelerator simulation and code generation targets, excluding Simulink Design Verifier(SLDV). This enhancement aims to minimize data copies, resulting in reduced memory usage.

Consider the model `mSFcnReleaseImplTest` with a MATLAB System block.

The code generated in R2023b contains a data copy where the values from `mSFcnReleaseImplTest_P.Constant_Value` are copied to the unnecessary variable `varargin_1`.

Generated code in 2023b

```
/* Model step function */
void mSFcnReleaseImplTest_step(void)
{
  creal_T Y0[15];
  real_T U0[15];
  real_T varargin_1[15];
  char_T *sErr;

  /* MATLABSystem: '<Root>/MLSys' incorporates:
   *  Constant: '<Root>/Constant'
   */
  memcpy(&varargin_1[0], &mSFcnReleaseImplTest_P.Constant_Value[0], 15U * sizeof
         (real_T));
  if (mSFcnReleaseImplTest_DW.obj.cAudioFileWriter.S0_isInitialized != 1)
  {
    mSFcnReleaseImplTest_DW.obj.cAudioFileWriter.S0_isInitialized = 1
```

Code generated in R2024a uses the values from `mSFcnReleaseImplTest_P.Constant_Value` directly, without creating a copy of the variable, resulting in reduced memory usage and improved performance.

Generated code in 2024a

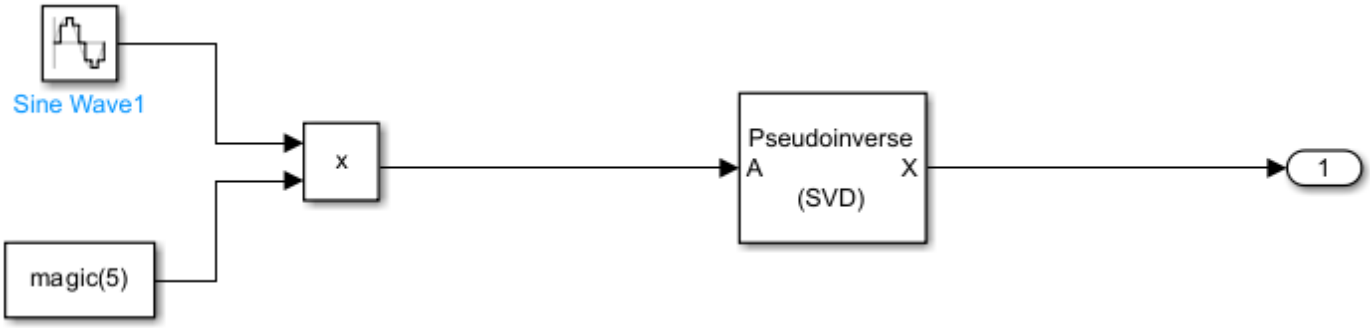
```
/* Model step function */
void mSFcnReleaseImplTest_step(void)
{
  creal_T Y0[15];
  real_T U0[15];
  char_T *sErr;

  /* MATLABSystem: '<Root>/MLSys' incorporates:
   *  Constant: '<Root>/Constant'
   * */
  if (mSFcnReleaseImplTest_DW.obj.cAudioFileWriter.S0_isInitialized != 1)
  {
    mSFcnReleaseImplTest_DW.obj.cAudioFileWriter.S0_isInitialized = 1;
```

For more information, see Implement Blocks with System Objects, Optimize Performance and Data Copy Reduction.

Improved parameter handling for MATLAB System blocks

Starting in R2024a, for models that contain MATLAB System block or blocks implemented by using a System object, the code generator optimizes the generated code by passing parameters to functions by reference instead of by value. This enhancement reduces the need for creating extra data copies, offering major improvements in memory usage.

For example, consider the model `mSFcnRefPassTest` that includes a Pseudoinverse block.



The code generated in R2023b creates an array `b_y` to store updated values. It duplicates data by copying values from the array `y` to the array `b_y`.

Generated code in 2023b

```
static void mSFcnRefPassTest_xaxpy(int32_T n, real_T a, int32_T ix0, const real_T
  y[25], int32_T iy0, real_T b_y[25])
{
  int32_T b_y_tmp;
  int32_T ix;
  int32_T iy;
  int32_T k;

  /* Start for MATLABSystem: '<Root>/Pseudoinverse' */
  memcpy(&b_y[0], &y[0], 25U * sizeof(real_T));
  if (!(a == 0.0)) {
    ix = ix0 - 1;
    iy = iy0 - 1;
    for (k = 0; k < n; k++) {
      b_y_tmp = iy + k;
      b_y[b_y_tmp] += b_y[ix + k] * a;
    }
  }
```

Code generated in R2024a eliminates the need for an additional array and directly updates values in the array `y` by passing `y` by reference, thereby reducing data duplication.

Generated code in 2024a

```
static void mSFcnRefPassTest_xaxpy(int32_T n, real_T a, int32_T ix0, real_T y[25],
  int32_T iy0)
{
  int32_T ix;
  int32_T iy;
  int32_T k;
  int32_T tmp;

  /* Start for MATLABSystem: '<Root>/Pseudoinverse' */
  if (!(a == 0.0)) {
    ix = ix0 - 1;
    iy = iy0 - 1;
    for (k = 0; k < n; k++) {
      tmp = iy + k;
      y[tmp] += y[ix + k] * a;
    }
  }
```

For more information, see Implement Blocks with System Objects and Data Copy Reduction.

Generate SIMD code for MinMax blocks with support for non-finite numbers

Starting in R2024a, you can generate SIMD code for MinMax blocks when **Support: non-finite numbers** is set to `on`. Before R2024a, you could not generate SIMD code for MinMax blocks when non-finite support was enabled. For more information, see Generate SIMD Code from Simulink Blocks for Intel Platforms.

Avoid data loss in code replacement arguments

In R2024a, you can avoid data loss in code replacement arguments by fixing entries that the new warning flags during the code replacement entry validation process. The warning flags code replacement entries that can cause data loss due to conversion from a conceptual argument data type to an implementation argument data type. For more information, see Interactively Develop a Code Replacement Library.

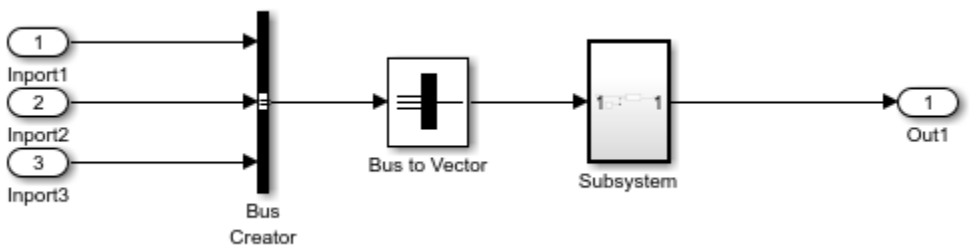
Add code replacement miss reasons for custom entries

Make custom code replacement entries easier to troubleshoot by adding code replacement miss reasons in error messages in the `do_match` method of the entry. If a code replacement miss occurs for the entry due to one of the errors, the **Trace Information** table in the Code Replacement Viewer indicates the miss reason and includes the message from the error that caused `do_match` to fail. For more information, see Customize Match and Replacement Process.

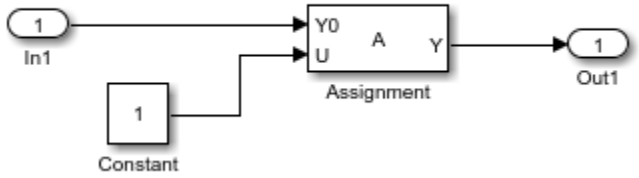
Optimized code reusing buffers for reusable subsystems, MATLAB Function blocks, and Charts

Before R2024a, the generated code contained unnecessary data copies for copying the input data of reusable subsystems, MATLAB Function blocks, and Stateflow charts to output variables. Starting in R2024a, the generated code reuses buffers for the input and output signals and eliminates the unnecessary data copies. Reusing buffers reduces RAM consumption and improves code execution speed. However, the code generator does not use this optimization for conditional subsystems and subsystems that use the `Reusable` storage class.

For example, consider the `mReuseSubsystemIO` model.



The model contains a reusable subsystem that assigns a constant value of `1` to the first element of the Assignment block input signal.



Previously, the code generator produced this code for the subsystem.

```
void mReuseSubsystemIO_Subsystem(const real_T rtu_In1[3], real_T rty_Out1[3])
{
   rty_Out1[0] = rtu_In1[0];
   rty_Out1[1] = rtu_In1[1];
   rty_Out1[2] = rtu_In1[2];
   rty_Out1[0] = 1.0;
}
```

The generated code unnecessarily copied the input data to the output variable `rty_Out1`, then assigned the constant value of `1` to the first element of `rty_Out1`.

In R2024a, the code generator produces this code for the subsystem.

```
void mReuseSubsystemIO_Subsystem(real_T rty_Out1[3])
{
   rty_Out1[0] = 1.0;
}
```

The code does not generate a separate variable for the input data. Instead, the code reuses the output variable `rty_Out1` to hold the input data and then directly assigns the constant value of `1` to the first element of `rty_Out1`. For more information, see Specify Buffer Reuse for Signals in a Path.

Optimized code that reuses Matrix Concatenate block buffers

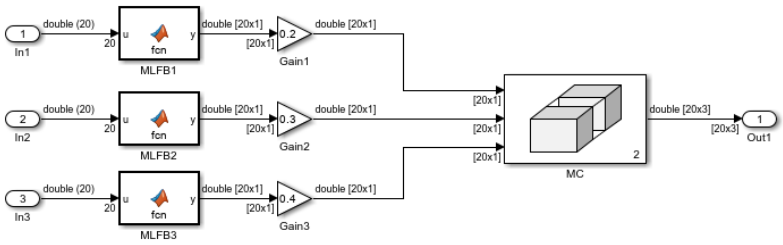
Starting in R2024a, the generated code contains fewer data copies for models containing a Matrix Concatenate block when all of these conditions are true:

- The input signals of the Matrix Concatenate block are vectors or matrices.
- The blocks whose output signals directly feed into the Matrix Concatenate block each have only one input.
- The input blocks of the Matrix Concatenate block do not receive input signals from these blocks:
  - Root Inport blocks
  - Virtual blocks
  - Selector blocks
- If the Matrix Concatenate block parameter **Mode** is set to `Multidimensional array`:
  - For 1-D input signals, **Concatenate dimension** is set to `1` or `2`.
  - For 2-D input signals, **Concatenate dimension** is set to `2`.
  - For input signals of `n`-D where `n` is greater than `2`, **Concatenate dimension** is set to `n`.

  - The model configuration parameter **Array layout** is set to `Column-major`.
- The model does not contain a chain of Matrix Concatenate blocks.

If a model meets all of the conditions, the generated code reuses portions of the Matrix Concatenate block output buffer for its input signals when they pass from other blocks. Reusing the portions of the output buffer reduces RAM consumption and data copies. To enable this optimization, select the **Reuse buffers of different sizes and dimensions** parameter.

Consider the model `subRegionReuse`, which satisfies the modeling conditions.



The signals `In1`, `In2`, and `In3` pass through the MATLAB Function blocks and Gain blocks and feed into the Matrix Concatenate block `MC`.

Previously, the code generator produced this code.

```
void subRegionReuse_step(void)
{
  real_T rtb_y[20];
  real_T rtb_y_g[20];
  real_T rtb_y_g0[20];
  int32_T i;
  subRegionReuse_MLFB1(subRegionReuse_U.In1, rtb_y_g);
  subRegionReuse_MLFB1(subRegionReuse_U.In2, rtb_y_g0);
  subRegionReuse_MLFB1(subRegionReuse_U.In3, rtb_y);
  for (i = 0; i < 20; i++) {
    subRegionReuse_Y.Out1[i] = 0.2 * rtb_y_g[i];
    subRegionReuse_Y.Out1[i + 20] = 0.3 * rtb_y_g0[i];
    subRegionReuse_Y.Out1[i + 40] = 0.4 * rtb_y[i];
  }
}
```

The generated code contained the unnecessary variables `rtb_y`, `rtb_y_g`, and `rtb_y_g0` for the signals passing from the MATLAB Function blocks `MLFB1`, `MLFB2`, and `MLFB3`.

In R2024a, the code generator produces this code.

```
void subRegionReuse_step(void)
{
  int32_T i;
  subRegionReuse_MLFB1(subRegionReuse_U.In1, &subRegionReuse_Y.Out1[0]);
  subRegionReuse_MLFB1(subRegionReuse_U.In2, &subRegionReuse_Y.Out1[20]);
  subRegionReuse_MLFB1(subRegionReuse_U.In3, &subRegionReuse_Y.Out1[40]);
  for (i = 0; i < 20; i++) {
    subRegionReuse_Y.Out1[i] *= 0.2;
    subRegionReuse_Y.Out1[i + 20] *= 0.3;
    subRegionReuse_Y.Out1[i + 40] *= 0.4;
```

```
  }
}
```

The generated code does not contain the unnecessary variables `rtb_y`, `rtb_y_g`, and `rtb_y_g0`. Instead, the code reuses the portions of the output buffer `Out1` for the signals passing from the MATLAB Function blocks.

Identify performance hotspots in generated code

Use the `coder.profile.test.runTests` function to automate test execution, which enables you to identify performance hotspots in generated code. If you have a set of typical model inputs or test cases, you can use the function to run software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations that test the generated code. Then, using the Code Profile Analyzer, you can:

- For each generated task, identify the longest execution times and the associated test cases.
- For the most demanding task executions, analyze the function-call stack to identify performance bottlenecks.

For more information, see Identify Hotspots in Generated Code.

Configure code profiling in model hierarchy using Code Profile Analyzer

The Code Profile Analyzer provides a **Configure** tab, which enables you to specify code profiling settings of different models. You can:

- Easily configure code profiling settings across a model hierarchy.
- Detect and resolve configuration issues before generating code.

For more information, see Configure Profiling in Model Hierarchy.

Generate OpenMP compatible C code with matchFeatures

Starting in R2024a, the function `matchFeatures` (Computer Vision Toolbox) generates Open Multiprocessing (OpenMP) compatible portable C code for nonhost target hardware when using the `Exhaustive` method, increasing the execution speed of the generated code.

Detect task overrun events during XCP external mode simulations

For each generated task, an XCP-based external mode simulation produces a task diagnostic signal that represents the number of overrun events. During the simulation, you can stream the signals to the Simulation Data Inspector. At the end of the simulation, you can also use the Code Profile Analyzer to view the overrun rate.

Use the diagnostic signals and overrun rates to check that time budgets are met for task execution on the target hardware. You can identify scheduling issues early if the target hardware is overloaded.

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

⚠ Functionality being removed or changed

timeline function will be removed

Warns

The `timeline` function will be removed in a future release. You can use the `schedule` function to visualize task scheduling using the Simulation Data Inspector.

Hardware Support

★Embedded Coder Support Package for Infineon AURIX TC3x Processors: Generate, build, and deploy Simulink models on Infineon AURIX 2nd generation (TC3xx series) of processors

The Embedded Coder Support Package for Infineon AURIX TC3x Microcontrollers is available from release R2024a onwards. You can use the support package to design real-time applications for Infineon 32-bit TriCore AURIX TC3x family of processors using Simulink and generate processor optimized code that you can compile and execute on AURIX TC3x family of processors.

The first release of this support package includes:

- Support for Digital Port Read, Digital Port Write, Hardware Interrupt, EVADC, PWM, QSPI Controller, QSPI Peripheral blocks.
- Support for Hardware Mapping and Peripheral Configurations.

For more information, see Infineon AURIX TC3x

★STMicroelectronics STM32 Processors: Support for STM32F3xx and STM32H7xx (Dual-core) based boards

The Embedded Coder Support Package for STMicroelectronics STM32 Processors now supports the new STM32F3xx-based boards and STM32H7xx-dual core series for M7 and M4. Starting in R2024a, you can:

- Generate and build code using an STM32CubeMX project file.
- Use the Analog to Digital Converter, PWM Output, Digital Port Read, Hardware Interrupt, Timer, Encoder, and UART/USART Read blocks to implement Model-Based Design workflows.
- Monitor signals and tune parameters in the external mode. For more information, see Monitoring and Tuning Using STMicroelectronics STM32 Processor Based Boards
- Run processor-in-the-loop (PIL) simulation in the serial communication mode.

★ARM Cortex-M Processors: Code generation and verification support for ARM Cortex-M4, ARM Cortex-M7, and ARM Cortex-M55 boards

Starting in R2024a, you can use the ARM Cortex M4 (MPS2), ARM Cortex M7 (MPS2), and ARM Cortex M55 (MPS3) hardware boards with Simulink and MATLAB for code generation and verification using QEMU over PIL. It also supports code replacement library (CRL) using Embedded Coder Support Package for ARM Cortex-M Processors.

STMicroelectronics STM32 Processors: Publish and subscribe to messages using MQTT blocks

Starting in R2024a, you can use the MQTT Publish and MQTT Subscribe blocks to publish and subscribe to messages from STM32F7xx and STM32H7xx processor-based boards. Message queuing telemetry transport (MQTT) is a publish-subscribe architecture that connects to bandwidth- and

power-constrained devices over wireless networks. For more information, see Process ECG Signals Using MQTT on STM32 Processor Boards.

STMicroelectronics STM32 Processors: Enhanced PDM filtering with multichannel and variable decimation using STM32CubeMX

The updated I2S Mic In block now supports PDM filtering using STM32CubeMX. The PDM filter provides support for multiple channels and decimation factors on STM32F4xx and STM32F7xx processor-based boards. For more information, see Capture PDM Stereo Audio on STM32 Processor.

STMicroelectronics STM32 Processors: Idle Task block support using STM32CubeMX

Starting in R2024a, you can use the Idle Task block to create a free-running task that executes the downstream subsystem using STM32CubeMX.

STMicroelectronics STM32 Processors: Enhancements to CORDIC co-processor, Comparator, FDCAN, and Digital to Analog Converter blocks

Starting R2024a, you can use these blocks with STM32 processor-based boards.

- Use the updated CORDIC co-processor block, which now supports DMA mode in STM32H7xx and STM32G4xx processor-based boards
- Use the updated Comparator block with STM32U5xx processor-based board to compare two analog inputs to the peripheral and view the comparison result as a logical value at the block output port.
- Use the updated FDCAN Read and FDCAN Write blocks with STM32U5xx processor-based board to read and write data from the CAN FD bus.
- Use the updated Digital to Analog Converter (DAC) block with STM32U5xx processor-based board to convert a digital value to its equivalent analog voltage on the specified channel.

STMicroelectronics STM32 Processors: New examples that demonstrate the capabilities of STM32 Processors

Use the following new examples to explore capabilities of the Embedded Coder Support Package for STMicroelectronics STM32 Processors.

- Capture PDM Stereo Audio on STM32 Processor - This example shows how to use the I2S Mic In block in a Simulink model to acquire pulse density modulation (PDM) stereo audio data, convert it to pulse code modulation (PCM) format and visualize the audio signal using Embedded Coder Support Package for STMicroelectronics STM32 Processors.
- Process ECG Signals Using MQTT on STM32 Processor Boards - This example shows how to use the STM32 Nucleo F767ZI board using Embedded Coder Support Package for STMicroelectronics STM32 Processors to process an ECG signal input from an ECG sensor, extract the heart rate in beats per minutes (bpm), and send the ECG signal and heart rate to the ThingSpeak internet of things (IoT) analytics platform service.
- Getting Started with STM32H7xx-Based Dual-Core Boards - This example shows how to use the Embedded Coder Support Package for STMicroelectronics STM32 Processors to run a Simulink model on a STM32 H7xx-based dual-core board.

- Using I2C to Read and Write Data to Accelerometer on STM32 Processor Board - This example shows how to configure and use I2C protocol blocks to read and write accelerometer data using Embedded Coder Support Package for STMicroelectronics STM32 Processors.
- Generate Motor Control Models for Selected Algorithm and Hardware - This example shows how to use Motor Control Blockset to generate a Simulink model that is configured for a specific hardware and motor control technique.
- **Using CORDIC Co-Processor Block with DMA in STMicroelectronics STM32 Processor Based Boards** - This example shows how to use and generate code for a CORDIC block in a Simulink model for STMicroelectronics NUCLEO-G431RB board to calculate SINE and COSINE values for given values of theta by using DMA.

STMicroelectronics STM32 Processors: Use SPI blocks for data exchange

Use the SPI Transmit, SPI Receive, and SPI Controller Transfer blocks to write data to and read data from an SPI peripheral device in STM32F4xx, STM32L4xx, STM32L5xx, STM32WBxx, and STM32U5xx processor-based boards.

BeagleBone Black Hardware: Support package documentation moved into Embedded Coder documentation

Starting in R2024a, the Embedded Coder Support Package for BeagleBone® Black Hardware documentation is included within the Embedded Coder documentation. In previous releases, the support package documentation installs with the support package software. To access archived release notes from the previous release, see R2023b Beaglebone Black Hardware Release Notes.

BeagleBone Black Hardware: Transition to native build compiler toolchain

Starting in R2024a, Embedded Coder Support Package for BeagleBone Black Hardware is transitioned to the native build compiler toolchain. The support package utilizes the compiler that is available on the BeagleBone Black hardware. This change makes the support package compatible with the latest version of Debian Linux operating system running on the hardware. The host-based Linaro cross-compiler toolchain, used in earlier releases, imposed a limitation on the supported Linux version.

ARM Cortex-A Processors: Updated code replacement library selection for ARM Cortex-A Examples

The code replacement library (CRL) selection in examples for DSP System Toolbox Support for ARM Cortex-A Processors has been updated to GCC ARM Cortex-A.

Infineon AURIX TC4x Microcontrollers: Support package documentation moved into Embedded Coder documentation

Starting in R2024a, the Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers documentation is included within the Embedded Coder documentation. In previous releases, the

support package documentation installs with the support package software. To access archived release notes from the previous release, see R2023b Infineon Aurix TC4X Release Notes.

Infineon AURIX TC4x Microcontrollers: Support for Infineon low-level driver (iLLD) 2.0.1.2.19

Starting in R2024a, you can use the Infineon low-level driver (iLLD) 2.0.1.2.19 with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Resolver block to measure resolver sensor angles

Use the new Resolver block from Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to measure the position of the secondary or rotor coil of the resolver sensor used in motor control applications. You can also use the block to generate the carrier waveform, which excites the primary coil of the resolver sensor. After adding the block to your model, you can configure the parameters related to carrier wave, input, timestamp, filter, boundary and events by using Resolver Peripheral Configuration tool.

Infineon AURIX TC4x Microcontrollers: Fast Compare Comparator (FCC) block to detect analog signal crossings

Use the new FCC block from Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers for fast detection of analog input signal crossings against a 10-bit programmable threshold value. To program the threshold value, enable the **Threshold Update** parameter in the FCC block and configure corresponding parameters in FCC Peripheral Configuration tool. You can use this block to detect over-current or voltage conditions that trigger an immediate PWM switch-off in DC-DC converter applications.

Infineon AURIX TC4x Microcontrollers: Converter Digital Signal Processing (CDSP) block to process data output from the ADC peripherals of IFX TC4x

Use the new CDSP block from Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers to apply signal processing techniques on the data arriving from DSADC block. After adding the block to your model, you can configure the parameters related to timestamp, boundary and events by using CDSP Peripheral Configuration tool.

Infineon AURIX TC4x Microcontrollers: Enhancements to SENT

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers now includes these enhancements for configuring SENT block:

- You can configure the serial output port. On the **Basic** tab, select the **Serial** parameter.
- You can configure the SENT Protocol. On the **Advanced** tab, set the **SENT protocol** parameter to `Standard` or `Short PWM Code (SPC)`.
- If you set the **SENT protocol** parameter to `Short PWM Code (SPC)`, you can set the **SPC mode** parameter to `Synchronization`, `Feature selection`, or `Sensor selection`. For each **SPC mode**, you can configure the corresponding parameters by using SENT Configuration tool.

.

Infineon AURIX TC4x Microcontrollers: Enhancements to PWM

The Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers includes these enhancements for configuring PWM block:

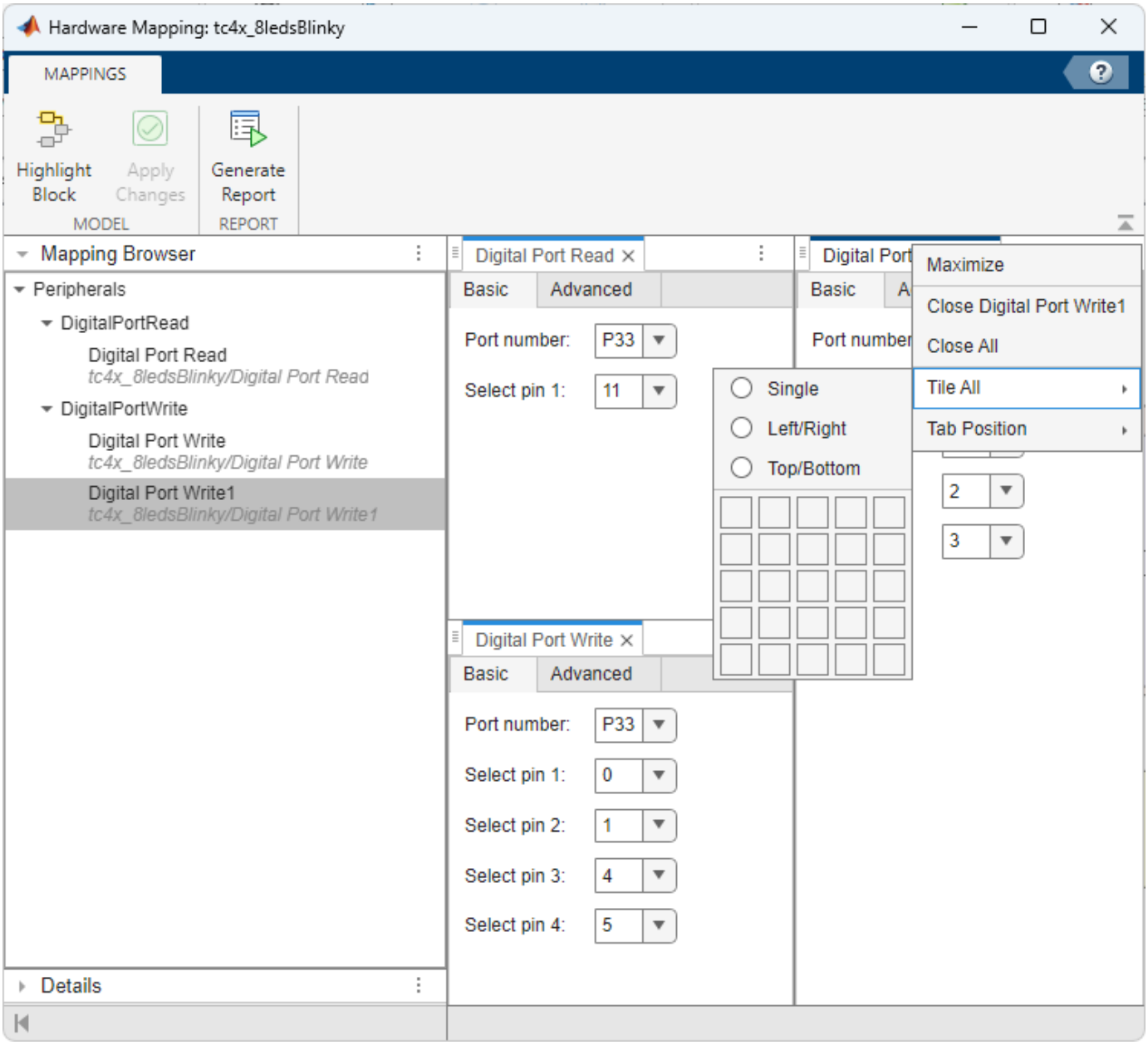
- You can configure the fast shut off trigger functionality. On the **Output #** tab of Hardware Mapping window, select **Enable fast shut off trigger** parameter.
- You can configure the fast shut off trigger parameters for Dead Time Module (DTM) functionality in **DTM** tab of PWM Peripheral Configuration tool.
- You can trigger SENT block from the PWM module. On the **Output #** tab of the PWM Peripheral Configuration tool, select **Sent trigger signal** parameter.

.

Infineon AURIX TC4x Microcontrollers: New layout options in Hardware Mapping tool

The Hardware Mapping tool in Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers includes these enhancements:

- You can view the peripheral block name and its path from the model in the **Mapping Browser** pane in the Hardware Mapping tool.
- You can select blocks in the **Mapping Browser** pane to create tabs for the blocks in the right pane. You can compare the parameters of different blocks by splitting the tabs vertically or horizontally through drag and drop, or by right-clicking and selecting from the tile options.



ARM Cortex-A Processors: CMSIS CRL Support for Basic Math Functions

The Embedded Coder Support Package for ARM Cortex-A Processors now includes a Code Replacement Library (CRL) `ARM Cortex-A CMSIS` to generate calls to CMSIS-DSP library optimized for ARM Cortex-A processors. Refer to Supported CMSIS Library Functions for ARM Cortex -A Processors to see the list of MATLAB functions that can be replaced with optimized code.

The Generate Optimized Code for Math Functions Using ARM Cortex-A CMSIS CRL shows how to use the ARM Cortex-A CMSIS code replacement library (CRL) to generate optimized code for math functions on ARM Cortex-A hardware targets.

ARM Cortex-M Processors: CMSIS CRL Support for Additional Math Functions

Starting in R2024a, you can use the Embedded Coder Support Package for ARM Cortex-M Processors to generate code for these Simulink blocks and MATLAB functions that calls the CMSIS-DSP library using ARM Cortex-M CRL.

- Scale/Gain
- Bias/Offset
- Dot Product
- Saturate/Limit
- Unary Minus

ARM Cortex-M Processors: Multichannel support for DSP Blocks and System object Code Replacement

The ARM Cortex-M CMSIS CRL now supports multichannel in these DSP blocks and System objects:

- Biquad Filter
- FFT
- IFFT
- Discrete FIR Filter
- Decimator
- Interpolator
- `dsp.SOSFilter`
- `dsp.FFT`
- `dsp.IFFT`
- `dsp.FIRFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`

ARM Cortex-M Processors: Code Generation for Sound Classification on ARM Cortex-M Targets with CMSIS-NN

The Code Generation for Sound Classification on ARM Cortex-M Targets using CMSIS-NN example shows how to generate code for the Classify Sound Using Deep Learning (Audio Toolbox) using a pretrained network and the CMSIS-NN library. In this example, you classify white noise, brown noise and pink noise by generating a PIL MEX function, which allows you to execute the generated code on the target hardware.

Check bug reports for issues and fixes

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

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

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

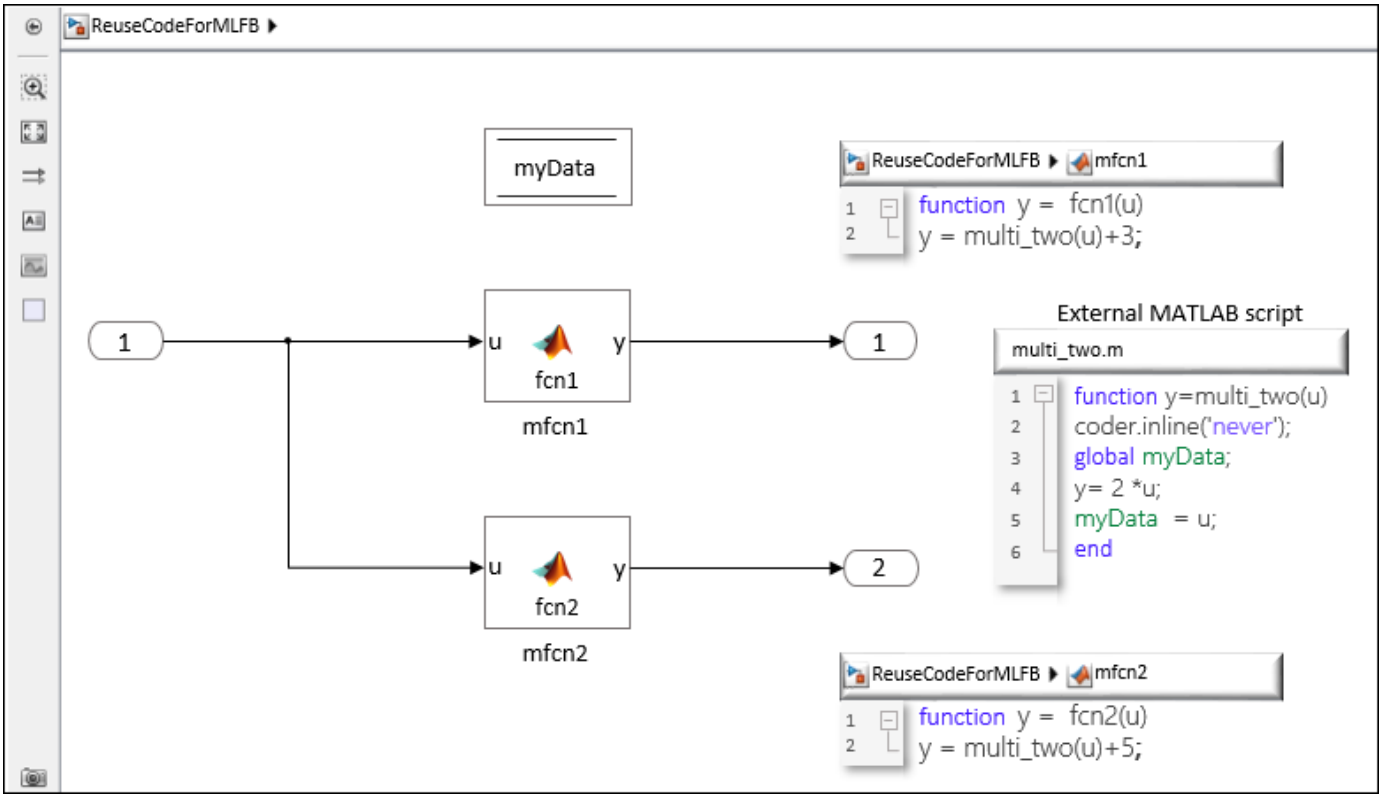
R2023b

Code Generation from MATLAB Code

Eliminate duplicate code by reusing single static functions in MATLAB Function blocks

Starting in R2023b, the code generator eliminates redundant code by reusing identical static functions generated for each MATLAB Function block in a model. Eliminating redundant code reduces code complexity, reduces memory usage at run time, and improves code readability. Previously, the code generator produced separate functions, resulting in duplication, despite having identical implementations and sharing the same external MATLAB file.

Consider the model `ReuseCodeForMLFB`, which includes two MATLAB Function blocks `mfcn1` and `mfcn2` that call the function `multi_two`. Set the model configuration parameter **Shared code placement** to either `Auto` or `Shared Location`. The associated global data store stores the input of the model.



The table compares the code generated in R2023a and R2023b for the model `ReuseCodeForMLFB`. In R2023a, the code contains duplicate functions `ReuseCodeForMLFB_multi_two` and `ReuseCodeForMLFB_multi_two_f` to represent the calls to `multi_two` from `mfcn1` and `mfcn2`. In R2023b, the code reuses the function `ReuseCodeForMLFB_multi_two`, thus removing code redundancy.

Generated Code in R2023a

```
/* Real-time model */
static RT_MODEL_ReuseCodeForMLFB_T ReuseCodeForMLFB_M_;
RT_MODEL_ReuseCodeForMLFB_T *const ReuseCodeForMLFB_M = &ReuseCodeForMLFB_M_;
static void fcn1(void);
static void fcn2(void);

/* Forward declaration for local functions */
static real_T ReuseCodeForMLFB_multi_two(real_T u);

/* Forward declaration for local functions */
static real_T ReuseCodeForMLFB_multi_two_f(real_T u);

/* Function for MATLAB Function: '<Root>/mfcn1' */
static real_T ReuseCodeForMLFB_multi_two(real_T u)
{
  real_T y;
  y = 2.0 * u;
  myData = u;
  return y;
}

/* Output and update for atomic system: '<Root>/mfcn1' */
static void fcn1(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  ReuseCodeForMLFB_Y.Out1 = ReuseCodeForMLFB_multi_two(ReuseCodeForMLFB_U.In1) +
    3.0;
}

/* Function for MATLAB Function: '<Root>/mfcn2' */
static real_T ReuseCodeForMLFB_multi_two_f(real_T u)
{
  real_T y;
  y = 2.0 * u;
  myData = u;
  return y;
}

/* Output and update for atomic system: '<Root>/mfcn2' */
static void fcn2(void)
{
  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In1'
   */
  ReuseCodeForMLFB_Y.Out2 = ReuseCodeForMLFB_multi_two_f(ReuseCodeForMLFB_U.In1)
    + 5.0;
}

/* Model step function */
void ReuseCodeForMLFB_step(void)
{
  /* MATLAB Function: '<Root>/mfcn1' */
  fcn1();

  /* MATLAB Function: '<Root>/mfcn2' */
  fcn2();
}
```

Generated Code in R2023b

```
/* Real-time model */
static RT_MODEL_ReuseCodeForMLFB_T ReuseCodeForMLFB_M_;
RT_MODEL_ReuseCodeForMLFB_T *const ReuseCodeForMLFB_M = &ReuseCodeForMLFB_M_;

/* Exported functions */
extern real_T ReuseCodeForMLFB_multi_two(real_T u);
static void fcn1(void);
static void fcn2(void);

/* Function for MATLAB Function: '<Root>/mfcn1' */
real_T ReuseCodeForMLFB_multi_two(real_T u)
{
  real_T y;
  y = 2.0 * u;
  myData = u;
  return y;
}

/* Output and update for atomic system: '<Root>/mfcn1' */
static void fcn1(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   */
  ReuseCodeForMLFB_Y.Out1 = ReuseCodeForMLFB_multi_two(ReuseCodeForMLFB_U.
}

/* Output and update for atomic system: '<Root>/mfcn2' */
static void fcn2(void)
{
  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In1'
   */
  ReuseCodeForMLFB_Y.Out2 = ReuseCodeForMLFB_multi_two(ReuseCodeForMLFB_U.
}

/* Model step function */
void ReuseCodeForMLFB_step(void)
{
  /* MATLAB Function: '<Root>/mfcn1' */
  fcn1();

  /* MATLAB Function: '<Root>/mfcn2' */
  fcn2();
}
```

Stack size includes input arguments and return value in static code metrics report

Starting in R2023b, the static code metrics report includes the sizes of function input arguments and the return value, in addition to the nonstatic local variables, to measure the function stack size. You can see the increase in the estimated stack size in the **Accumulated Stack Size (bytes)** and **Self Stack Size (bytes)** columns of the static code metrics report.

Prior to R2023b, only the nonstatic local variables were considered when measuring the function stack size. The static code metrics assumed that the compiler optimized the function arguments to

register memory. So, the sizes of these arguments were not included when calculating the function stack size.

For more information, see Generating a Static Code Metrics Report for Code Generated from MATLAB Code.

Model Architecture and Design

Code generation support for concurrent execution of message blocks operating at different rates within a model

Starting in R2023b, Embedded Coder supports code generation for concurrent execution of message blocks that run at different rates within a model.

To model this in Simulink, you must configure the model by setting the configuration parameter **Allow tasks to execute concurrently on target** under **Tasking and sample time options** in the **Solver** pane. Simulink allows each rate in the model to execute as an independent concurrent task on the target processor.

The generated code uses a mutex to ensure thread safety. You can replace the default mutex implementations with user-authored functions by using the code replacement library. For more information about code replacement, see Semaphore and Mutex Function Replacement.

Branching and merging root-level ports in export-function models

When concurrent data access is a concern, branching root-level input ports and merging root-level output ports are limited to specific scenarios. Starting in R2023b, you can safely branch and merge root-level ports in more scenarios than previously. The scenarios, both before and now, apply only to export-function models.

Before R2023b, you could safely perform these actions:

- Branch root-level input ports configured to use the outside-execution data communication method.
- Merge nonscalar root-level output ports configured to use the outside-execution data communication method.

Starting in R2023b, you can also safely perform these actions:

- Branch root-level input ports configured to use the during-execution data communication method.
- Merge scalar root-level output ports configured to use the outside-execution data communication method.
- Merge root-level output ports configured to use the during-execution data communication method.

Note When concurrent data access is a concern:

- Do not design the model to be rate-based.
- Do not branch or merge root-level ports configured to use the direct-access data communication method.
- Do not merge virtual root-level output ports.

For more information see:

- Data Communication Methods
- Create a Service Interface Configuration

- Export-Function Models Overview

MISRA check to identify variant blocks that do not have a default choice

This table lists the new check that has been introduced under MISRA C™:2012 Coding Standards checks. It checks for variant blocks with startup variant activation time that do not have a default choice when the **Casting modes** is set to `Standards Compliant`. Refer Casting modes for further information.

Model Advisor Check	Check ID
Check for variant blocks that do not have a default choice (Simulink Check)	`mathworks.misra.DefaultChoiceVariantChecks`

Update guidelines about the creation of data copies for component deployment

Starting in R2023b, the information in these code generation modeling guidelines is modified or removed.

Modeling Guideline	Description
cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks	Removed information about the creation of data copies.
cgsl_0402: Signal interfaces for component deployment	Added information about the creation of data copies when the signal type is In Bus Element or Out Bus Element.

Code Interface Configuration and Integration

★Define file packaging for generated entry-point functions

Starting in R2023b, you can define custom filenames for entry-point functions generated from a model. Use custom filenames to group related functions together in generated code.

In the Embedded Coder Dictionary, under **Function Customization Templates**, use the **Header File** and **Definition File** properties to define filenames for a function customization template. Use the Code Mappings Editor to map functions to the template.

For more information, see Control File Packaging of Generated Entry-Point Functions and Simulink Functions and Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture.

Define service interface configurations programmatically by using new programming interface

Create and configure a service interface configuration programmatically by using the new programming interface for service interfaces and the existing Embedded Coder Dictionary programming interface, which now supports service interface configurations. The programmatic interface enables you to define service interfaces in an Embedded Coder Dictionary by using a script.

These classes and functions include new behavior that supports service interface definitions.

Class	New behavior in R2023b

`coder.Dictionary`	• Create a `coder.Dictionary` object for a dictionary that contains a service interface configuration.
	• Configure properties of the service interface configuration such as the header filename used for services.
	• Select default interface definitions for service interface categories.
	• Use these new methods:
	  • `getCodeInterfaceType`
	  • `getDictionaryDefault`
	  • `setDictionaryDefault`
	  • `get`
	  • `set`

`coder.dictionary.Section`	• Access `coder.dictionary.Section` objects for the service interface categories. • Add entries for service interface definitions.

Class	New behavior in R2023b
`coder.dictionary.Entry`	• Create `coder.dictionary.Entry` objects for service interface definitions. • Use the new method `isDictionaryDefault`.

Function	New behavior in R2023b
`coder.dictionary.copy`	Copy definitions from one service interface configuration to another service interface configuration.
`coder.dictionary.move`	Move definitions between two service interface configurations.
`coder.dictionary.create`	Create a service interface configuration.

For more information, see Create Service Interface Configuration Programmatically.

Timer service interface support for custom blocks that use time values

Before R2023b, if a model included an inlined S-Function block for a C MEX S-function that relied on time values and the model was configured to use a service code interface, the code generator returned an error.

Starting in R2023b, you can implement inlined S-Function blocks for C MEX S-functions that access 32-bit time values and use these blocks in models configured to use timer service interfaces. You inline an S-Function block by associating the block with a target file. The target file calls Target Language Library (TLC) library functions, which set up communication with Simulink and provide the code generator with specifications for generating code for the block. To generate timer service interface code, the code generator requires the target file to use the enhanced TLC interface, which enables block function optimizations.

For more information, see:

- “Improved code efficiency and better integration of inlined S-Functions”
- S-Functions That Support Timer Service Interfaces for Accessing Time Values
- Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture
- Configure Timer Service Interfaces

Bitfield storage class support for fixed-point and integer data types

Previously, you could use the `Bitfield` storage class to generate a structure that stores Boolean data in named bit fields. In R2023b, you can reduce memory usage by using the `Bitfield` storage class to store fixed-point and integer data in named bit fields. The size of the fixed-point or integer type must not be greater than the size of a native integer type.

For more information, see Choose Storage Class for Controlling Data Representation in Generated Code.

Code Generation

Open example models from the documentation or command line

Use `openExample` to open example models from the command line. For example, to open the `CodeBuildModel` model, enter:

```
openExample('CodeBuildModel')
```

You can also find and open models by using the function `modelfinder` and by using the documentation, including the examples in this table. This table shows example models that have been renamed in R2023b.

R2023a model name	New model name	Example
`rtwdemo_atomic`	`SubsystemAtomic`	Generate Subsystem Code as Separate Function and Files
`rtwdemo_blockreduction`	`BlockReductionOptimization`	Remove Code for Blocks That Have No Effect on Computational Results
`rtwdemo_codebuild`	`CodeBuildModel`	Compile Code in Another Development Environment
`rtwdemo_codebuild_ref`	`CodeBuildRefModel`	
`rtwdemo_comments`	`CustomCodeComments`	Customize Code Comments to Enhance Readability and Traceability
`rtwdemo_comp`	`ClientServerCommunication`	Generate Reentrant Code from Simulink Function Blocks
`rtwdemo_comp_cpp`	`CppComponent`	Generate Reentrant Code from Simulink Function Blocks
`rtwdemo_configinterface`	`ConfigurationInterface`	Create Data Interface Configuration Programmatically
`rtwdemo_configrpinterface`	`ConfigurationRapidPrototypingInterface`	Access Signal, State, and Parameter Data During Execution
`rtwdemo_controlflow_opt`	`ControlFlowOptimization`	Optimize Generated Code by Consolidating Redundant If-Else Statements
`rtwdemo_counter`	`CounterModel`	Register Custom Toolchain and Build Executable
`rtwdemo_cpp`	`CppSFunctionExternalCode`	Integrate External C++ Code into a Model Using S-Functions
`rtwdemo_cppclass_export_functions`	`CppClassFunctionsHarness`	Generate C++ Function and Class Code for Export-Function Model
`rtwdemo_cppclass_functions`	`CppClassFunctions`	Generate C++ Function and Class Code for Export-Function Model

R2023a model name	New model name	Example
rtwdemo_crossrelease_counter	CrossReleaseCounter	Generate Component Source Code for Export to External Code Base
rtwdemo_crossrelease_integration	CrossReleaseIntegration	
rtwdemo_crossrelease_protected_model	CrossReleaseProtectedModel	Use Protected Models from Previous Releases to Perform SIL Testing and Generate Code
rtwdemo_crossrelease_unprotected_counter	CrossReleaseUnprotectedCounter	
rtwdemo_cscpredef	CustomStorageClasses	Create Storage Classes by Using the Custom Storage Class Designer
rtwdemo_dimension_variants	DimensionVariants	Implement Symbolic Dimensions for Array Sizes in Generated Code
rtwdemo_dynamicio	DynamicIO	Share Data Between Code Generated from Simulink, Stateflow, and MATLAB
rtwdemo_fileprocess	CustomFileProcess	Custom File Processing (CFP) Templates
rtwdemo_fixpt1	FixedPointCodeGeneration	Optimize Generated Code Using Fixed-Point Data with Simulink, Stateflow, and MATLAB
rtwdemo_fixptdiv	FixedPointOperations	Optimize Generated Code for Fixed-Point Data Operations
rtwdemo_func_dinteg	DiscreteIntegratorFunction	Generate Reentrant Code from Simulink Function Blocks
rtwdemo_func_dinteg_cpp	FuncDintegCPP	Generate Reentrant Code from Simulink Function Blocks
rtwdemo_irtshared	StartupResetShutdownShared	Startup, Reset, and Shutdown Function Interfaces
rtwdemo_lct_bus	LctStructureArguments	Integrate External C Functions That Use Structure Arguments
rtwdemo_lct_cplxgain	LctComplexSignal	Integrate External C Functions That Pass Input and Output Arguments as Signals with Complex Data
rtwdemo_lct_cpp	LctExternalCodeImportCpp	Integrate External C++ Object Methods
rtwdemo_lct_fixpt_params	LctFixedParams	Integrate External C Functions That Pass Input and Output Arguments as Parameters with a Fixed-Point Data Type

R2023a model name	New model name	Example
rtwdemo_lct_fixpt_signals	LctFixedSignals	Integrate External C Functions That Pass Input and Output Arguments as Signals with a Fixed-Point Data Type
rtwdemo_lct_gain	LctExternalCodeImportGain	Integrate External C Functions That Pass the Output Argument as a Return Argument
rtwdemo_lct_inherit_dims	LctInheritedDimensions	Integrate External C Functions That Pass Arguments That Have Inherited Dimensions
rtwdemo_lct_lut	LctTableLookups	Integrate External C Functions That Implement N-Dimensional Table Lookups
rtwdemo_lct_ndarray	LctMultiDimensionalSig	Integrate External C Functions That Pass Arguments as Multi-Dimensional Signals
rtwdemo_lct_sampletime	LctSampleTime	Integrate External C Functions with a Block Sample Time Specified, Inherited, and Parameterized
rtwdemo_lct_start_term	LctStartTermActions	Integrate External C Functions That Implement Start and Terminate Actions
rtwdemo_lct_work	LctInstanceSpecificMemory	Integrate External C Functions with Instance-Specific Persistent Memory
rtwdemo_memset	MemsetOptimization	Optimize Generated Code Using `memset` Function
rtwdemo_mrmtbb	MultirateMultitaskingBareBoard	Analyze Generated Data Code Interface
rtwdemo_mrmtos	MultirateMultitaskingOS	Time-Based Scheduling Example Models
rtwdemo_mrstbb	MultirateSingletaskingBareBoard	Time-Based Scheduling Example Models
rtwdemo_paraminline	InlineBlockParameters	Inline Numeric Values of Block Parameters
rtwdemo_libcodegen_lib	LibraryCodeGenerationLibrary	Library-Based Code Generation for Reusable Library Subsystems
rtwdemo_libcodegen_mdl	LibraryCodeGeneration	
rtwdemo_localizable_csc	LocalizableStorageClass	Generate Local Variables with `Localizable` Storage Class
rtwdemo_namerules	NameRules	Apply Custom Naming Conventions to Identifiers

R2023a model name	New model name	Example
rtwdemo_nzcheck	DivisionExceptions	Remove Code That Guards Against Division Exceptions for Integers and Fixed-Point Data
rtwdemo_parabuild_a_1	ParallelBuildA1	Reduce Build Time for Referenced Models by Using Parallel Builds
rtwdemo_parabuild_b_1	ParallelBuildB1	
rtwdemo_parabuild_b_2	ParallelBuildB2	
rtwdemo_parabuild_b_3	ParallelBuildB3	
rtwdemo_pmsmfoc	PMSMController	Field-Oriented Control of Permanent Magnet Synchronous Machine
rtwdemo_pmsmfoc_system	PMSMSystem	
rtwdemo_preprocessor	PreprocessorConditionalsUsingVariantModel	Use Variant Models to Generate Code That Uses C Preprocessor Conditionals
rtwdemo_linl	Variants_linl	
rtwdemo_linr	Variants_linr	
rtwdemo_nlinl	Variants_nlinl	
rtwdemo_nlinr	Variants_nlinr	
rtwdemo_preprocessor_subsys	PreprocessorConditionalsUsingVariantSubsystem	Use Variant Subsystem to Generate Code That Uses C Preprocessor Conditionals
rtwdemo_ratetrans	MultirateMultitaskingRateTransitions	Separate Rate Transition Block Code and Data from Algorithm Code and Data
rtwdemo_rtwecintro	EmbeddedCoderIntro	Generate Code Using Embedded Coder
rtwdemo_sfcn_rls	SFunctionForCodeReuse	S-Functions for Code Reuse
rtwdemo_sfcn_rls_lib	SFunctionForCodeReuseLibrary	
rtwdemo_sfcpp	CppStateflowExternalCode	Insert External C++ Code into Stateflow Charts for Code Generation
rtwdemo_shrlib	SharedLibraryCode	Interface to a Development Computer Simulator by Using a Shared Library
rtwdemo_srbb	SingleRateBareBoard	Single-Rate Modeling (Bare Board, No OS)
rtwdemo_ssreuse	GeneratedCodeFunctionReuse	Function Reuse in Generated Code
rtwdemo_symbols	IdentifierFormatting	Control Formatting of Identifiers
rtwdemo_targetsettings	TargetSettings	Configure Run-Time Environment Options

R2023a model name	New model name	Example
`rtwdemo_underspecified_datatype`	`UnderspecifiedDataType`	Use `single` Data Type as Default for Underspecified Types
`rtwdemo_vxworks`	`OSIntegration`	Deploy Generated Component Software to Application Target Platforms

For more information, see Open code generation example models from the documentation or command line.

Enhanced support for stop conditions in C++11 example main

In R2023b, Embedded Coder adds code generation support for conditional stops in example `ert_main.cpp` for models configured for the C++11 language standard. An example of stop conditions now supported includes using a Stop Simulation block in your model.

Avoid MISRA violations by initializing local variables to zero in generated code

Before R2023b, the code generator did not initialize local variables to zero. For example, see this code snippet.

```
/* Model step function */
void LocalDataInitializationExample_step(void)
{
  real_T rtb_Gain4;
  real_T rtb_Sum1;
```

You can now generate code that initializes local variables to zero by clearing the new model configuration parameter **Remove local variable initialization to zero value**. This parameter is selected by default.

Explicitly initialize local variables to zero to avoid generating code that contains violations of MISRA C++:2008 Rule 0-1-4 (Polyspace Bug Finder) and to comply with other necessary coding standards.

For example, see this code snippet.

```
/* Model step function */
void LocalDataInitializationExample_step(void)
{
  real_T rtb_Gain4 = 0.0;
  real_T rtb_Sum1 = 0.0;
```

For more information, see Control Generation of Initialization Code for Local Variables Set to Zero.

Support for multiple uses of $N and $R naming rule tokens for memory sections

Before R2023b, an identifier naming rule for a memory section could not contain more than one instance each of the `$N` and `$R` naming rule tokens. You can now use these naming rule tokens

multiple times within a memory section naming rule. Use multiple instances of these tokens to name memory sections in a way that is consistent with AUTOSAR rules.

For more information about naming rule tokens, see Identifier Format Control.

Quick Start presents relevant next steps based on your modeling style and code generation goals

When you prepare your model for code generation by using Embedded Coder Quick Start, the tool presents relevant next steps based on these criteria of your modeling style and code generation goals:

- Export-function or rate-based model
- Top model or subsystem build
- Single-instance or multi-instance code

The next steps can include opening the code generation report, opening the SIL/PIL Manager app, or exploring examples that show how to further customize the generated code. For more information, see Generate Code by Using the Quick Start Tool.

Speed up generated code execution with Halide

In R2023b, the code generator supports code generation for Halide, a programming language designed for fast and efficient computation on images and tensors. You can generate Halide code from Simulink models by using the new model configuration parameter **Generate Halide code**.

You can generate Halide code for these blocks in a Simulink model:

- Matrix Multiply
- MATLAB Function with the matrix multiplication (*) operation

In R2023b, the code generator supports Halide code generation only when the **Hardware board** parameter is set to `None`.

For computationally intensive operations such as cascaded matrix multiplication, Halide can significantly improve the execution speed of the generated code. For more information, see Speed Up Generated Code Execution with Halide Code.

Stack size includes input arguments and return value in static code metrics report

Starting in R2023b, the static code metrics report includes the sizes of function input arguments and the return value in addition to the non-static local variables to measure the function stack size. You can see the increase in the estimated stack size in the static code metrics report in the **Accumulated Stack Size (bytes)** and **Self Stack Size (bytes)** columns.

Prior to R2023b, only the non-static local variables were considered to measure the function stack size. This is because the static code metrics assumed that the compiler could optimize the function arguments to register memory and was not included to calculate the function stack size.

For more information, see Generate Static Code Metrics Report for Simulink Model.

Use of C99 data types in generated service interface

When you specify a file that contains a service interface configuration for the **Shared coder dictionary** configuration parameter, the software sets the **Data type replacement** configuration parameter to `Use C data types with fixed-width integers`. The code generator creates a service interface that uses C99 data types.

For more information, see:

- **Shared coder dictionary**
- **Data type replacement**
- Embedded Coder Dictionary

Enhancements for code generation with C99 data types

When you set **Data type replacement** to `Use C data types with fixed-width integers`:

- The **Specify custom data type names** (`EnableUserReplacementTypes`) configuration parameter is supported. Instead of generating the C99 data type names and limit identifiers, you can specify custom names and custom limit identifiers.
- The word size values for target and production integer data types do not need to include the values 8 and 16 bits. In addition, a word size value can be a value other than 8, 16, 32, or 64 bits.
- You can run external mode simulations by using XCP, TCP/IP, or serial communication.

For more information, see Manage Replacement of Simulink Data Types in Generated Code.

⚠Functionality being removed or changed

crossReleaseImport supports only last eight releases

`crossReleaseImport` supports the import of generated code from only the previous eight releases. For example, in R2023b, you can use `crossReleaseImport` to import code generated only by releases R2019b to R2023a.

For more information, see Cross-Release Code Integration and `crossReleaseImport`.

Deployment

Deploy AUTOSAR Adaptive Architecture Models Using Embedded Coder Support Package for Linux Applications

In R2023b the Embedded Coder Support Package for Linux Applications has added the following enhancements:

- You can deploy AUTOSAR Adaptive architecture models.
- You can deploy and calibrate AUTOSAR adaptive applications on WSL (Windows Subsystem for Linux) over Windows machine as target.

For more information, see Build Simulink Model and Deploy Application.

Introduction to Custom Hardware App

Starting in R2023b, Custom Hardware App supports the Linux Runtime Manager External Mode workflow. DDS Blockset models and Adaptive AUTOSAR models configured for external mode gets access to the custom hardware app. Use the app to deploy applications on target and perform external mode simulation by using the buttons provided in hardware tab in the toolstrip. Use the Linux Runtime Manager tool to setup and connect to a target.

Customize Groups in ASAP2 File

Starting in R2023b, Embedded Coder allows you to customize groups in ASAP2 file. This feature allows you to add, delete, modify, find, filter, and fetch groups by using the ASAP2 programming interface. For more information, see Export Characteristic and Measurement Objects in Groups.

⚠Functionality being removed or changed

linuxRuntimeManager to replace linux.RuntimeManager.open

In R2023b, `linuxRuntimeManager` function replaces the existing `linux.RuntimeManager.open` function which is used to open the Linux Runtime Manager. For more information, see Linux Runtime Manager.

Export Compu Methods and Record Layouts in Separate Files

Starting from R2023b, the Generate Calibration Files tool for Embedded Coder allows you to generate separate ASAP2 files for compu methods and record layouts for component models. For more information, see coder.asap2.export function.

lcc-win64 compiler will be removed

Warns

The `lcc-win64` compiler will be removed in a future release. For information about supported compilers, see Supported and Compatible Compilers - Windows.

Performance

★Aggregation of code profiling results and identification of worst-case execution

Use a `coder.profile.ExecutionTimeSet` or `coder.profile.ExecutionStackSet` object to aggregate execution-time or stack usage profiles, respectively, from multiple software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations of a model. Then, using the Code Profile Analyzer, identify the most demanding execution and test for each task and view the corresponding code or model path.

For more information, see Aggregate Execution-Time Profiles to Identify Worst Execution and Aggregate Stack Usage Profiles and Identify Most Demanding Task Execution.

★Generate SIMD code for complex data types

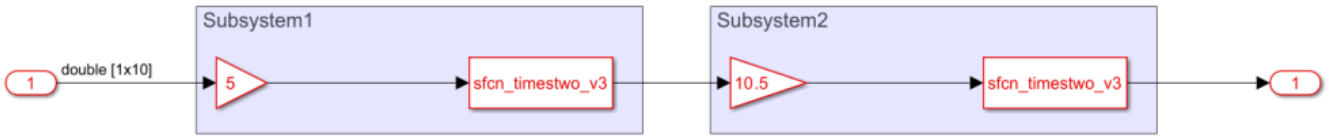
In R2023b, you can generate SIMD code for operations on complex data types that have `single` or `double` base types. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. To generate SIMD code, specify an instruction set to use by setting the model configuration parameter **Leverage target hardware instruction set extensions** and optionally selecting the **Optimize reductions** parameter. For more information, see Generate SIMD Code from Simulink Blocks and Optimize Code for Reduction Operations by Using SIMD.

★Improved code efficiency and better integration of inlined S-Functions

Starting in R2023b, the enhanced TLC block interface more effectively integrates code for S-Function blocks into the generated code of the model as compared to the existing TLC block interface. This improved integration significantly enhances the code quality and offers these benefits:

- Improved buffer allocation for the inputs and outputs of S-Function blocks in the generated code
- Improved inlining of the code generated for the S-Function block
- Reuse of local variables used in `for`-loop iterations
- A more explicit set of cross-release compatible TLC library functions

For example, consider the model `mtimestwo` that includes two identical S-Function blocks.



In this case, the TLC for these S-Functions is expressed as:

```
%function Outputs(block, system) Output
  /* Multiply input by two start */
  %assign rollVars = ["U", "Y"]
  %assign paramVal = (1 == block.SFcnParamSettings.GenUncompilableCode) ? "2_0" : "2.0"
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
```

```
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * %<paramVal>;
  %endroll
  /* Multiply input by two end */
%endfunction
```

This table compares the code generated with the TLC block interface and the enhanced TLC block interface.

The generated code with the enhanced TLC block interface eliminates the redundant temporary variable `i1` and data copies `u0` and `y0`. It also eliminates unnecessary creation of local scope for S-Function blocks.

Code with TLC block interface	Code with enhanced TLC block interface

`void mTLC_timestwo_demo_step(void)`
`{`
`  real_T rtb_Gain[10];`
`  int32_T i;`
`  for (i = 0; i < 10; i++) {`
`    /* Outport: '<Root>/Out1' incorporates:`
`     *  Gain: '<S1>/Gain'`
`     *  Inport: '<Root>/In1'`
`     */`
`    mTLC_timestwo_demo_Y.Out1[i] = 5.0 * mTLC_timestwo_demo_U.In1[i];`
`  }`
 	
`  /* S-Function (sfcn_timestwo_v3): '<S1>/S-Function' incorporates:`
`   *  Outport: '<Root>/Out1'`
`   */`
`  /* Multiply input by two start */`
`  {`
`    int_T i1;`
`    const real_T *u0 = &mTLC_timestwo_demo_Y.Out1[0];`
`    real_T *y0 = &rtb_Gain[0];`
`    for (i1=0; i1 < 10; i1++) {`
`      y0[i1] = u0[i1] * 2.0;`
`    }`
`  }`
 	
`  /* Multiply input by two end */`
 	
`  /* Gain: '<S2>/Gain' */`
`  for (i = 0; i < 10; i++) {`
`    rtb_Gain[i] *= 10.5;`
`  }`
 	
`  /* End of Gain: '<S2>/Gain' */`
 	
`  /* S-Function (sfcn_timestwo_v3): '<S2>/S-Function' incorporates:`
`   *  Outport: '<Root>/Out1'`
`   */`
`  /* Multiply input by two start */`
`  {`
`    int_T i1;`
`    const real_T *u0 = &rtb_Gain[0];`
`    real_T *y0 = &mTLC_timestwo_demo_Y.Out1[0];`
`    for (i1=0; i1 < 10; i1++) {`
`      y0[i1] = u0[i1] * 2.0;`
`    }`
`  }`
 	
`  /* Multiply input by two end */`
`}`
	`void mTLC_timestwo_demo_step(void)`
	`{`
	`  real_T rtb_Gain[10];`
	`  int32_T i;`
	`  for (i = 0; i < 10; i++) {`
	`    /* Outport: '<Root>/Out1' incorporates:`
	`     *  Gain: '<S1>/Gain'`
	`     *  Inport: '<Root>/In1'`
	`     */`
	`    mTLC_timestwo_demo_Y.Out1[i] = 5.0 * mTLC_timestwo_demo_U.In1[i];`
	`  }`
	
	`  /* S-Function (sfcn_timestwo_v3): '<S1>/S-Function' incorporates:`
	`   *  Outport: '<Root>/Out1'`
	`   */`
	`  /* Multiply input by two start */`
	`  for (i = 0; i < 10; i++) {`
	`    rtb_Gain[i] = mTLC_timestwo_demo_Y.Out1[i] * 2.0;`
	`  }`
	
	`  /* Multiply input by two end */`
	
	`  /* Gain: '<S2>/Gain' */`
	`  for (i = 0; i < 10; i++) {`
	`    rtb_Gain[i] *= 10.5;`
	`  }`
	
	`  /* End of Gain: '<S2>/Gain' */`
	
	`  /* S-Function (sfcn_timestwo_v3): '<S2>/S-Function' incorporates:`
	`   *  Outport: '<Root>/Out1'`
	`   */`
	`  /* Multiply input by two start */`
	`  for (i = 0; i < 10; i++) {`
	`    mTLC_timestwo_demo_Y.Out1[i] = rtb_Gain[i] * 2.0;`
	`  }`
	
	`  /* Multiply input by two end */`
	`}`

For more information, see Improve Code Efficiency and Integration of Inlined S-Functions.

Code Profile Analyzer improvements

Use the enhanced Code Profile Analyzer to analyze memory usage and execution-time metrics from software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations. In R2023b, you can:

- Generate PDF reports from the metrics produced by the simulations.
- Show function-call stack memory usage for specific steps of SIL and PIL simulations.

For more information, see View and Compare Code Execution Times and Stack Usage Profiling for Code Generated from Simulink Models.

C++ code profiling for XCP external mode simulations

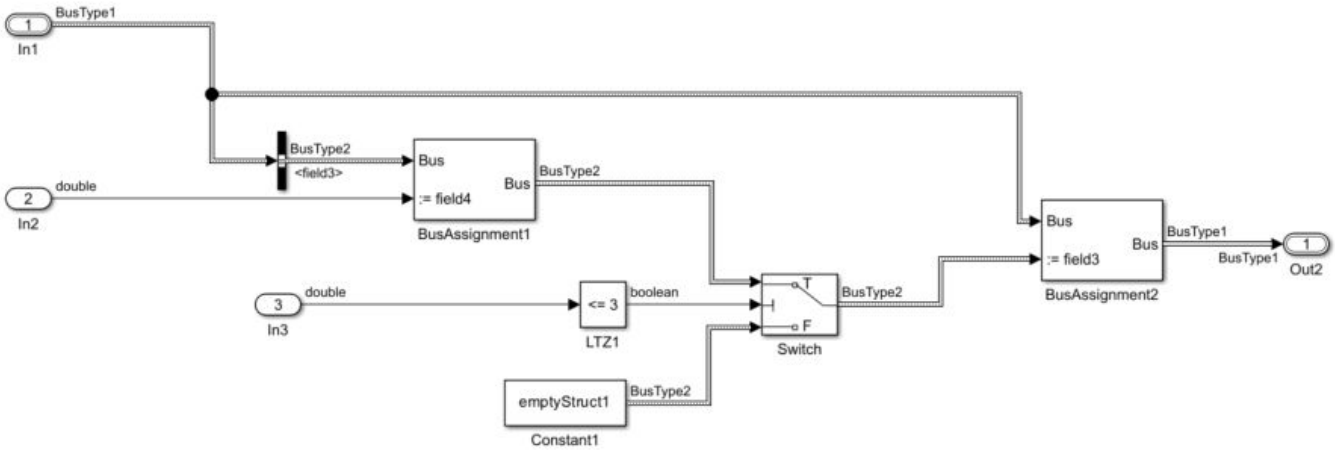
You can now use XCP-based external mode simulations to perform execution-time profiling of generated C++ code. In previous releases, the simulations produce an error if you set the `CodeExecutionProfiling` configuration parameter to `'on'` and the `CodeInterfacePackaging` configuration parameter to `'C++ class'`.

For more information, see Execution Profiling for Generated Code and Create Execution-Time Profile for Generated Code.

Data copy reduction for Bus Assignment blocks that change the value of a nested bus element

Before R2023b, the generated code contained unnecessary temporary variables and data copies for modeling patterns that used multiple Bus Assignment blocks to change the value of a nested bus element. Starting in R2023b, the code generator does not generate the unnecessary temporary variables and data copies. Instead, the generated code performs inplace updates by reusing the input bus buffers. Eliminating unnecessary temporary variables and data copies reduces RAM consumption and improves code execution speed.

Consider the `mBusRouting` model in which the bus signal `BusType1` feeds into the Bus Selector and Bus Assignment block `BusAssignment2`. The Bus Selector block outputs the nested bus element `field3` and the `Bus Assignment1` block changes the value of the `field4` of the `field3`. Depending on the value of `In3`, the `Bus Assignment2` block assigns the new values to the bus element `field3` of the input bus signal.



In R2023a, the code generator produced this code:

```
void mBusRouting_step(void)
{
  BusType2 rtb_Switch;
  rtb_Switch.field4 = rtU.In2;
  if (!(rtU.In3 <= 3.0)) {
    memset(&rtb_Switch, 0, sizeof(BusType2));
  }

  rtY.Out2 = rtU.In1;
  rtY.Out2.field3 = rtb_Switch;
}
```

The generated code contained an unnecessary temporary variable `rtb_Switch` for holding the updated value before assigning the value to `rtY.Out2.field3`.

In R2023b, the code generator produces this code:

```
void mBusRouting_step(void)
{
  rtY.Out2 = rtU.In1;
  rtY.Out2.field3.field4 = rtU.In2;
  if (!(rtU.In3 <= 3.0)) {
    memset(&rtY.Out2.field3, 0, sizeof(BusType2));
  }
}
```

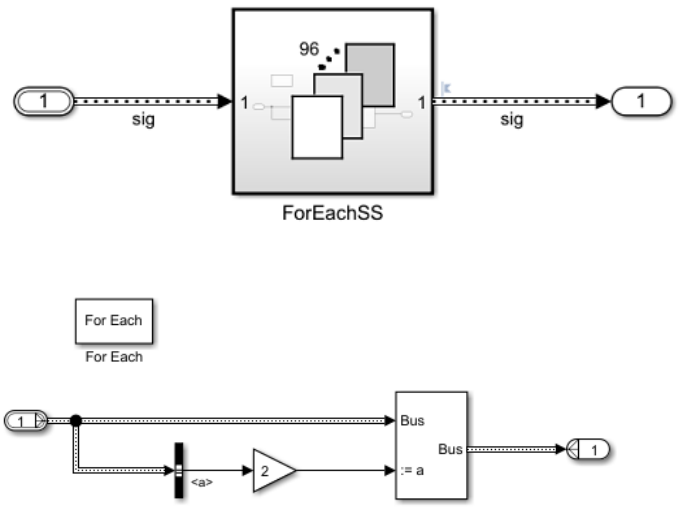
The generated code does not contain the temporary variable `rtb_Switch` for holding the updated value. The code directly assigns the value to `rtY.Out2.field3`. For more information, see Reduce Data Copies for Bus Assignment Blocks in Generated Code.

Data copy reduction for atomic For Each subsystems

Starting in R2023b, the generated code contains fewer data copies for models containing an atomic For Each subsystem whose:

- Input and output signals use the `Reusable` storage class.
- **Function packaging** is set to `inline` or `Nonreuasable` with the `void_void` function interface setting.
- For Each block has the same dimension specified for the input **Partition Dimension** and output **Concatenation Dimension**.

For example, the model `mRCSCInForEachSS` contains an atomic inlined For Each subsystem `ForEachSS`. The input and output signals of `ForEachSS` are configured to use the `Reusable` storage class through the Code Mappings editor.



In R2023a, the code generator produced this code:

```
void mRCSCInForEachSS_step(void)
{
  BusObject sig_1_0[96];
  int32_T ForEach_itr;
  for (ForEach_itr = 0; ForEach_itr < 96; ForEach_itr++) {
    sig_1_0[ForEach_itr].a = 2.0 * sig[ForEach_itr].a;
    sig_1_0[ForEach_itr].b = sig[ForEach_itr].b;
    sig[ForEach_itr] = sig_1_0[ForEach_itr];
  }
}
```

The generated code contained the unnecessary temporary array `sig_1_0` and data copies.

In R2023b, the code generator produces this code:

```
void mRCSCInForEachSS_step(void)
{
  int32_T ForEach_itr;
  for (ForEach_itr = 0; ForEach_itr < 96; ForEach_itr++) {
    sig[ForEach_itr].a = 2.0 * sig[ForEach_itr].a;
  }
}
```

The generated code does not contain the unnecessary temporary array `sig_1_0` and data copies. Instead, the code performs inplace updates by reusing the `sig` buffers. Reusing the buffers reduces RAM consumption and improves code execution speed. For more information, see Specify Buffer Reuse for Signals in a Path.

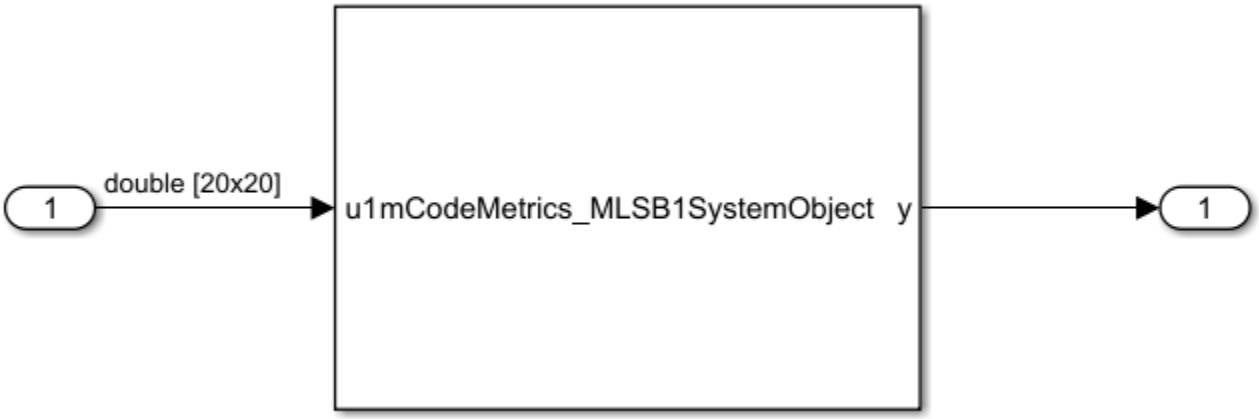
SIMD optimizations for code containing control flow constructs

In R2023b, you can generate SIMD instructions for code that contains control flow constructs, such as blocks containing conditional logic or saturation arithmetic. SIMD code generation is supported for control flow constructs that the code generator can convert to data flow constructs. For some computationally intensive operations, SIMD intrinsics can improve the performance of the generated code on Intel platforms. To generate SIMD code, specify an instruction set to use by setting the model configuration parameter **Leverage target hardware instruction set extensions** and optionally selecting the **Optimize reductions** parameter. For more information, see Generate SIMD Code from Simulink Blocks and Optimize Code for Reduction Operations by Using SIMD.

Improved code efficiency for models containing MATLAB System blocks

In R2023b, the code generator may eliminate unnecessary temporary variables and associated data copies, which reduces RAM consumption and improves code execution speed.

Consider the model `mCodeMetrics_MLSB` with a MATLAB System block.



The following compares code generated in R2023a to the code generated in R2023b for the model. In R2023a, the generated code contains a temporary variable `u0` and redundant data copies of the variable. In R2023b, the temporary variable `u0` and data copies are eliminated.

R2023a Generated Code

```
void mCodeMetrics_MLSB1_step(void)
{
  real_T u0[400];
  real_T i;
  real_T j;
  real_T rtb_MATLABSystem;
  boolean_T found;
```

```
  // MATLABSystem: '<Root>/MATLAB System' incorporates:
  //   Inport: '<Root>/In4'

  std::memcpy(&u0[0], &mCodeMetrics_MLSB1_U.In4[0], 400U * sizeof(real_T));

  //  Implement algorithm. Calculate y as a function of input u and
  //  discrete states.
  // 'mCodeMetrics_MLSB1SystemObject:32' i = 1;
  i = 1.0;

  // 'mCodeMetrics_MLSB1SystemObject:33' found = false;
  found = false;

  // 'mCodeMetrics_MLSB1SystemObject:34' y = 0;
  rtb_MATLABSystem = 0.0;

  // 'mCodeMetrics_MLSB1SystemObject:35' while (i < size(u1, 1)) && (found == false)
  while (i < 20.0 && !found) {
    // 'mCodeMetrics_MLSB1SystemObject:36' j = 1;
    j = 1.0;

    // 'mCodeMetrics_MLSB1SystemObject:37' while (j < size(u1, 2))
    while (j < 20.0 && !(u0[(int32_T)i - 1 + 20 * ((int32_T)j - 1)] == 0.0)) {
      // 'mCodeMetrics_MLSB1SystemObject:38' if u1(i, j) == 0
      // 'mCodeMetrics_MLSB1SystemObject:40' else
      // 'mCodeMetrics_MLSB1SystemObject:41' j = j + 1;
      j = j + 1.0;
    }

    // 'mCodeMetrics_MLSB1SystemObject:45' if (j < size(u1, 2))
    if (j < 20.0) {
      // 'mCodeMetrics_MLSB1SystemObject:46' found = true;
      found = true;

      // 'mCodeMetrics_MLSB1SystemObject:47' y = i + j*size(u1, 1);
      rtb_MATLABSystem = i + j * 20.0;
    }

    // 'mCodeMetrics_MLSB1SystemObject:49' i = i + 1;
    i = i + 1.0;
  }

  // Outport: '<Root>/Out2'
  mCodeMetrics_MLSB1_Y.Out2 = rtb_MATLABSystem;
}
```

R2023b Generated Code

```
void mCodeMetrics_MLSB1_step(void)
{
  real_T i;
  real_T j;
  real_T rtb_MATLABSystem;
  boolean_T found;

  // MATLABSystem: '<Root>/MATLAB System'
  //  Implement algorithm. Calculate y as a function of input u and
  //  discrete states.
  // 'mCodeMetrics_MLSB1SystemObject:32' i = 1;
  i = 1.0;

  // 'mCodeMetrics_MLSB1SystemObject:33' found = false;
  found = false;

  // 'mCodeMetrics_MLSB1SystemObject:34' y = 0;
  rtb_MATLABSystem = 0.0;

  // 'mCodeMetrics_MLSB1SystemObject:35' while (i < size(u1, 1)) && (found == false)
  while (i < 20.0 && !found) {
    // 'mCodeMetrics_MLSB1SystemObject:36' j = 1;
    j = 1.0;
```

```
  // 'mCodeMetrics_MLSB1SystemObject:37' while (j < size(u1, 2))
  while (j < 20.0 && !(mCodeMetrics_MLSB1_U.In4[(int32_T)i - 1 + 20 *
                       ((int32_T)j - 1)] == 0.0)) {
    // 'mCodeMetrics_MLSB1SystemObject:38' if u1(i, j) == 0
    // 'mCodeMetrics_MLSB1SystemObject:40' else
    // 'mCodeMetrics_MLSB1SystemObject:41' j = j + 1;
    j = j + 1.0;
  }

  // 'mCodeMetrics_MLSB1SystemObject:45' if (j < size(u1, 2))
  if (j < 20.0) {
    // 'mCodeMetrics_MLSB1SystemObject:46' found = true;
    found = true;

    // 'mCodeMetrics_MLSB1SystemObject:47' y = i + j*size(u1, 1);
    rtb_MATLABSystem = i + j * 20.0;
  }

  // 'mCodeMetrics_MLSB1SystemObject:49' i = i + 1;
  i = i + 1.0;
}
```

Functionality being removed or changed

Code replacement argument objects use code descriptor types system instead of embedded types system

Behavior change in future release

In R2023b, the `Type` property of code replacement argument objects, such as `RTW.TflArgNumeric` and `RTW.TflArgMatrix` objects, now uses the code descriptor types system instead of the embedded types system. For example, a numeric argument that previously had a `Type` of `embedded.numerictype` now has a `Type` of `coder.descriptor.types.Numeric`.

Verification

SIL/PIL unit-testing of function-call subsystems

You can use the SIL/PIL Manager to unit-test code generated from function-call subsystems, including AUTOSAR runnables in export-function models. For more information, see:

- Unit Test Subsystem Code with SIL/PIL Manager
- Atomic Subsystem Workflow Limitations

PIL connectivity for hardware emulators using Target Framework

To run processor-in-the-loop (PIL) simulations on an emulator, you can define the emulator by using a `target.Emulator` object. The simulation uses the object for starting and stopping the emulator. For more information, see Define Custom Emulator for Target Connectivity.

SIL/PIL data logging for referenced models

For referenced models in a top-model or Model block SIL/PIL simulation, you can log signals and discrete states. During the simulation, stream the signal and state data to the Simulation Data Inspector. At the end of the simulation, store logged data in `logsout` and `xout`.

For more information, see SIL/PIL Manager Verification Workflow.

Hardware Support

★STMicroelectronics STM32 Processors: Support for STMicroelectronics STM32U5xx-based boards

The Embedded Coder Support Package for STMicroelectronics STM32 Processors now supports the new STM32U5xx-based boards. Starting R2023b, you can:

- Generate and build code using an STM32CubeMX project file.
- Use the ADC, PWM, GPIO, Hardware Interrupt, Timer, Encoder, I2C, and UART/USART blocks to implement Model-Based Design workflows.
- Monitor signals and tune parameters in the external mode.
- Run processor-in-the-loop (PIL) simulation in the serial communication mode.

Embedded Coder Support Package for ARM Cortex-A Processors

In R2023b, the Embedded Coder Support Package for ARM Cortex-A Processors documentation is included within the Embedded Coder documentation.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for ARM Cortex-A Processors documentation.

Embedded Coder Support Package for ARM Cortex-R Processors

In R2023b, the Embedded Coder Support Package for ARM Cortex-R Processors documentation is included within the Embedded Coder documentation.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for ARM Cortex-R Processors documentation.

Embedded Coder Support Package for Intel SoC Devices

In R2023b, the Embedded Coder Support Package for Intel SoC Devices documentation is included within the Embedded Coder documentation.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for Intel SoC Devices documentation.

Embedded Coder Support Package for Xilinx Zynq Platform

In R2023b, the Embedded Coder Support Package for AMD SoC Devices documentation is included within the Embedded Coder documentation.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for Xilinx Zynq Platform documentation.

STMicroelectronics STM32 Processors: Support package documentation moved into Embedded Coder documentation

In R2023b, the Embedded Coder Support Package for STMicroelectronics STM32 Processors documentation is included within the Embedded Coder documentation and support package updates are announced in the Embedded Coder release notes.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for STMicroelectronics STM32 Processors documentation.

STMicroelectronics STM32 Processors: Support for CORDIC, Comparator, I2S Audio Out, I2S Mic In, and DAC blocks

Starting R2023b, you can use the following blocks with STM32 processor-based boards.

- Use the CORDIC block to enable hardware acceleration of trigonometric and hyperbolic mathematical functions.
- Use the Comparator block to compare two analog inputs to the peripheral and provide the logical value of the result for STM32G4xx, STM32H7xx, STM32L4xx, and STM32L5xx processor-based boards.
- Use the updated I2S Audio Out block to send an audio stream and I2S Mic In block to read an audio stream on STM32F7xx and STM32G4xx processor-based boards.
- Use the updated Digital to Analog Converter (DAC) block to receive the digital value and convert it to the equivalent analog voltage on a specified channel in STM32F4xx, STM32F7xx, STM32H7xx, STM32L4xx, and STM32L5xx processor-based boards.

STMicroelectronics STM32 Processors: Establish CAN or FDCAN Communication for STMicroelectronics STM32 Processor Based Boards

The Communication Using CAN or FDCAN Blocks for STMicroelectronics STM32 Processor Based Boards example shows how to use CAN or FDCAN blocks and establish CAN or FDCAN communication protocol on STM32 processor based boards.

Using this example, you can implement the CAN communication protocol in the following modes:

- Internal Loopback Mode - Write and read data using CAN/FDCAN Write and CAN/FDCAN Read blocks in internal loopback mode.
- Polling Based - Send and receive data from one board to another using polling-based FDCAN protocol.
- Interrupt Based - Send and receive data from one board to another using interrupt-based FDCAN protocol.
- Interrupt Based and HTS221 Sensor Data - Send more than 8 data bytes at a higher bit rate with Hardware Interrupt block and read temperature and humidity values.
- Classic CAN Interrupt Based - Establish communication between FDCAN and CAN peripherals using the classic CAN frame format. Send the humidity and temperature values from the B-L475E-IOT01A2 board upon receiving the remote frame from NUCLEO-L552ZE-Q.

STMicroelectronics STM32 Processors: Support for PIL executions in MATLAB Using STM32 Cube MX

The Embedded Coder Support Package for STMicroelectronics STM32 Processors supports processor-in-the-loop (PIL) executions using STM32 CubeMX to verify the code generated using:

- MATLAB Coder app
- MATLAB functions using MATLAB Coder at the command line.

STMicroelectronics STM32 Processors: Support for ARM GNU Toolchain 11.3.Rel1, CMSIS V5.9.0, and CMSIS-DSP v1.14.3

In R2023b, enhancements to the Embedded Coder Support Package for STMicroelectronics STM32 Processors include:

- Support for ARM GNU Toolchain 11.3.Rel1
- Support for Cortex Microcontroller Software Interface Standard (CMSIS) v5.9.0. and CMSIS DSP 1.14.3
- Modifications to the CMSIS libraries to improve the performance of the code replacement library (CRL)

ARM Cortex-M Processors: Support package documentation moved into Embedded Coder documentation

In R2023b, the Embedded Coder Support Package for ARM Cortex-M Processors documentation is included within the Embedded Coder documentation.

To view support package documentation from previous releases, see Archived R2023a Embedded Coder Support Package for ARM Cortex-M Processors documentation.

ARM Cortex-M Processors: Support for ARM GNU Toolchain 11.3.Rel1, CMSIS V5.9.0, and CMSIS-DSP v1.14.3

In R2023b, enhancements to the Embedded Coder Support Package for ARM Cortex-M Processors include:

- Support for ARM GNU Toolchain 11.3.Rel1
- Support for Cortex Microcontroller Software Interface Standard (CMSIS) v5.9.0. and CMSIS DSP 1.14.3
- Modifications to the CMSIS libraries to improve the performance of the code replacement library (CRL)

ARM Cortex-M Processors: Support for Log() and Exp() Math Functions using ARM Cortex-M CRL

Starting 2023b, you can generate code for the vector exponential and natural logarithm Simulink blocks and MATLAB functions that calls CMSIS-DSP library using ARM Cortex- M CRL.

ARM Cortex-M Processors: Support for addition and subtraction of fixed point inputs using ARM Cortex-M CRL

Starting 2023b, you can generate code that calls the CRL for addition and subtraction of word lengths 8 and 16 for fixed-point inputs with different fractional lengths.

Infineon AURIX TC4x Microcontrollers: Support for HighTec 6.1.0 Compiler

Starting 2023b, you can compile and run the code using the HighTec 6.1.0 compiler in Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers.

Infineon AURIX TC4x Microcontrollers: Support for Monitor and Tune (External Mode) over Serial

Starting 2023b, you can use the support package to monitor signals and tune parameters in the external mode over the serial communication interface.

Infineon AURIX TC4x Microcontrollers: Support for Infineon Low Level Driver (iLLD) 2.0.1.2.11.

Starting 2023b, the Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers supports the Infineon Low Level Driver (iLLD) 2.0.1.2.11.

Infineon AURIX TC4x Microcontrollers: CRL Support for FFT and IFFT Block Replacement

The code replacement library (CRL) now supports fast Fourier transform (FFT) and inverse fast Fourier transform (IFFT) blocks, which you can use in signal processing algorithms targeting a parallel processing unit (PPU) on the Infineon AURIX TC4x microcontrollers. CRL requires SoC Blockset.

Check bug reports for issues and fixes

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

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

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

R2023a

Code Generation from MATLAB Code

instrumentCode: Add instrumentation to code you already generated for SIL or PIL execution

Starting in R2023a, you can use the `instrumentCode` function to add instrumentation to code you already generated for software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. You can add instrumentation to track stack memory usage, analyze C/C++ code coverage, profile entry-point functions, or profile functions called within entry-point functions. You can also use `instrumentCode` to specify the toolchain to use to build the generated code, and compiler optimization and debug settings for the specified toolchain.

The `instrumentCode` function decouples the code generation and code instrumentation steps. This functionality enables you to analyze the execution behavior of the code you intend to deploy without altering the original functional code. Once you generate the code, you can apply the instrumentation as many times as you need.

For example, first generate code for the entry-point function `foo`. In the code generation configuration object, set the verification mode to SIL. Also, specify the location of the generated files using the `-d` option.

```
cfg = coder.config('lib');
cfg.VerificationMode = 'SIL';
codegen -config cfg foo -args {0} -d my_codegen_folder
```

To add instrumentation to the generated code to profile both entry-point functions and functions called within entry-point functions, call `instrumentCode`:

```
instrumentCode('my_codegen_folder', 'CodeExecutionProfiling', true, 'CodeProfilingInstrumentatio
```

This instrumented code has the same behavior as the code generated with the `CodeExecutionProfiling` and `CodeProfilingInstrumentation` properties enabled in the configuration object `cfg`.

Analyze coverage of C/C++ code during SIL and PIL simulations

Starting in R2023a, you can perform coverage analysis of both the generated C/C++ code and custom C/C++ during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. To include instrumentation that performs coverage analysis in the generated code, do one of the following:

- When generating a SIL or PIL MEX from your MATLAB entry-point functions by using the `codegen` command, set the `coder.EmbeddedCodeConfig` property `CodeCoverage` to `true`. Alternatively, if you generate the SIL or PIL MEX by using the MATLAB Coder app, set the configuration parameter **Enable C/C++ code coverage** to `Yes`.

  For example, to generate a SIL MEX with code coverage enabled for the entry-point function `foo` that accepts a double scalar input, run these commands:

  ```
  cfg = coder.config('lib');
  cfg.VerificationMode = 'SIL';
  cfg.CodeCoverage = true;
  codegen -config cfg foo -args {0}
  ```

- Use the `instrumentCode` function to add instrumentation for coverage analysis to the code you already generated for software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution.

  For example, first generate code for the entry-point function `foo`. In the code generation configuration object, set the verification mode to SIL. Also, specify the location of the generated files using the `-d` option.

  ```
  cfg = coder.config('lib');
  cfg.VerificationMode = 'SIL';
  codegen -config cfg foo -args {0} -d my_codegen_folder
  ```

  To add instrumentation to the generated code to perform coverage analysis, call `instrumentCode`:

  ```
  instrumentCode('my_codegen_folder', 'CodeCoverage', true)
  ```

To use this functionality, you must have a MATLAB Test™ license.

For an example of this workflow, see Add Instrumentation That Performs Coverage Analysis.

Generate execution time profile for custom code during SIL and PIL simulations

Starting in R2023a, the execution time profile report generated during software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation shows execution times for your custom C/C++ code that you integrated with the generated code. In previous releases, the profile showed only execution times for the generated code.

See Generate Execution Time Profile.

Debugging for PIL execution

Provide debugging for processor-in-the-loop (PIL) execution of C and C++ code generated from MATLAB code by following these steps:

1. When you set up PIL connectivity, specify a debugger by using `target.ExecutionService` and `target.DebugExecutionTool` objects.
2. In the MATLAB Coder app, select the **Enable source-level debugging for SIL or PIL** check box. Or, from the command line, set the `SILPILDebugging` property of the `coder.EmbeddedCodeConfig` object to `true`.

In previous releases, debugging is available only for software-in-the-loop (SIL) execution.

In MATLAB scripts, the use of the `coder.EmbeddedCodeConfig.SILDebugging` property is still supported.

For more information, see:

- Support PIL Debugging
- `DebugExecutionTool` Template
- Debug Generated Code During SIL or PIL Execution

Code Profile Analyzer

To analyze execution-time and stack usage profiles produced by software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations, use the Code Profile Analyzer. The new app enables you to:

- Analyze profiling results interactively.
- Investigate the function call-stack for the most demanding simulation step.
- Compare results from different simulations.

For more information, see:

- Code Profile Analyzer
- View Execution Times
- Stack Usage Profiling for Code Generated From MATLAB Code

Generate C/C++ code with annotations to suppress known MISRA C: 2012 and AUTOSAR C++14 violations

In R2023a, you can instruct the code generator to add annotations to the generated C/C++ code for known MISRA C: 2012 and AUTOSAR C++14 violations. These annotations enable static analysis tools, such as Polyspace®, to automatically recognize these comments and report the annotated violations as **Justified**.

To add MISRA C: 2012 and AUTOSAR C++14 annotations from the MATLAB Coder or Embedded Coder, select **Generate justification comments for known MISRA violations** from the **MISRA Compliance** tab and **Include comments** from the **Code Appearance** tab.

Alternatively, you can set the `JustifyMISRAViolations` and `GenerateComments` configuration options of your model to `true`.

The follow shows the generated C++ code with `JustifyMISRAViolations` set to `false` and code with `JustifyMISRAViolations` set to `true`.

MATLAB Code

```
function result = addNumbers(x, y)
  result = x + y;
end
```

Generated C++ Code with `JustifyMISRAViolations` set to `false` (default)

```
// Function Definitions
//
// Arguments    : void
// Return Type  : void
//
namespace sample {
void addNumbers_initialize()
{
}

} // namespace sample
```

Generated C++ Code with `JustifyMISRAViolations` set to `true`

```
// Function Definitions
//
// Arguments    : void
// Return Type  : void
//
namespace sample {
//
// MW:begin MISRA-CPP:0-1-8
// "Justified for external interface function"
// MW:begin AUTOSAR-CPP14:M0-1-8
// "Justified for external interface function"
void addNumbers_initialize()
{
}
//
// MW:end MISRA-CPP:0-1-8
// MW:end AUTOSAR-CPP14:M0-1-8

} // namespace sample
```

For more information on how to generate code that has improved MISRA C: 2012 and AUTOSAR C++14 compliance, see Generate C/C++ Code with Improved MISRA and AUTOSAR Compliance.

Reduction of violations for AUTOSAR C++14 rules in generated code

Starting in R2023a, the generated code has fewer violations of several rules in the required categories of the AUTOSAR C++14 coding standard. Some of these rules are:

- Standard conversions: `AUTOSAR C++14 Rule A4-10-1` (Polyspace Bug Finder)
- Declarators: `AUTOSAR C++14 Rule A8-4-9` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A8-4-10` (Polyspace Bug Finder)
- Preprocessing directives: `AUTOSAR C++14 Rule A16-2-2` (Polyspace Bug Finder)

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

Model Architecture and Design

Unused variable and macro elimination for Variant blocks in generated code

Prior to R2023a, when you generated code for a variant block with its **Variant activation time** set to `code compile`, the code generator added an unused variable inside the corresponding model data structure containing pre-processor conditionals. During code compilation, if the conditions inside the structure evaluated to `false`, the unused variable prevented the creation of an empty structure in the compiled code. However, adding the unused variable violated MISRA standard.

Starting in R2023a, if the structure has mutually exclusive conditions, the code generator eliminates unused variables. Also, if the structure has at least one unconditional variable, the code generator eliminates the macros that generate unused variables.

The following shows the code generated from the `slexVariantControlVariableChoices` model in R2022b and R2023a. In R2023a, the code generator eliminates the macro for unused variables because the structure contains an unconditional variable `In1`. During code compilation, even if the conditions inside the structure evaluate to `false`, `In1` is compiled, thus preventing the creation of an empty structure in the compiled code.

R2022b Generated Code

```
typedef struct {
  real_T In1;                          /* '<Root>/In1' */

#if v == 1
  real_T In2;                          /* '<Root>/In2' */
#define EXTU_VARIANT_EXISTS
#endif

#if v == 2
  real_T In3;                          /* '<Root>/In3' */
#define EXTU_VARIANT_EXISTS
#endif

} ExtU;
```

R2023a Generated Code

```
typedef struct {
  real_T In1;                          /* '<Root>/In1' */

#if v == 1
  real_T In2;                          /* '<Root>/In2' */
#endif

#if v == 2
  real_T In3;                          /* '<Root>/In3' */
#endif

} ExtU;
```

Improve code readability of variant blocks and variant parameters by placing utassert statements in a separate function

Starting in R2023a, when you generate code for variant blocks or variant parameters with a startup activation time, the code generator places `utassert` statements in a `startupVariantChecker` function. The `startupVariantChecker` function is then called in the `model_initiaize` function.

Placing `utassert` statements in a separate function makes the code more readable and understandable. Previously, the code generator placed `utassert` statements directly in `model_initialize`.

To avoid function name collisions during code compilation, by default the code generator mangles the `startupVariantChecker` function name to be the first nine characters of the model name followed by `_startupVariantChecker`.

This table compares the code generated in R2022b and R2023a for the model `slexVariantSubsystems`, which has a Variant Subsystem block with an activation time of `startup`.

R2022b Generated Code	R2023a Generated Code

`void slexVariantSubsystems_initialize(void)`
`{`
`  /* Enable for Sin: '<Root>/sine1' */`
`  slexVariantSubsystems_DW.systemEnable = 1;`
 	
`  /* Enable for Sin: '<Root>/sine2' */`
`  slexVariantSubsystems_DW.systemEnable_e = 1;`
 	
`  /* Enable for Sin: '<Root>/sine3' */`
`  slexVariantSubsystems_DW.systemEnable_b = 1;`
 	
`  /* startup variant condition checks */`
`  utAssert(VSS_LINEAR_CONTROLLER() +`
`  VSS_NONLINEAR_CONTROLLER() == 1);`
`}`
	`void slexVariantSubsystems_initialize(void)`
	`{`
	`  /* Enable for Sin: '<Root>/sine1' */`
	`  slexVariantSubsystems_DW.systemEnable = 1;`
	
	`  /* Enable for Sin: '<Root>/sine2' */`
	`  slexVariantSubsystems_DW.systemEnable_e = 1;`
	
	`  /* Enable for Sin: '<Root>/sine3' */`
	`  slexVariantSubsystems_DW.systemEnable_b = 1;`
	`  slexVaria_startupVariantChecker();`
	`}`
	
	`static void slexVaria_startupVariantChecker(void)`
	`{`
	`  /* startup variant condition checks */`
	`  utAssert(VSS_LINEAR_CONTROLLER() +`
	`  VSS_NONLINEAR_CONTROLLER() == 1);`
	`}`

Group variant parameter values in a single structure array in generated code

Starting in R2023a, you can use variant parameter banks to group variant parameters that have the same set of variant conditions into a structure array in the generated code. The code uses a pointer variable to access values from the structure array. The code generator initializes the pointer based on variant conditions in the `model_initialize` function. The code generator supports variant parameter banks only for variant parameters with a `startup` variant activation time.

Prior to R2023a, the code generator inlined values of the variant parameters with `startup` activation time in the `model_initialize` function, which involved reading and copying parameter values into the program memory.

For more information and an example, see *Release Notes for Simulink*.

Model Advisor checks for component deployment using a service interface configuration

Starting in R2023a, you can use these Embedded Coder Model Advisor checks to verify compliance of your model with the guidelines.

Model Advisor Check	Modeling Guideline
Check Startup and Shutdown Event (`mathworks.codegen.cgsl_0404`)	cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment
Check usage of partial data send (`mathworks.codegen.cgsl_0408`)	cgsl_0408: Partial data send for component deployment

Code Interface Configuration and Integration

Component timer service interface enhancements

In R2022b, for aperiodic exported functions that include Discrete Time Integrator and Weighted Sample Time blocks and rely on elapsed time values, Embedded Coder introduced support for configuring and generating timer service interfaces for accessing the function clock tick that is used by the target environment. Starting in R2023a, you can configure and generate timer service interface code for:

- Entry-point functions generated from export-function and single-rate, rate-based models.
- Periodic entry-point functions generated from models that use blocks that rely on absolute time values, such as Sine Wave and Pulse Generator blocks.
- Periodic entry-point functions generated from models that use blocks that rely on elapsed time values in an aperiodic context. An example of an aperiodic context is when the **Sample time type** parameter of the Trigger Port block of the model or subsystem that includes the time-based block is set to `triggered`.

Within a model, you represent requests for a clock tick implicitly when you use blocks that rely on a time value. For these blocks, depending on the context, the code generator assumes that the clock resolution is the sample period of the function or fixed-step size (fundamental sample time) of the model. You can override the clock resolution with a target environment clock resolution by setting model configuration parameter **Clock resolution** (see “Improve code generated for functions that include blocks that request time values by specifying target platform clock resolution” on page 7-11). The code generator produces a timer service interface based on:

- Content of the model
- Setting of the **Clock resolution** model configuration parameter
- Timer service interface configuration in the shared Embedded Coder Dictionary attached to the model

For more information about timer service interfaces, see:

- cgsl_0410: Timer service for component deployment
- Configure Timer Service Interfaces by Using the Code Mappings Editor
- Configure Timer Service Interfaces Programmatically
- Generate C Timer Service Interface Code for Component Deployment
- Create a Service Interface Configuration
- Data Communication Methods

Compatibility Considerations

Starting in R2023a, for these types of models that you configure with a service code interface, the code generator abstracts the timer requests by including calls to a target platform timer service:

- Models that include periodic functions that use blocks that rely on absolute time values
- Export-function models that include periodic functions that use Discrete Time Integrator or Weighted Sample Time blocks (rely on elapsed time values) within an aperiodic context

For models configured with a service code interface:

- Use of an S-Function block that relies on elapsed time is not supported. The code generator produces an error.
- It is the responsibility of the code integrator to provide the implementation of the called timer service.

  For example, in this generated code fragment, the `TimerService_tick_ComponentDeploymentFcn_Periodic()` function calls initiate timer requests. It is the responsibility of the code integrator to provide an implementation for the function `TimerService_tick_ComponentDeploymentFcn_Periodic`.

```
.
.
.
if (zcEvent != NO_ZCEVENT) {
  if (rtDWork.Subsystem_RESET_ELAPS_T) {
    Subsystem_ELAPS_T = 0U;
  } else {
    Subsystem_ELAPS_T = (uint32_T)
      (TimerService_tick_ComponentDeploymentFcn_Periodic() -
       rtDWork.Subsystem_PREV_T);
  }

  rtDWork.Subsystem_PREV_T =
    TimerService_tick_ComponentDeploymentFcn_Periodic();
```

  Prior to R2023a, for the same model, the code generator implemented the function clock tick code within the generated periodic entry-point function and used the execution interval specified for the Simulink function as the function clock resolution. In this example, the code generator implements the function clock tick as `rtM->Timing.clockTick1` and maintains the function within the algorithmic code as `rtM->Timing.clockTick1++`.

```
.
.
.
if (zcEvent != NO_ZCEVENT) {
  if (rtDWork.Subsystem_RESET_ELAPS_T) {
    Subsystem_ELAPS_T = 0U;
  } else {
    Subsystem_ELAPS_T = (uint32_T) (rtM->Timing.ClockTick1 -
       rtDWork.Subsystem_PREV_T);
  }

  rtDWork.Subsystem_PREV_T = rtM->Timing.clockTick1;
.
.
.
  rtM->Timing.clockTick1++;
.
.
.
```

Improve code generated for functions that include blocks that request time values by specifying target platform clock resolution

Starting in R2023a, for model functions that include blocks that request absolute or elapsed time values, you can improve the entry-point code generated for those functions by configuring the model to use a specific clock resolution. Clock resolution is the smallest increment of a clock value. For example, if a clock increments its value once per second, the clock resolution is 1 second.

Benefits of specifying a clock resolution include:

- Clock resolution that aligns with target environment clock requirements. For models configured to use a service code interface, specifying a clock resolution results in code that reads time values that produce more accurate results.
- Decoupling of the clock resolution and the solver properties that Simulink uses during simulation, such as the fixed-step size and sample times. The decoupling enables you to generate code that aligns with the target environment clock resolution and produces more accurate results.
- Influencing the data type that Simulink and the code generator use to represent time values. For example, in normal, accelerator, and rapid accelerator modes, Simulink uses the specified clock resolution to deduce fixed-point data types, which produces fixed-point simulation and generated code execution output that match.
- Portability of models between code interface configurations. You can attach a model that has a specified clock resolution to a shared Embedded Coder Dictionary that defines a data or service code interface configuration.
- Generated code that is easier to read.

To specify a clock resolution for the code generator to apply for a model, configure the model with these model configuration parameter settings:

- Set **System target file** (`SystemTargetFile`) to `ert.tlc`.
- Set solver parameter **Type** (`SolverType`) to `Fixed-step`.
- Set **Clock resolution** (`ClockResolution`) to a scalar value of type `double`, which represents the clock resolution in seconds.

For more information, see **Clock resolution** and Specify Clock Resolution Used by Target Environment Clock.

Use code definitions from packages in service interface configurations

In R2023a, you can use code definitions from a package in an Embedded Coder Dictionary service interface configuration. Previously, you could use package-based code definitions in only data interface configurations. Loading storage classes from a package enables you to simultaneously:

- Use existing storage classes that you defined by using the Custom Storage Class Designer
- Utilize the configuration and mapping capabilities of a service interface configuration

For more information, see Refer to Code Generation Definitions in a Package.

Generate code using built-in FFTW library

The required FFTW library is shipped with MATLAB and the code generation process is simpler in R2023a. You can generate code for models by using the shipped FFTW library and by selecting model configuration parameter **Built-in FFTW library callback**.

Prior to R2023a, to generate code by using the FFTW library, you had to install the FFTW library, write a custom callback class to specify the FFTW library installation using `coder.fftw.StandaloneFFTW3Interface`, and then set the model configuration parameter **Custom FFT library callback** to the name of the callback class. See Speed Up Fast Fourier Transforms in Code Generated from a MATLAB Function Block.

Coexisting code mapping configurations for data and service interfaces

Starting in R2023a, you can configure your models to have both data and service code interface mappings. Once configured, you can switch between the configurations to activate either the data or the service interface configuration. For example, when you switch a model from a data to a service interface, the data interface code mappings are retained for future use. When you switch the model back to using a data interface configuration, the saved data interface code mappings are reactivated. To learn more about mapping configurations, see Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture.

Convert subsystems with service interface mappings to referenced models

Starting in R2023a, when converting a subsystem to a referenced model, you can copy service interface code mappings of the parent model to the created referenced model. Code mappings for the following modeling elements are copied to the newly created model:

- Signals
- States
- Data stores
- Model parameters and model parameter arguments
- Function callers

To convert a subsystem to a referenced model along with the code mappings, use the Model Reference Conversion Advisor with the **Copy code mappings** parameter selected. Alternatively, you can use the `Simulink.SubSystem.convertToModelReference` function with the `CopyCodeMappings` argument specified as `true`. If the original model that contains the converted subsystem is set to the `Automatic` deployment type, then the newly created reference model is set to the `Automatic` deployment type as well. Otherwise, the newly created model is set to the `Subcomponent` deployment type. To learn more about code mapping configurations, see Define Service Interfaces, Storage Classes, Memory Sections, and Function Templates for Software Architecture. To learn more about converting subsystems to referenced models, see Convert Subsystems to Referenced Models.

Automatic deployment type for models with a service interface code configuration

In 2023a, you can now set the deployment type of models configured with a service interface configuration to `'Automatic'`. The code generator uses the deployment type to:

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

When you set the deployment type to `'Automatic'`, Embedded Coder determines the deployment type based on the model hierarchy context. In previous releases, `'Automatic'` was only supported for models with a data interface configuration.

For more information, see `setDeploymentType` and Configure C Code Deployment Types for Model Hierarchy.

Automatic code suggestions and completions for code mappings programming interface

Starting in R2023a, the `coder.mapping.api.CodeMapping` object and its functions support tab completion. After you enter the first few characters of a function, input argument, or object property, press the **Tab** key to let MATLAB automatically complete the typing. MATLAB adds the remaining characters of the function, argument, or property. If you do not enter anything or if there are multiple options that begin with the characters you enter, MATLAB opens a list of available alternatives you can choose from. To learn more about tab completion, see Code Suggestions and Completions.

Code Generation

Example models attached to examples and renamed

In R2023a, these example models have been renamed and are available in the examples indicated in this table.

R2022b model name	New model name	Example
`rtwdemo_accel_send`	`AccelerometerSendMessages`	Model Message-Based Communication Integrated with POSIX Message Queues
`rtwdemo_asap2`	`ASAP2Demo`	Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration
`rtwdemo_asap2_mdlref`	`ASAP2DemoModelRef`	Create a Host-Based ASAM-ASAP2 Data Definition File for Data Measurement and Calibration
`rtwdemo_caller`	`SimulinkFunctionCaller`	Generate Code for Simulink Function and Function Caller
`rtwdemo_col_interpselsubtable`	`SubtableInterpolationCol`	Interpolation with Subtable Selection Algorithm for Row-Major Array Layout
`rtwdemo_differentsizereuse`	`DifferentSizeReuse`	Reuse Buffers of Different Sizes and Dimensions
`rtwdemo_export_functions`	`SimulinkFunctionsTestHarness`	Generate Code for Simulink Function and Function Caller
`rtwdemo_float_mul_for_net_slope_correction`	`FloatMultiplicationNetSlope`	Floating-Point Multiplication to Handle a Net Slope Correction
`rtwdemo_forloop`	`ForLoopConstruct`	Optimize Generated Code by Combining Multiple `for` Constructs
`rtwdemo_functions`	`SimulinkFunctions`	Generate Code for Simulink Function and Function Caller
`rtwdemo_getset_matrix`	`GetSetMatrix`	Use `GetSet` with Matrix Data
`rtwdemo_getset_scalar`	`GetSetScalar`	Access Legacy Data Using Get and Set Functions
`rtwdemo_getset_struct`	`GetSetStruct`	Use `GetSet` with Structured Data
`rtwdemo_getset_vector`	`GetSetVector`	Use `GetSet` with Vector Data
`rtwdemo_gps_send`	`GPSSendMessages`	Model Message-Based Communication Integrated with POSIX Message Queues

R2022b model name	New model name	Example
rtwdemo_inline_invariant_signals	InvariantSignalsInline	Optimize Generated Code Using Inline Invariant Signals
rtwdemo_internal_init	InternalZeroInitialization	Remove Zero Initialization Code for Internal Data
rtwdemo_label_guided_reuse	SignalLabelReuse	Optimize Generated Code by Using Signal Labels to Guide Buffer Reuse
rtwdemo_logicalAsBoolean	LogicalAsBoolean	Optimize Generated Code Using Boolean Data for Logical Signals
rtwdemo_minmax	MinMaxOptimization	Optimize Generated Code Using Minimum and Maximum Values
rtwdemo_optimize_global	MinimizeGlobalDataAccess	Minimize Global Data Access
rtwdemo_optimize_global_ebf	UseGlobalsForTemporaryResults	Use Global to Hold Temporary Results
rtwdemo_optionalDisableResetFunc_bot	DisableResetFunctionBottom	Remove Reset and Disable Functions from the Generated Code
rtwdemo_optionalDisableResetFunc_top	DisableResetFunctionTop	Remove Reset and Disable Functions from the Generated Code
rtwdemo_pack_boolean	PackBooleanData	Optimize Generated Code by Packing Boolean Data into Bitfields
rtwdemo_parentheses	ParenthesizationStyle	Control Use of Parentheses
rtwdemo_pos_estimate	PositionEstimateMessages	Model Message-Based Communication Integrated with POSIX Message Queues
rtwdemo_preservedimensions	PreserveArrayDims	Preserve Dimensions of Multidimensional Arrays in Generated Code
rtwdemo_preservedimensions_slbus	PreserveBusDims	Preserve Dimensions of Bus Elements in Generated Code
rtwdemo_reusable	Reusable	Generate Reentrant Code from Top Models
rtwdemo_reusable_csc	ReusableStorageClass	Specify Buffer Reuse for Signals in a Path
rtwdemo_reuse_global	GlobalReuse	Reuse Global Block Outputs in the Generated Code
rtwdemo_roll	RollAxisAutopilot	Generate C Code from Simulink Models
rtwdemo_roll_harness	RollAxisAutopilotHarness	Generate C Code from Simulink Models

R2022b model name	New model name	Example
`rtwdemo_rootlevel_zero_initialization`	`RootZeroInitialization`	Remove Initialization Code from Root-Level Inports and Outports Set to Zero
`rtwdemo_row_interpselsubtable`	`SubtableInterpolationRow`	Interpolation with Subtable Selection Algorithm for Row-Major Array Layout
`rtwdemo_row_lutcol2row_workflow`	`RowLUTColToRow`	Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks
`rtwdemo_row_lutcol2row_workflow_rowrow`	`RowLUTColToRowPreconfigured`	Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks
`rtwdemo_rsim_param_tuning`	`RsimParamTuning`	Tune Parameters Interactively During Rapid Simulation
`rtwdemo_secondOrderSystem`	`SecondOrderSystem`	Generate C Code for a Model
`rtwdemo_slexprfold`	`FoldBlockComputations`	Fold Expressions
`rtwdemo_unicode`	`MixedLanguagesAndLocales`	Internationalization and Code Generation

In addition to searching in Help Center, you can use the functions modelfinder and openExample to find models and open examples.

Replacement of Simulink data types with C99 data types

Use the new Data type replacement (`DataTypeReplacement`) configuration parameter to specify the method for replacing Simulink data types in generated code. If you select the option that uses data types from the C99 language standard, you can improve generated code compliance with the MISRA C and MISRA C++ standards.

In previous releases:

- Data types are specified in the `rtwtypes.h` file and are based on the C89 language standard.
- To rename data types in generated code, you use the **Replace data type names in the generated code** configuration parameter. In R2023a, **Replace data type names in the generated code** is called Specify custom data type names. The `EnableUserReplacementTypes` command-line parameter is unchanged.

This table summarizes the changes.

Configuration Parameter		Options		Comments
Dialog Box	**Command Line**	**Dialog Box**	**Command Line**	
Data type replacement (new)	`DataTypeReplacement` (new)	`Use coder typedefs`	`'CoderTypedefs'`	If you select this option: • The code generator creates the `rtwtypes.h` header file, which specifies data types that are based on the C89 language standard. • The renamed configuration parameter **Specify custom data type names** (`EnableUserReplacementTypes`) is available.

Configuration Parameter		Options		Comments
Dialog Box	**Command Line**	**Dialog Box**	**Command Line**	
		`Use C data types with fixed-width integers`	`'CDataTypesFixedWidth'`	If you select this option, the generated code: • Uses data types from the C99 language standard, which includes definitions from the `stdint.h`, `stdbool.h`, and `complex_types.h` header files. • Does not require definitions from the `rtwtypes.h` header file. By default, the code generator does not create `rtwtypes.h`. • The advanced parameter **Coder typedefs compatibility** (`CoderTypedefsCompatibility`) is available.
Coder typedefs compatibility (new)	`CoderTypedefsCompatibility` (new)	`on`	`'on'`	If you use legacy custom code or static source files (under `matlabroot`) that require Simulink Coder data type definitions, you can force the generation of `rtwtypes.h` by selecting this option.

Configuration Parameter		Options		Comments
Dialog Box	**Command Line**	**Dialog Box**	**Command Line**	
		`off` (default)	`'off'` (default)	If you select this option, the code generator does not create `rtwtypes.h`.
Specify custom data type names (previously called **Replace data type names in the generated code**)	`EnableUserReplacementTypes` (unchanged)	`on`	`'on'`	Functionality for the renamed configuration parameter is unchanged. If you open a model that you created in a previous release, the software sets `DataTypeReplacement` to `'CoderTypedefs'` but does not change the value of `EnableUserReplacementTypes`.
		`off` (default)	`'off'` (default)	

If you use TLC files that contain hard-coded instances of Simulink Coder data types, you can modify the files for C99 code generation by using the `coder.updateTlcForLanguageStandardTypes` function.

For more information, see:

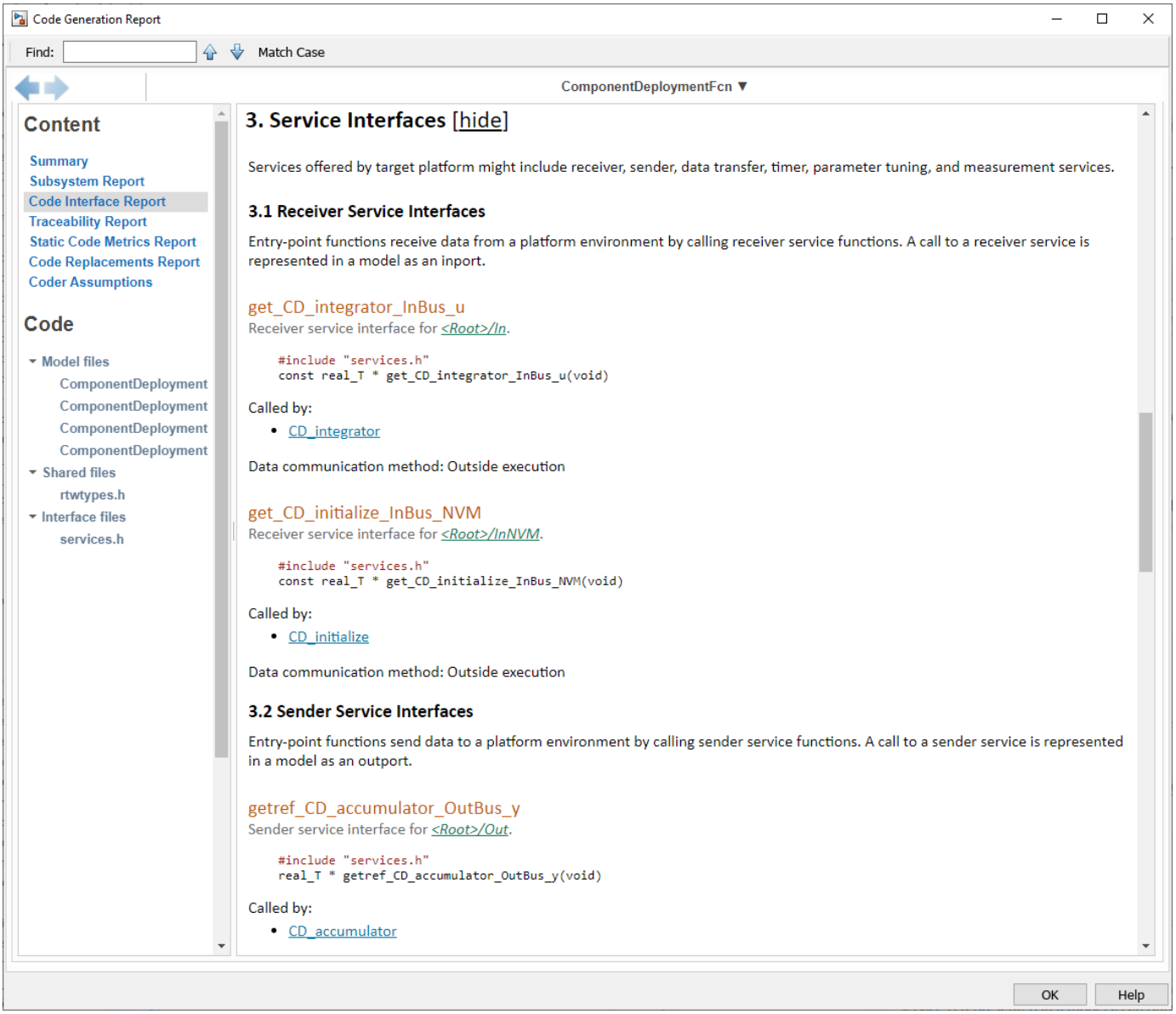
- Manage Replacement of Simulink Data Types in Generated Code
- Compatibility of TLC Files with Generated Code Data Types

Code interface report improvements for service interfaces

In R2023a, when you generate code using the service interface configuration, you can more easily assess the generated code interfaces by using the improved formatting and hyperlinks in the code interface report. The code interface report now includes these improvements:

- Table of contents with hyperlinks to report sections
- Service interfaces documented separately from execution function interfaces
- Hyperlinks between execution functions and the services they call
- Formatted documentation that shows how each service appears or is called in the generated code
- Documentation of measurement service interfaces

These improvements are not available for code generated using the data interface configuration.



C++ code generation support for models configured with service interfaces and nonreusable function code interface packaging

In R2023a, you can generate C++ code from models that:

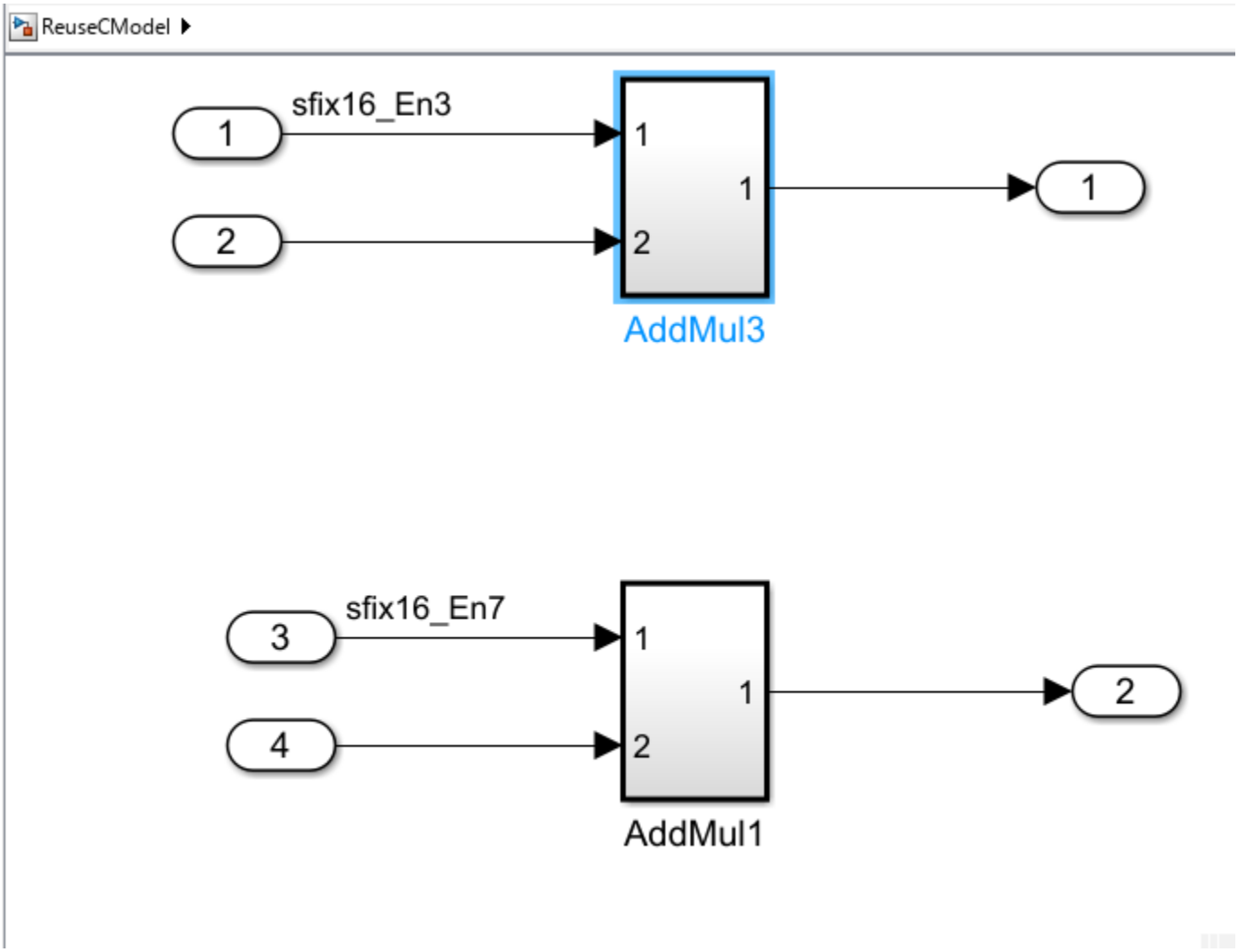
- Are linked to a shared Embedded Coder Dictionary that defines a service code interface configuration
- Have the Code interface packaging model configuration parameter set to `Nonreusable function`

R2023a does not support C++ code generation for service interfaces if the **Code interface packaging** configuration parameter is set to `Reusable function` or `C++ class`. For more information about service interfaces, see Service Interfaces.

Optimized C code for reusable subsystems

The code generator now eliminates duplicate code that it creates for multiple instances of a reusable subsystem with different fixed-point datatype inputs. Prior to R2023a, the code generator generated individual C functions for each instance of the reusable subsystem with different fixed-point datatype inputs.

Consider the model, `ReuseCModel`.



The following compares the code generated in R2022b and R2023a. In R2022b, the generated code includes function `ReuseCModel_AddMul3_e`, which is a duplicate of function `ReuseCModel_AddMul3`. The function `ReuseCModel_AddMul3_e` is removed from R2023a code.

R2022b Generated Code

```
/* Output and update for atomic system: '<Root>/AddMul1' */
int16_T ReuseCModel_AddMul3(int16_T rtu_In1, int16_T rtu_In2)
{
  /* Gain: '<S1>/Gain' incorporates:
   *  Sum: '<S1>/Add'
   */
```

```
  return (int16_T)((int16_T)(rtu_In1 + rtu_In2) * 3);
}

/* Output and update for atomic system: '<Root>/AddMul3' */
int16_T ReuseCModel_AddMul3_e(int16_T rtu_In1, int16_T rtu_In2)
{
  /* Gain: '<S2>/Gain' incorporates:
   *  Sum: '<S2>/Add'
   */
  return (int16_T)((int16_T)(rtu_In1 + rtu_In2) * 3);
}
/* Model step function */
void ReuseCModel_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/AddMul3' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  ReuseCModel_Y.Out1 = ReuseCModel_AddMul3_e(ReuseCModel_U.In1,
    ReuseCModel_U.In2);

  /* End of Outputs for SubSystem: '<Root>/AddMul3' */

  /* Outputs for Atomic SubSystem: '<Root>/AddMul1' */

  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   */
  ReuseCModel_Y.Out2 = ReuseCModel_AddMul3(ReuseCModel_U.In3, ReuseCModel_U.In4);

  /* End of Outputs for SubSystem: '<Root>/AddMul1' */
}
```

R2023a Generated Code

```
/* Output and update for atomic system: '<Root>/AddMul1' */
int16_T ReuseCModel_AddMul3(int16_T rtu_In1, int16_T rtu_In2)
{
  /* Gain: '<S1>/Gain' incorporates:
   *  Sum: '<S1>/Add'
   */
  return (int16_T)((int16_T)(rtu_In1 + rtu_In2) * 3);
}

/* Model step function */
void ReuseCModel_step(void)
{
  /* Outputs for Atomic SubSystem: '<Root>/AddMul3' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   */
  ReuseCModel_Y.Out1 = ReuseCModel_AddMul3(ReuseCModel_U.In1, ReuseCModel_U.In2);

  /* End of Outputs for SubSystem: '<Root>/AddMul3' */

  /* Outputs for Atomic SubSystem: '<Root>/AddMul1' */

  /* Outport: '<Root>/Out2' incorporates:
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   */
  ReuseCModel_Y.Out2 = ReuseCModel_AddMul3(ReuseCModel_U.In3, ReuseCModel_U.In4);

  /* End of Outputs for SubSystem: '<Root>/AddMul1' */
}
```

For more information, see Generate Reusable Code for Subsystems Shared Across Models.

Code replacement validation check detects unspecified rounding modes for multiplication

In R2023a, when you create a code replacement entry for a multiplication operation that can lose precision during rounding, you must specify the **Rounding mode** (`RoundingModes`) for the entry. When you validate these entries, they produce a warning if you do not specify the required setting. The new validation check enables you to identify entries that can lead to unintended replacements in the generated code and produce different results from the model.

Previously, the validation check reported these entries as valid even if the rounding settings were not specified. When the settings were not specified, operations with different rounding needs could map to the same code replacement entry, leading to generated code that produced different results from the model. For more information, see `Rounding modes`.

Compatibility Considerations

Some code replacement entries for multiplication operations that Embedded Coder previously reported as valid produce warnings in R2023a. To make the entries valid, specify the **Rounding mode** for the entries. In a future release, the validation check will produce errors instead of warnings.

Embedded Coder features available in Simulink Online

In R2023a, you can use most of the Embedded Coder features through your web browser for teaching, learning, and convenient lightweight access. To access these features, sign in with your MathWorks® account. For more information, visit the Simulink Online product page.

Functionality being removed or changed

crossReleaseImport will support only last eight releases
Warns

In future releases, `crossReleaseImport` will support the import of generated code from only the previous eight releases. For example, in R2023b, you will be able to use `crossReleaseImport` to import only code generated by releases R2019b to R2023a.

For more information, see Cross-Release Code Integration and `crossReleaseImport`.

lcc-win64 compiler will be removed
Still runs

The `lcc-win64` compiler will be removed in a future release. For information about supported compilers, see Supported and Compatible Compilers - Windows.

Deployment

Embedded Coder Support Package for Linux Applications

In R2023a, the Embedded Coder Support Package for Linux Applications has added the following enhancements:

- You can deploy DDS Blockset models.
- You can deploy models configured for external mode simulation.

For more information, see External Mode Simulation of Deployed Applications.

Calibration file customization

Starting in R2023a, Embedded Coder allows you to merge multiple A2L files to a model by using the coder.asap2.merge function. For more information, see Merge ASAP2 Files.

You can also add, delete, modify, find, filter, and fetch record layouts by using the ASAP2 programming interface. For more information, see coder.asap2.RecordLayout.

TLC function FULLFILE for full path of the file

Starting in R2023a, you can use the Target Language Compiler (TLC) function `FULLFILE` to find a full path of the file. The TLC function `FULLFILE` accepts the folder or file names and returns the full file specification. Using this function improves the efficiency of the TLC code when compared to `FEVAL` function calls to MATLAB function `fullfile`.

For more information, see Target Language Compiler Directives.

Support of coder.asap2.export API for DDS Blockset Models

Starting in R2023a, the `coder.aspa2.export` function can be used to generate an A2L file for DDS Blockset models.

Code Descriptor API service interface enhancements

Starting in R2023a, you can use the code descriptor API to programmatically retrieve information about measurement service interfaces, parameter tuning service interfaces, and parameter argument tuning service interfaces. For information about specific code descriptor classes for each service interface, see coder.descriptor.MeasurementServiceInterface, coder.descriptor.ParameterTuningServiceInterface, and coder.descriptor.ParameterArgumentTuningServiceInterface. For general information about using the code descriptor API with service interfaces, see Get Metadata About Service Interface.

⚠Functionality being removed or changed

Embedded Coder Support Package for Texas Instruments C2000 Processors has transitioned into C2000 Microcontroller Blockset

Starting in R2023a, Embedded Coder Support Package for Texas Instruments® C2000 Processors has transitioned into the C2000 Microcontroller Blockset. Existing support package functionality is available in the new product.

Embedded Coder Support Package for STMicroelectronics STM32 Processors: Support for STM32L4xx, STM32L5xx, and STM32WBxx-based boards

- Use the Embedded Coder Support Package for STMicroelectronics STM32 Processors to generate and build code using an STM32CubeMX project file for STM32L4xx, STM32L5xx, and STM32WBxx-based boards.
- The new STM32-based boards supports the following peripherals ADC, PWM, GPIO, Hardware Interrupt, Timer, Encoder, I2C and UART/USART blocks for model base design using Embedded Coder Support Package for STMicroelectronics STM32 Processors
- For the new STM32-based boards, you can use the support package to:
  - Monitor signals & tune parameters in the external mode.
  - Run processor-in-loop (PIL) simulation in the serial communication mode.

For more additional improvements, see release notes in Archived R2023a Embedded Coder Support Package for STMicroelectronics STM32 Processors documentation.

Embedded Coder Support Package for STMicroelectronics STM32 Processors: Support for CAN Read, CAN Write, FDCAN Read, FDCAN Write, SPI Receive, SPI Transmit, SPI Controller Transfer, and Digital to Analog Converter blocks

Starting R2023a, you can use the following blocks with STM32 processor-based boards.

- Use the CAN Read and CAN Write blocks to read and write data from a CAN Bus in STM32F4xx, STM32F7xx, and STM32L4xx processor-based boards.
- Use the FDCAN Read and FDCAN Write blocks to read and write data from a CAN FD Bus in STM32G4xx, STM32H7xx, and STM32L5xx processor-based boards.
- Use the SPI Transmit, SPI Receive, and SPI Controller Transfer blocks to write and read data from an SPI peripheral device in STM32F4xx, STM32F7xx, STM32G4xx, and STM32H7xx processor-based boards.
- Use the Digital to Analog Converter (DAC) block to receive the digital value and convert it to the equivalent analog voltage on a specified channel in STM32G4xx processor-based board.
- Use the updated Analog to Digital Converter (ADC) block to support buffering of group conversions when the block outputs N-by-M data.
- Use the Protocol Encoder and Protocol Decoder blocks to encode and decode the input data on the communication protocol.

For more additional improvements, see release notes in Archived R2023a Embedded Coder Support Package for STMicroelectronics STM32 Processors documentation.

Embedded Coder Support Package for STMicroelectronics STM32 Processors: Support for I2S Audio Out, I2S Mic In, TCP Receive, TCP Send, UDP Receive, and UDP Send blocks

Starting R2023a, you can use the following blocks with STM32F4xx-based boards.

- Use the I2S Audio Out block to send an audio stream and I2S Mic In block to read an audio stream on STM32F4xx processor-based boards.
- Use the updated UDP Receive and UDP Send blocks for stateless and connectionless data transmission on STM32F4xx processor-based boards.
- Use the updated TCP Receive and TCP Send blocks for data transmission from a remote host or other target hardware over a TCP/IP network on STM32F4xx processor-based boards.

For more additional improvements, see release notes in Archived R2023a Embedded Coder Support Package for STMicroelectronics STM32 Processors documentation.

Performance

Code Profile Analyzer

To analyze execution-time and stack usage profiles produced by software-in-the-loop (SIL), processor-in-the-loop (PIL), or XCP-based external mode simulations, use the Code Profile Analyzer. Use the new app to perform these tasks:

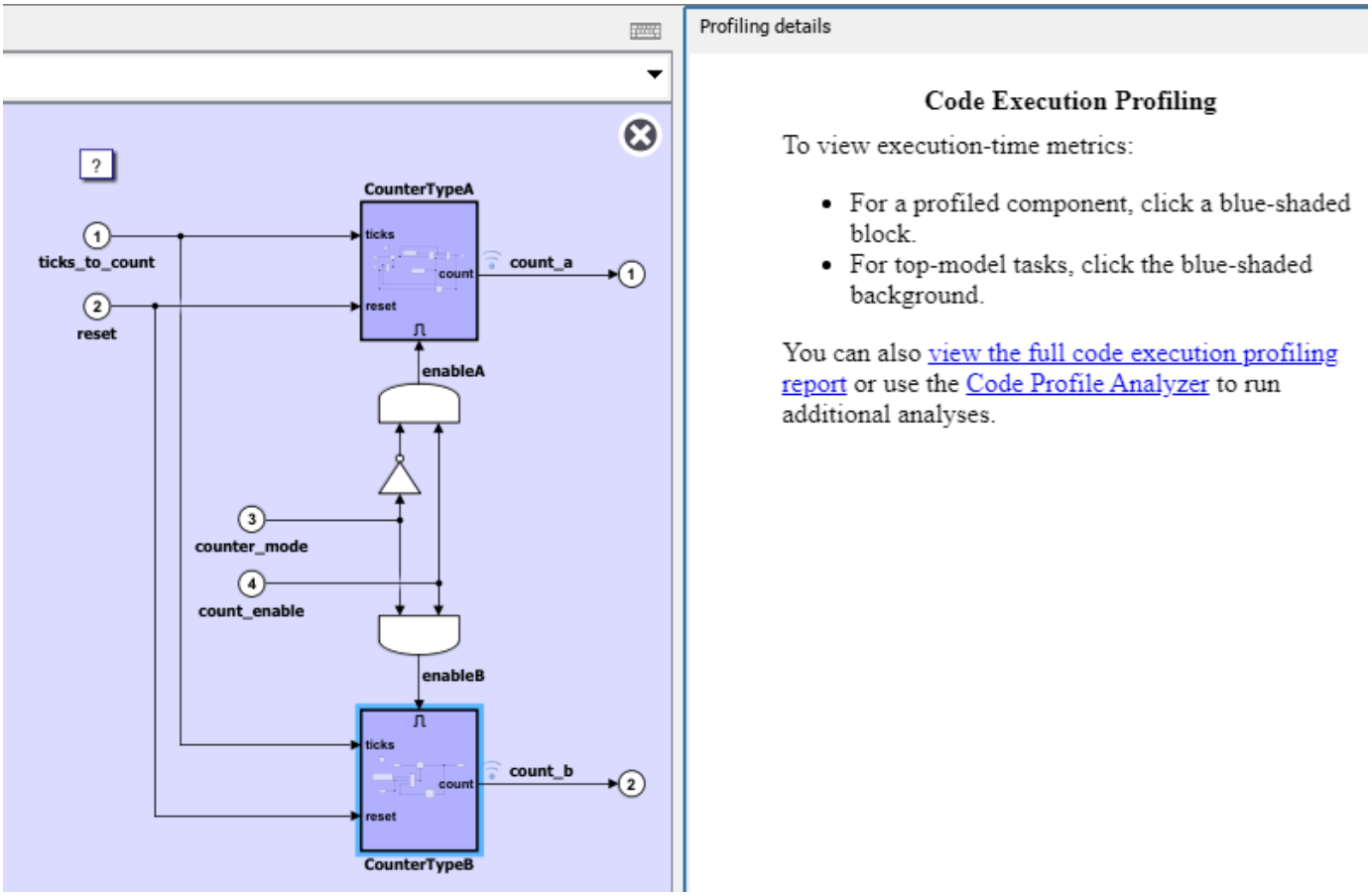
- Analyze profiling results interactively.
- Investigate the function call-stack for the most demanding simulation step.
- Compare results from different simulations.

For more information, see:

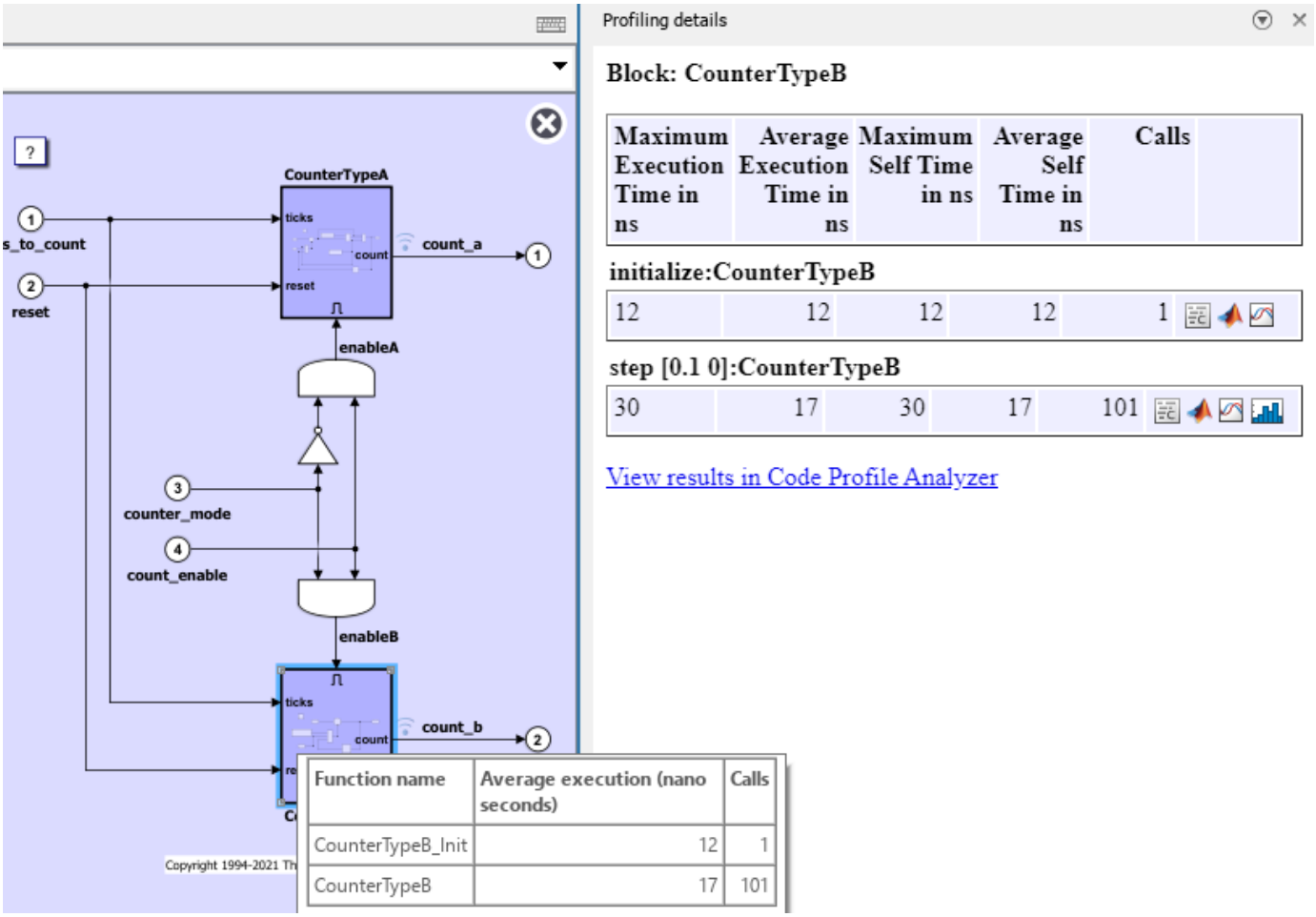
- Code Profile Analyzer
- View and Compare Code Execution Times
- Stack Usage Profiling for Code Generated from Simulink Models

Display of profiling results in Simulink Editor

If you enable execution-time profiling for a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation, at the end of the simulation, the Simulink Editor displays the **Profiling details** panel. The panel provides links to the code execution profiling report and the Code Profile Analyzer.



To view execution-time metrics for a profiled component, place the cursor over a blue-shaded block. Alternatively, click the block to display execution-time metrics in the panel.



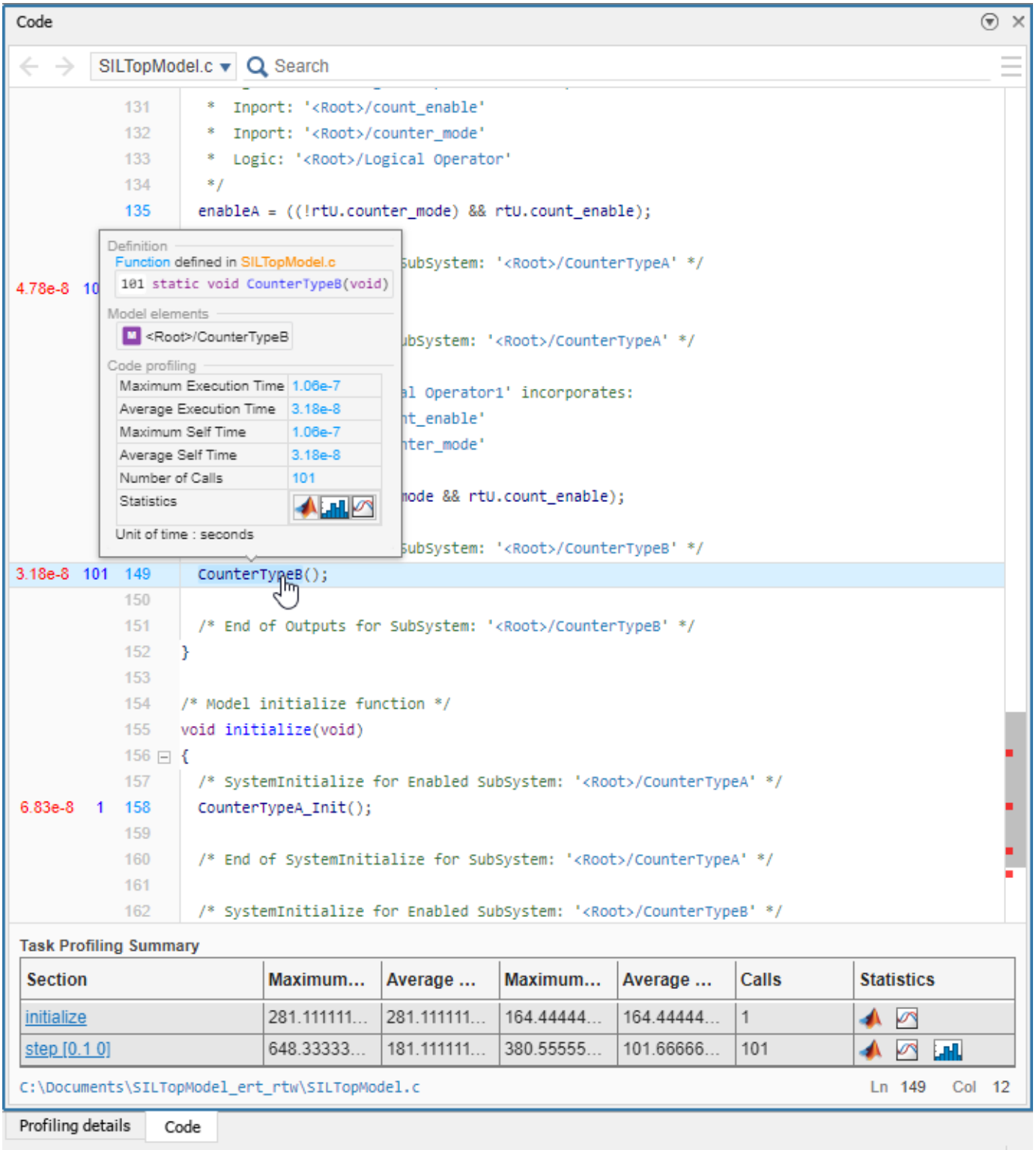
In previous releases, the software displays the metrics in a separate window.

For more information, see View and Compare Code Execution Times.

View additional code execution profiling results in Code view

In R2023a, when you run your model in the SIL/PIL Manager app, you can view additional code execution profiling information in the **Code** view. You can:

- View a summary of task profiling information below the code.
- Access links to detailed statistics from the profiling tooltip when you point to a function call in the **Code** view.
- View detailed profiling information in the model window by clicking the **Profiling details** tab.



For more information, see View and Compare Code Execution Times.

Stack usage profiles for child functions of tasks

To determine the size of stack memory that is required to run generated code, you can run a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation that produces a stack usage profile. You can use stack usage profiles to observe the effect of compiler optimization and data input. In previous releases, the simulation produces stack usage metrics for generated tasks but not child functions of the tasks. Now, the simulation also evaluates stack memory usage for child functions of tasks. For more information, see Stack Usage Profiling for Code Generated from Simulink Models.

Memory allocation for execution-time profiling with XCP external mode simulations

To optimize memory allocation on target hardware with limited resources and reduce communication channel bandwidth usage, you can configure profiling memory allocation in XCP external mode simulations by using these Simulink parameters:

- `CodeProfilingXCPMaxMemory` — Specify the maximum amount of memory to use for code profiling.
- `CodeProfilingMaxBufferSize` — Specify the maximum size of the buffer that the simulation uses to upload profiling data from the target hardware to your development computer.

For more information, see Specify Memory Allocation for Code Execution Profiling.

SIMD code for integer operations for ARM Cortex-A

In R2023a, when you generate code for ARM Cortex-A devices by using the ARM Cortex-A code replacement library, the generated code contains SIMD instructions for these integer operations:

- Addition, multiplication, and subtraction
- Bitwise
- Shift left for signed integers
- Load, store, and broadcast

For more information, see Optimize Code for ARM Cortex -A Processors (Embedded Coder Support Package for ARM Cortex-A Processors).

Generate SIMD code for FIR Interpolation and FIR Decimation blocks

In R2023a, if you have DSP System Toolbox™, you can generate SIMD code for the FIR Interpolation and FIR Decimation blocks. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. To generate SIMD code from the blocks, set these model configuration parameters:

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

Your model must meet the requirements for code generation described in FIR Decimation (DSP System Toolbox), FIR Interpolation (DSP System Toolbox), and Generate SIMD Code from Simulink Blocks.

Improved C code for models using parfor-loops

To preserve the maximum stack limit, the code generator might promote local variables to global variables. Prior to R2023a, if the code generator promoted the local variable of the `parfor` loop to a global variable due to the stack limitation, the code generator then produced a normal `for` loop instead of a `parfor` loop and could not generate OpenMP (Open Multiprocessing) code.

Starting in R2023a, you can generate OpenMP code even if the code generator promotes the local variables of the `parfor` loop to a global variables. For more information, see Parallel for-Loops (parfor) in Generated Code.

Data store buffer reuse for referenced models irrespective of inplace specifications

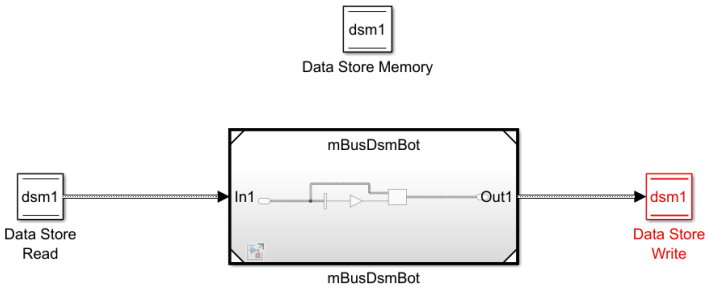
Starting in R2023a, you can generate optimized code for a model containing a referenced model whose input values are read from and output values are written to the same top-level data store.

If you select these model configuration parameters:

- **Reuse buffers for Data Store Read and Data Store Write blocks** for the top model
- **Reuse output buffers of Model blocks** for the top and referenced models

The code generator analyzes referenced model contents to determine if it is possible to reuse the data store buffers for holding referenced model input and output values. If reuse is possible, the code generator reuses the data store buffers, which improves RAM efficiency.

Consider, the model `mBusDsmTop` containing the referenced model `mBusDsmBot`. The referenced model does not have function prototype control specifications to use the same input and output variable. A Data Store Read block reads the referenced mode input values from the top-level data store `dsm1`. The output values of the referenced model are written to `dsm1` by a Data Store Write block.



In R2022b, the code generator produced this code:

```
void mBusDsmTop_step(RT_MODEL *const rtM)
{
  D_Work *rtDWork = rtM->dwork;
  BusType1 rtb_mBusDsmBot;
  mBusDsmBot(&rtDWork->dsm1, &rtb_mBusDsmBot);
  rtDWork->dsm1 = rtb_mBusDsmBot;
}
```

The code contained an unnecessary temporary variable `rtb_mBusDsmBot` and data copy.

In R2023a, the code generator produces this code:

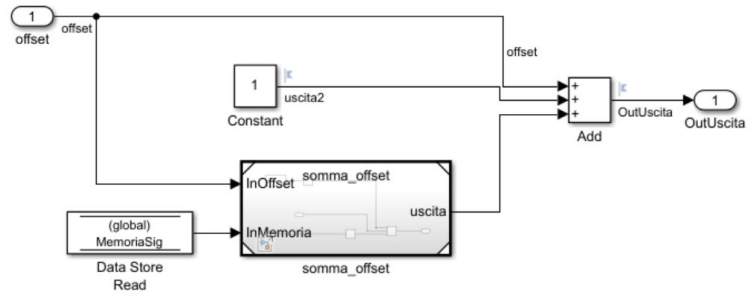
```
void mBusDsmTop_step(RT_MODEL *const rtM)
{
  D_Work *rtDWork = rtM->dwork;
  mBusDsmBot(&rtDWork->dsm1, &rtDWork->dsm1);
}
```

The code does not contain the unnecessary temporary variable `rtb_mBusDsmBot` and data copy. The code reuses the data store memory buffer for both input and output values, which improves RAM efficiency. For more information, see Data Copy Reduction for Data Store Read and Data Store Write Blocks.

Enhanced global data store reuse in the presence of referenced models

Before R2023a, for models that used data stores with the `ExportToFile` storage class, the generated code contained redundant data copies when the top model read input or wrote output values of the referenced models from or to the global data store. If no read or write operation happens for the global data store inside the referenced models, the code generator can now eliminate redundant data copies when the top model reads input or writes output values of the referenced models. Eliminating the extra data copies reduces RAM and ROM consumption and improves execution speed. To enable this optimization, select the **Reuse buffers for Data Store Read and Data Store Write blocks** model configuration parameter.

Consider the model `mTopMdlRef` with a Data Store Read block that reads data from a data store, which uses the `ExportToFile` storage class. The read data is input to the referenced model `somma_offset`. Inside the referenced model, no read or write operation happens for the global data store.



In R2022b, the code generator produced this code:

```
/* Model step function */
void mTopMdlRef_step(void)
{
  uint32_T rtb_uscita;

  /* DataStoreRead: '<Root>/Data Store Read' */
  memcpy(&mTopMdlRef_B.InMemoria[0], &MemoriaSig[0], 100U * sizeof(uint32_T));

  /* ModelReference: '<Root>/somma_offset' incorporates:
```

```
   *  Inport: '<Root>/offset'
   */
  somma_offset_step(&offset, &mTopMdlRef_B.InMemoria[0], &rtb_uscita);

  /* Sum: '<Root>/Add' incorporates:
   *  Inport: '<Root>/offset'
   */
  OutUscita = (offset + mTopMdlRef_B.uscita2) + rtb_uscita;
}
```

The code contained an unnecessary temporary variable `mTopMdlRef_B.InMemoria` and data copy.

In R2023a, the code generator produces this code:

```
/* Model step function */
void mTopMdlRef_step(void)
{
  uint32_T rtb_uscita;

  /* ModelReference: '<Root>/somma_offset' incorporates:
   *  DataStoreRead: '<Root>/Data Store Read'
   *  Inport: '<Root>/offset'
   */
  somma_offset_step(&offset, &MemoriaSig[0], &rtb_uscita);

  /* Sum: '<Root>/Add' incorporates:
   *  Inport: '<Root>/offset'
   */
  OutUscita = (offset + mTopMdlRef_B.uscita2) + rtb_uscita;
}
```

The code does not contain the unnecessary temporary variable `mTopMdlRef_B.InMemoria` and data copy. It reuses the data store memory buffer `MemoriaSig` for passing the reference model input values to `somma_offset_step` function, which improves RAM efficiency. For more information, see Data Copy Reduction for Data Store Read and Data Store Write Blocks.

Change to reuse referenced model buffers model configuration parameter settings

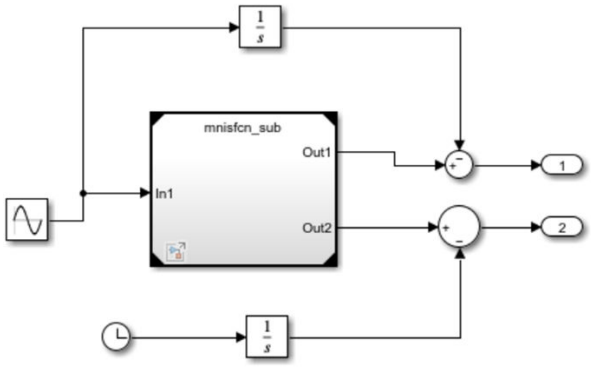
Starting in R2023a, when you configure models to use maximum optimization **Level**, by default, model configuration parameter **Reuse output buffers of Model blocks** is selected.

If you use different settings of the **Reuse output buffers of Model blocks** parameter for the top model and the model referenced by a Model block, the code generator no longer issues build errors. However, the optimization is enabled only when the top model and the referenced model have the parameter selected. Reusing referenced model output buffers conserves RAM usage and improves the execution efficiency of the generated code. For more information, see Reduce Memory Usage for Models Containing Referenced Models.

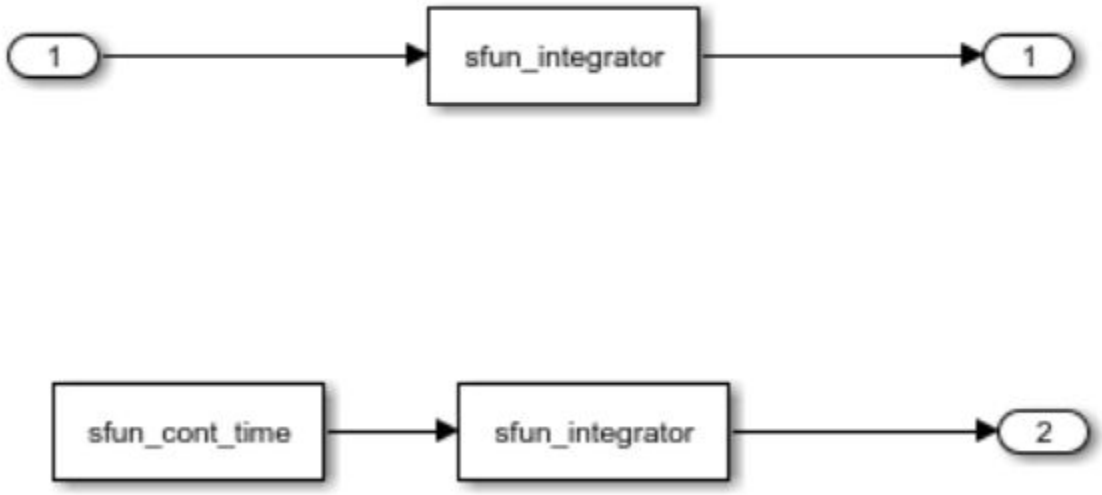
Data copy reduction for referenced model buffers reuse optimization

Before R2023a, for some modeling patterns containing referenced models, the **Reuse output buffers of Model blocks** optimization produced extra data copies in the generated code. The code generator now optimizes the code by eliminating the unnecessary data copies, which improves RAM consumption.

Consider, the model `mnisfcn_top` contains a Model block to reference the model `mnisfcn_sub`.



The referenced model imports external C code through S-Function blocks.



The top and the model referenced by the Model block are configured to use the maximum optimization **Level** to balance RAM usage and code speed, which automatically enables the **Reuse output buffers of Model blocks** parameter for models.

In R2022b, the code generator produced this code:

```
/* ModelReference: '<Root>/Model' */
  mnisfcn_sub(&mnisfcn_top_B.SineWave, &rtb_Integrator1, &rtb_Model_o2);

  /* Outport: '<Root>/Out1' incorporates:
   *  Sum: '<Root>/Sum'
   */
  mnisfcn_top_Y.Out1 = rtb_Integrator1 - rtb_Integrator;

  /* Integrator: '<Root>/Integrator1' */
  rtb_Integrator1 = mnisfcn_top_X.Integrator1_CSTATE;
```

```
/* Outport: '<Root>/Out2' incorporates:
 *  Sum: '<Root>/Sum1'
 */
mnisfcn_top_Y.Out2 = rtb_Model_o2 - rtb_Integrator1;
```

The code reused the `rtb_Integrator1` buffer to hold the referenced model output `out1`, but the code contained an unnecessary data copy to `rtb_Integrator1`.

In R2023a, the code generator produces this code:

```
/* ModelReference: '<Root>/Model' */
mnisfcn_sub(&mnisfcn_top_B.SineWave, &rtb_Integrator1, &rtb_Model_o2);

/* Outport: '<Root>/Out1' incorporates:
 *  Sum: '<Root>/Sum'
 */
mnisfcn_top_Y.Out1 = rtb_Integrator1 - rtb_Integrator;

/* Outport: '<Root>/Out2' incorporates:
 *  Integrator: '<Root>/Integrator1'
 *  Sum: '<Root>/Sum1'
 */
mnisfcn_top_Y.Out2 = rtb_Model_o2 - mnisfcn_top_X.Integrator1_CSTATE;
```

The code eliminates the unnecessary data copy to the `rtb_Integrator1` by expression folding the computation. For more information, see Reduce Memory Usage for Models Containing Referenced Models.

Improve code efficiency by using code efficiency tools and techniques

Embedded Coder documentation now contains a new topic Optimize Generated Code Using Code Efficiency Tools and Techniques that describes how using different code efficiency tools and techniques, you can improve the efficiency of the generated code. If the efficiency of the code generated from your model does not meet your requirements, review the tips and techniques discussed in the topic and choose an approach for your model.

Verification

Debugging for PIL simulations

Provide debugging for processor-in-the-loop (PIL) simulations by following these steps:

1. When you set up PIL connectivity, specify a debugger by using `target.ExecutionService` and `target.DebugExecutionTool` objects.
2. In the Configuration Parameters dialog box, select the **Enable source-level debugging for SIL or PIL** check box. Or, from the Command Window, set `SILPILDebugging` to `'on'`

In previous releases, debugging is available only for software-in-the-loop (SIL) simulations.

MATLAB scripts still support `SILDebugging`, the previous command-line parameter.

For more information, see:

- Support PIL Debugging
- `DebugExecutionTool` Template
- Debug Generated Code During SIL or PIL Simulation

Initialization of model workspace parameters for Model block SIL/PIL simulations

You can run Model block software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations with generated code that has uninitialized or undefined model workspace parameters. In previous releases, the simulations produce errors or numerical mismatches between results from the model and the generated code.

For Model block SIL or PIL simulations, you can use:

- Model workspace parameters with imported storage classes
- Exported model workspace parameters with no data initialization
- Model workspace parameters that map to AUTOSAR shared parameters

You can also tune model workspace parameter values between simulations, including when fast restart is enabled.

For more information, see:

- Configure and Run SIL Simulation
- Run Automated Verification, Model Simulation, or SIL/PIL Simulation
- General SIL and PIL Limitations

Specify whether to open Code View automatically

You can use a new Simulink preference to specify whether or not the Code View window opens automatically after you build a model. In the Simulink Preferences dialog box, select **Editor**, then set

the preference **Open Code View window after building a model**. For more information, see Simulink Preferences.

Check bug reports for issues and fixes

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

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

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

R2022b

Code Generation from MATLAB Code

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

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

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

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

MATLAB Code	R2022b Generated Code	R2022a Generated Code

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

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

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

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

- Dead code: `MISRA C:2012 Rule 2.2` (Polyspace Bug Finder), `AUTOSAR C++14 Rule M0-1-9` (Polyspace Bug Finder)
- Lexical conventions: `AUTOSAR C++14 Rule M2-10-1` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A2-10-6` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A2-3-1` (Polyspace Bug Finder)

- Identifiers: `MISRA C:2012 Rule 5.6` (Polyspace Bug Finder)
- Compilation directive: `MISRA C:2012 Rule 4.1` (Polyspace Bug Finder), `MISRA C:2012 Dir 4.12` (Polyspace Bug Finder)
- Side effects and expressions: `MISRA C:2012 Rule 13.2` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A5-0-1` (Polyspace Bug Finder), `AUTOSAR C++14 Rule M5-0-8` (Polyspace Bug Finder)
- Standard libraries: `MISRA C:2012 Rule 21.3` (Polyspace Bug Finder)
- Other restrictions: `MISRA C:2012 Dir 2.1` (Polyspace Bug Finder), `AUTOSAR C++14 Rule M0-1-3` (Polyspace Bug Finder), `AUTOSAR C++14 Rule M17-0-2` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A12-0-1` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A0-1-3` (Polyspace Bug Finder), `AUTOSAR C++14 Rule A18-0-1` (Polyspace Bug Finder)

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

Model Architecture and Design

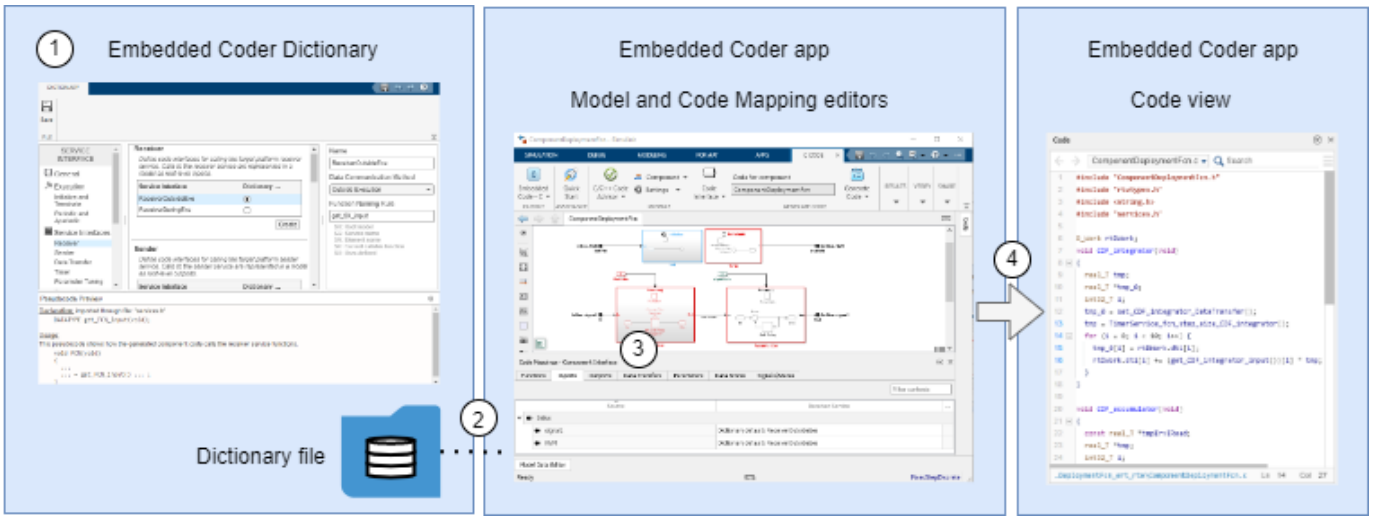
Deploy models as components that include comprehensive service interface support

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

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

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

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



The complete set of features enables you to:

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

guidelines and Model Advisor checks for component deployment using a service interface configuration” on page 8-9.

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

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

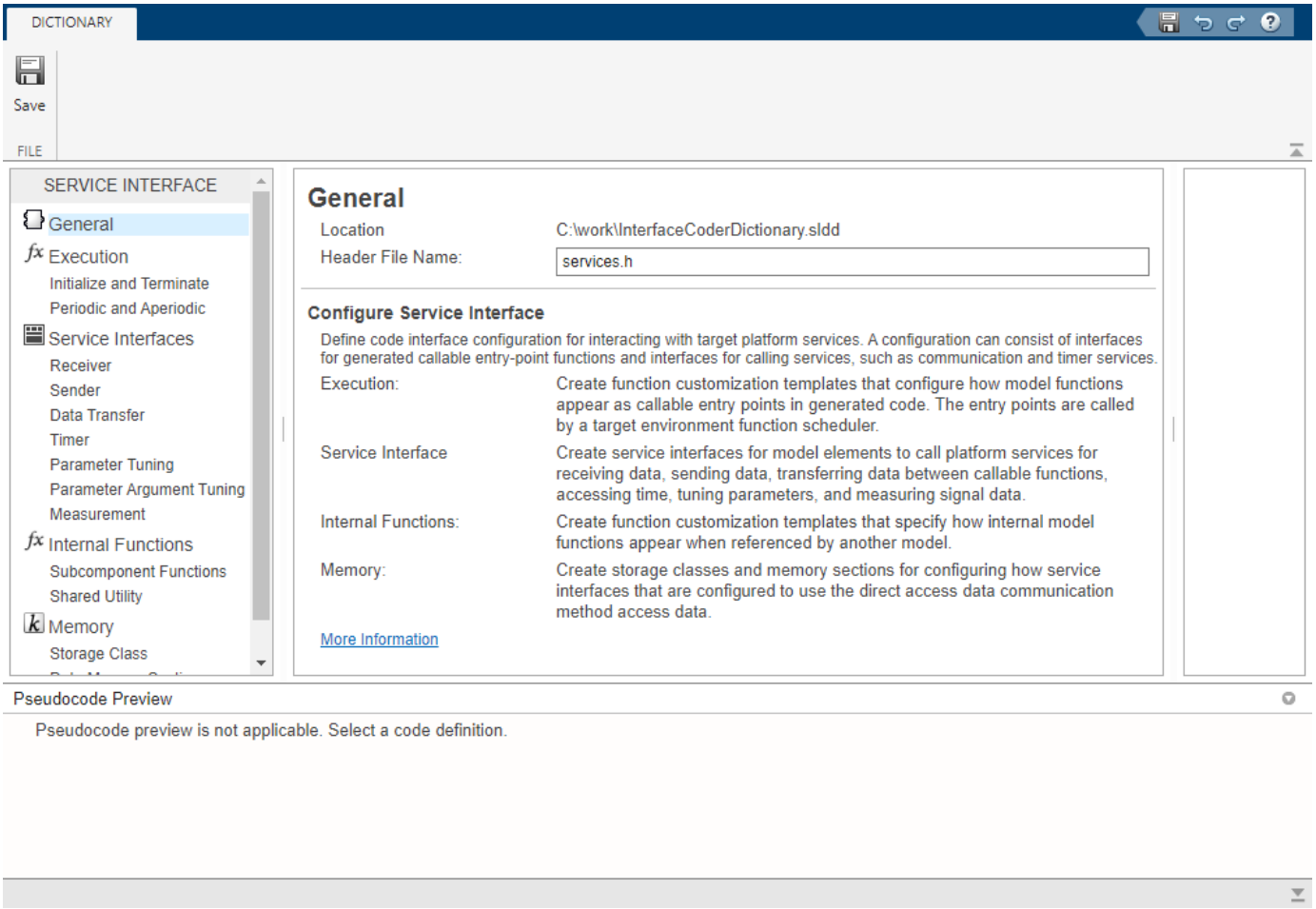
For background and high-level workflow information, see Embedded Coder Fundamentals.

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

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

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

- Service interface configuration — The configuration contains service interfaces, storage classes, function customization templates, and memory sections. The new service interface configuration enables you to define comprehensive service interfaces to generate code that interacts with services provided by specific target platforms. For more information, see “Deploy models as components that include comprehensive service interface support” on page 8-4.
- Data interface configuration — The configuration contains storage classes, function customization templates, and memory sections. If you created an Embedded Coder Dictionary in an earlier release, when you open the dictionary in R2022b, the dictionary contains a data interface configuration with the existing code interface definitions.



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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Component service interface support for callable entry-point functions

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

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Component service interface support for target platform data transfer service

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

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Component service interface support for target platform timer service

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

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

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

For information on how to set up a service interface configuration with Embedded Coder that includes comprehensive service support for a target platform, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

Code Interface Configuration and Integration

Map model elements to service interfaces

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

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

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

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

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

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

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

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

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

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

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

Functionality being removed or changed

Model parameters and parameter arguments returned separately by find function

Behavior change

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

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

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

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

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

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

Embedded Coder Dictionary refreshes when loading model from earlier release

Behavior change

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

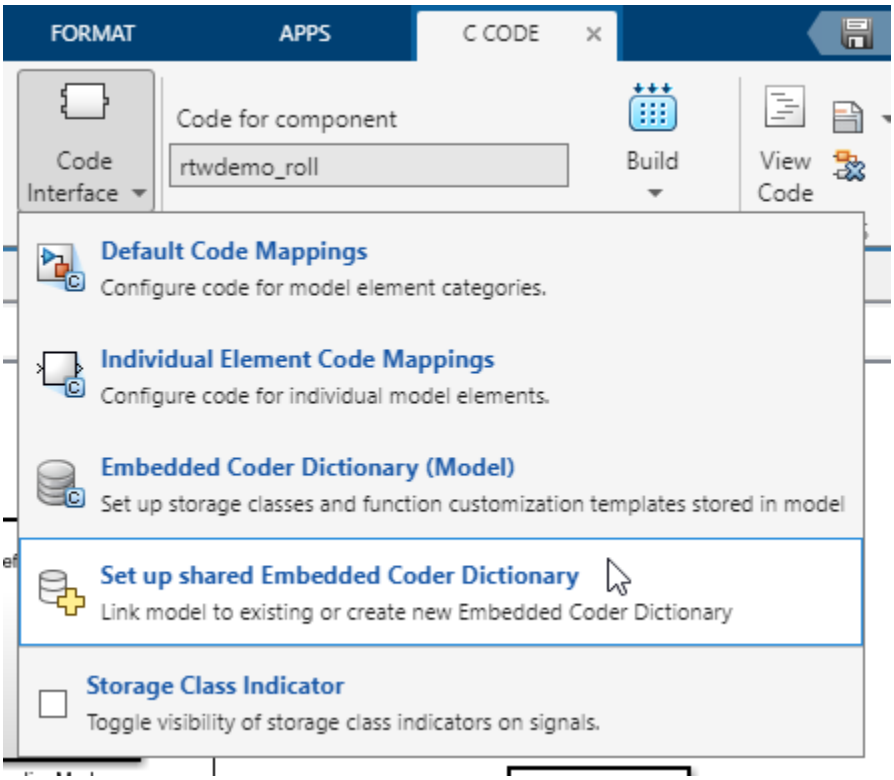
earlier release. To prevent the dictionary refresh, resave the model, or the Simulink data dictionary that contains the Embedded Coder Dictionary, in the current release.

Code Generation

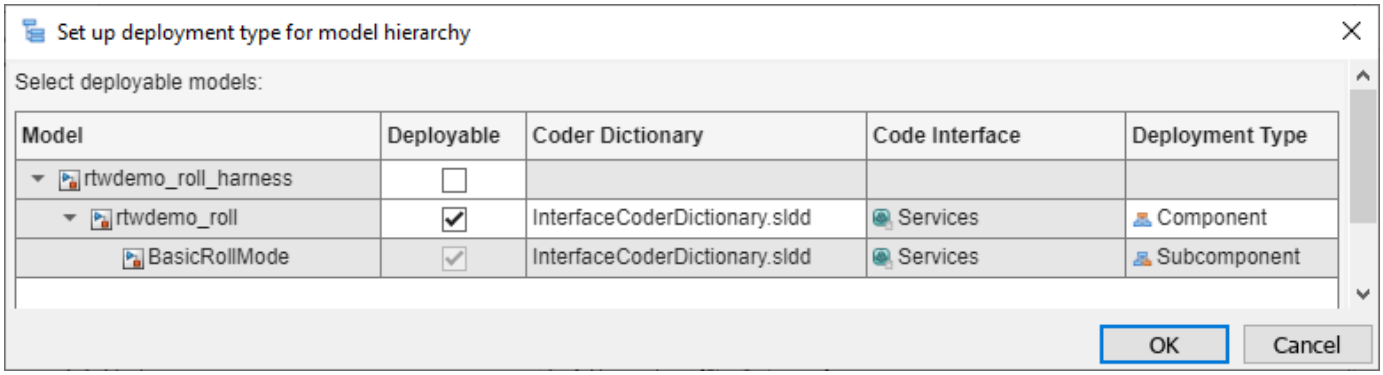
Select code interface configuration using new configuration parameter

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

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



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



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

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

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

For more information, see Select Code Generation Output for Target Platform Deployment and Configure C Code Deployment Types for Model Hierarchy. For more information about deployment component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

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

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Generated C++11 example main program simplified

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

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

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

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

    case 1 :
     rtwdemo_cppclass_Obj.EngineEntrypoint1();
     break;

    case 2 :
     rtwdemo_cppclass_Obj.EngineEntrypoint2();
     break;

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

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

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

Include requirement comments in the generated code

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

Files and folders for target platform services

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

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

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

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

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

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

For more information, see:

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

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Code interface report for service interfaces

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

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

For more information, see Analyze Generated Service Code Interface. For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Generate code for Reusable custom storage classes with symbolic dimension inputs

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

For more information, see Implement Dimension Variants for Array Sizes in Generated Code.

New $X naming rule token

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

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

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

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

For more information about naming rule tokens, see Identifier Format Control. For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Example models attached to examples and renamed

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

R2022a model name	New model name	Example
`rtwdemo_condinput`	`ConditionalInput`	Use Conditional Input Branch Execution
`rtwdemo_deadpathElim`	`DeadPathElimination`	Eliminate Dead Code Paths in Generated Code
`rtwdemo_foreachreuse`	`ForEachReuse`	Generate Reusable Code from For Each Subsystems
`rtwdemo_col_dlut3d_selplane`	`ColumnDLUT3DSelectPlane`	Direct Lookup Table Algorithm for Row-Major Array Layout
`rtwdemo_col_dlut3d_selvector`	`ColumnDLUT3DSelectVector`	Direct Lookup Table Algorithm for Row-Major Array Layout
`rtwdemo_row_dlut3d_selplane`	`RowDLUT3DSelectPlane`	Direct Lookup Table Algorithm for Row-Major Array Layout
`rtwdemo_row_dlut3d_selvector`	`RowDLUT3DSelectVector`	Direct Lookup Table Algorithm for Row-Major Array Layout
`rtwdemo_mdlreftop`	`TopModelCode`	File Packaging for Models (Code and Data)
`rtwdemo_mdlrefbot`	`ReferenceModelCode`	File Packaging for Models (Code and Data)
`rtwdemo_row_interpalg`	`RowInterpolationAlgorithm`	Interpolation Algorithm for Row-Major Array Layout
`rtwdemo_row_lut2d`	`RowLUT2D`	Interpolation Algorithm for Row-Major Array Layout
`rtwdemo_udt`	`UserDefinedDataTypes`	Define Abstract Numeric Types and Rename Types

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

New Simulink Model Advisor check for numeric efficiency

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

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

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

⚠Code replacement validation detects ambiguous overflow and rounding modes

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

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

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

Compatibility Considerations

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

Deployment

Retrieve metadata about service interface by using code descriptor programming interface

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

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

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

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

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

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

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

For more information, see Get Metadata About Service Interface.

For more information about deploying component models that are configured with a service code interface, see “Deploy models as components that include comprehensive service interface support” on page 8-4.

Target Language Compiler search functions for regular expressions

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

TLC Built-In Functions

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

For more information, see Target Language Compiler Directives.

Introducing Embedded Coder Support Package for Linux Applications

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

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

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

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

Introducing Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers

Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers is available from release R2022b. You can use the support package to design real-time applications for Infineon 32-bit TriCore AURIX TC4x family of microcontrollers using Simulink and generate processor optimized code that you can compile and execute on AURIX TC4x family of microcontrollers.

The first release of this support package includes:

- Support for Digital Port Read, Digital Port Write, Hardware Interrupt, Encoder, PWM, QSPI, and TMADC blocks.
- Support for Hardware Mapping and Peripheral Configurations.
- Examples that demonstrate the capabilities of the support package:
  - Getting Started with Embedded Coder Support Package for Infineon AURIX TC4x Microcontrollers

- Code Verification and Validation with PIL
- Field-Oriented Control of BLDC with Encoder Using Infineon AURIX Microcontrollers

Calibration File Customization

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

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

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

Also, the new enhancements allow you to

- Insert custom code fragments in different sections of the ASAP2 file.
- Modify the `Name` and `Comments` for the project and module sections.
- Provide address extension for the ECU address to measurements, characteristics, and axis points.
- Insert functions hierarchy by adding function as subfunction in another function.

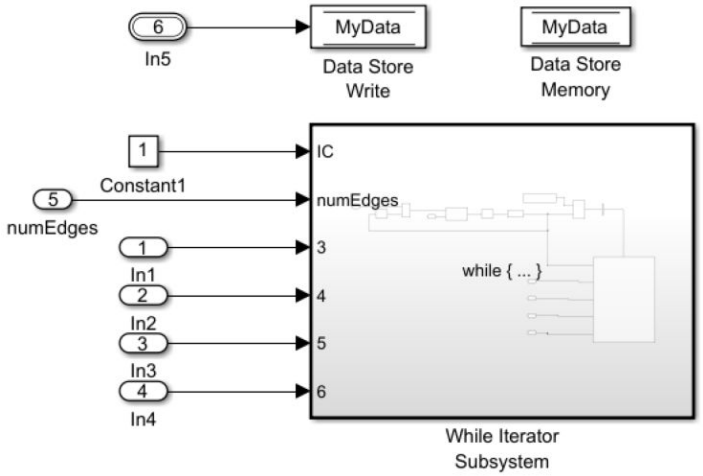
For more information, see Customize Generated ASAP2 File.

Performance

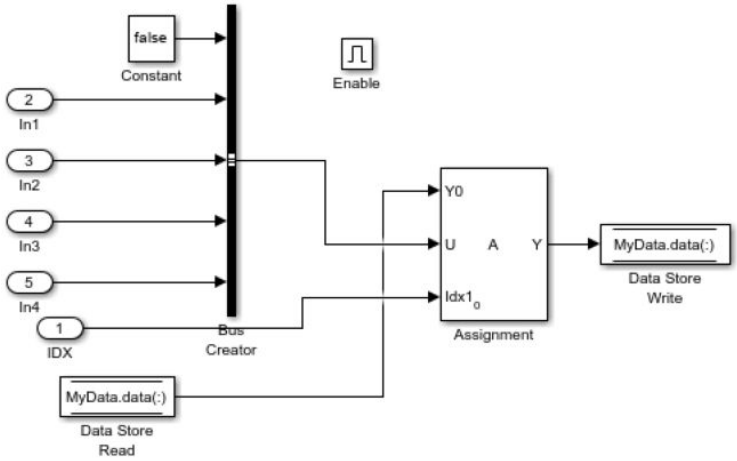
Data Store Memory block reuse in reusable subsystems inside While Iterator subsystems

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

For example, the `mDSMReuse` model has a While Iterator subsystem and a Data Store Memory block.



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



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

```
if (rtu_Enable) {
      if ((*rtd_WhileIterator_IterationMarker) < 2ULL) {
```

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

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

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

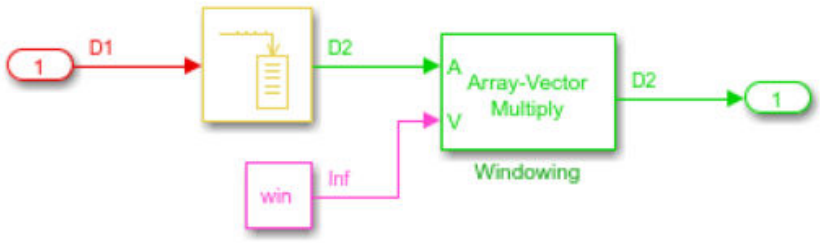
```
  if (rtu_Enable) {
      if ((*rtd_WhileIterator_IterationMarker) < 2ULL) {
         *rtd_WhileIterator_IterationMarker = 2U;
      }
      rtd_MyData->data[rtu_IDX].flag = false;
      rtd_MyData->data[rtu_IDX].a1 = rtu_In1;
      rtd_MyData->data[rtu_IDX].a2 = rtu_In2;
      rtd_MyData->data[rtu_IDX].a3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

The code does not contain the local variable `localB` and the data copies, which improves RAM efficiency of generated code. For more information, see Enable and Reuse Local Block Outputs in Generated Code.

Removed redundant multirate block output buffers

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

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



In R2022a, the code generator produced this code:

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

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

In R2022b, the code generator produces this code:

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

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

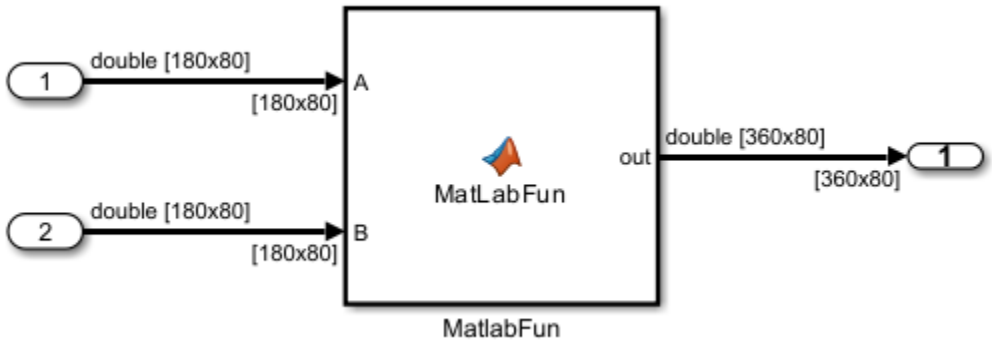
Buffer reuse optimization for referenced models

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

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

In R2022b, you can generate optimized code containing loop interchange and distribution. These loop transformations increase the number of cache hits and improve the code execution time. The optimizations apply to code generation targets for which the cache information is available to the code generator. To increase the availability of cache information to the code generation target,

specify the target hardware information by using the configuration parameters Device vendor and Device type.



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

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

sizeRow=90;
sizeCol=80;

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

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

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

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

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

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

```
      rtDW.B_m[B_tmp - 1] = (real_T)((i + 2) * (j + 2)) + rtDW.B_m[B_tmp - 2];
      for (B_tmp = 0; B_tmp < 79; B_tmp++) {
        int32_T A_tmp;
        A_tmp = (B_tmp + 1) * 180;
        rtDW.A[(B_tmp_tmp + A_tmp) - 1] = (rtDW.A[A_tmp + i] + ((real_T)i + 2.0))
          + ((real_T)j + 2.0);
      }
    }
  }

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

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

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

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

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

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

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

  /* Outport: '<Root>/Out1' */
  for (j = 0; j < 80; j++) {
    for (i = 0; i < 180; i++) {
```

```
      B_tmp = 180 * j + i;
      A_tmp = 360 * j + i;
      rtY.Out1[A_tmp] = rtDW.A[B_tmp];
      rtY.Out1[A_tmp + 180] = rtDW.B_m[B_tmp];
    }
  }

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

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

Generate SIMD code for Discrete FIR Filter block

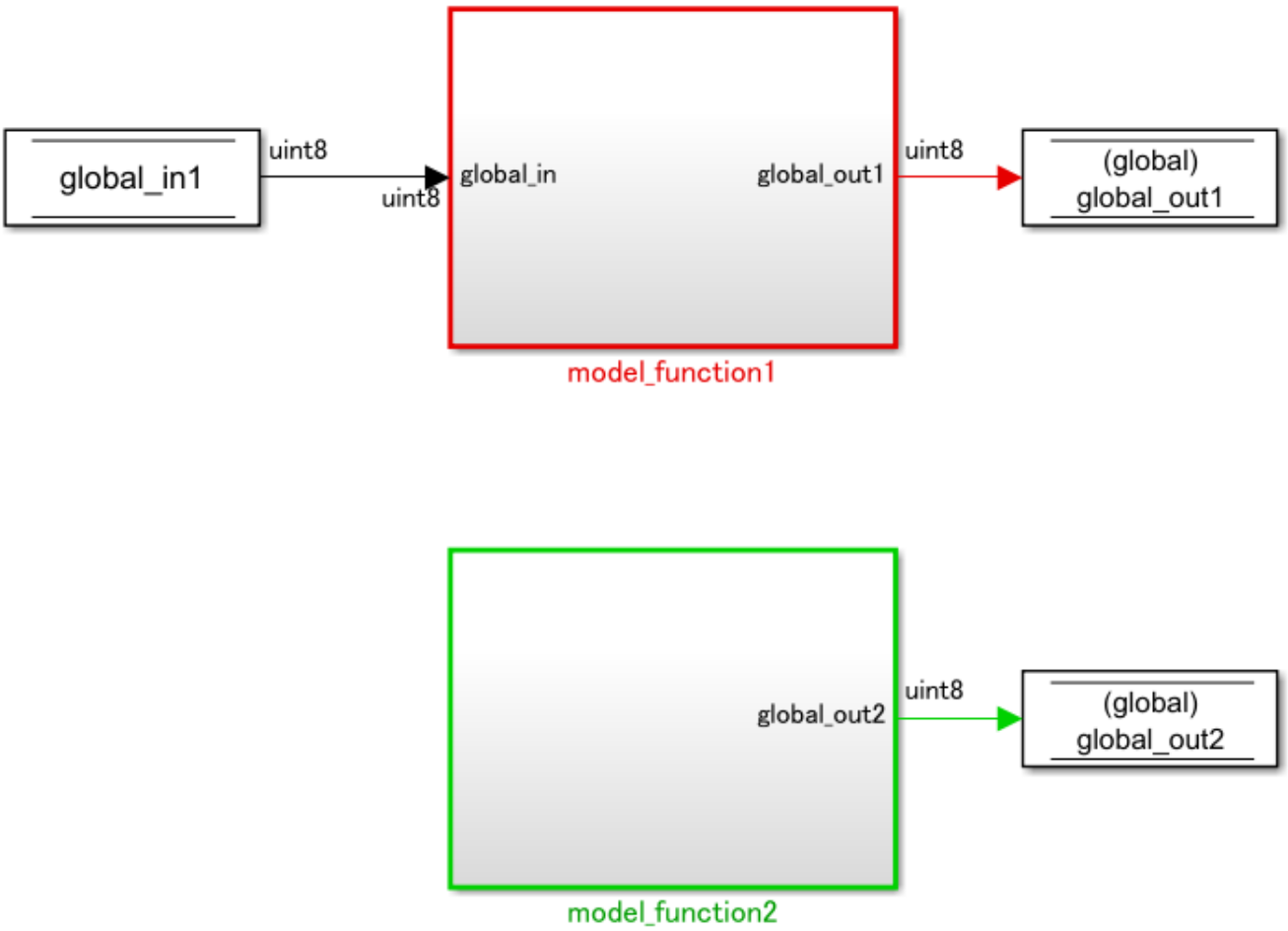
In R2022b, if you have DSP System Toolbox, you can generate SIMD code for the Discrete FIR Filter block. For computationally intensive operations on supported blocks, SIMD intrinsics can significantly improve the performance of the generated code on Intel platforms. To generate SIMD code from the Discrete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

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

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

Improved function argument generation eliminates extra global variable assignment

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



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

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

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

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

This code enhancement occurs when:

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

SIMD code for bitwise and shift operations

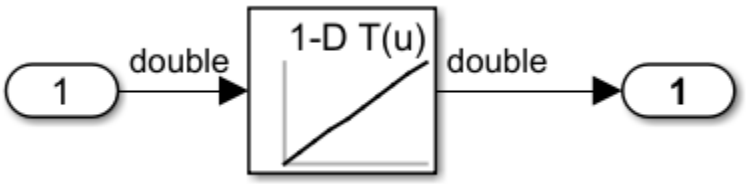
In R2022b, you can generate SIMD code for bitwise operations and shift operations. When you select an instruction set by using the Leverage target hardware instruction set extensions parameter, the generated code includes the associated instructions for these bitwise operations and shift operations:

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

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

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

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



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

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

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

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

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

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

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

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

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

Code execution profiling for models that use GRT system target files

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

For more information, see Configure Code Execution Profiling.

Task scheduling visualization with XCP external mode simulations

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

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

Optimized bandwidth usage during XCP external mode profiling

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

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

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

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

For more information, see Display Absolute Time.

Verification

SIL or PIL block workflow

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

Reusable subsystems with input signals that map to const variables

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

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

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

Check bug reports for issues and fixes

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

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

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

R2022a

Code Generation from MATLAB Code

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

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

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

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

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

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

MATLAB Code	Generated Code with `cfg.PreserveUnusedStructFields = false;` (default)	Generated Code with `cfg.PreserveUnusedStructFields = true;`

`classdef myClass`
`  properties`
`  a`
`  b`
`  c`
`  end`
 	
`  methods`
`  function obj = myClass(x)`
`  coder.inline('never')`
`  obj.a = x;`
`  obj.b = x + 1;`
`  obj.c = x + 2;`
`  end`
`  end`
`end`
 	
`function y = myAdd(n)`
`  o = myClass(n);`
`  y = o.a + o.b;`
`end`
	`% unused property ‘c’ is removed`
	
	`class myClass {`
	`public:`
	`  void init(double x);`
	`  double a;`
	`  double b;`
	`};`
		`% unused property ‘c’ is present`
		
		`class myClass {`
		`public:`
		`  void init(double x);`
		`  double a;`
		`  double b;`
		`  double c;`
		`};`

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

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

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

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

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

Stack usage profiling for code generated from MATLAB code

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

Identification of performance bottlenecks in generated code

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

Model Architecture and Design

Symbolic dimension inputs for Squeeze block

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

For more information, see Implement Dimension Variants for Array Sizes in Generated Code.

Embedded Coder Dictionary interface improvements

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

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

Code Interface Configuration and Integration

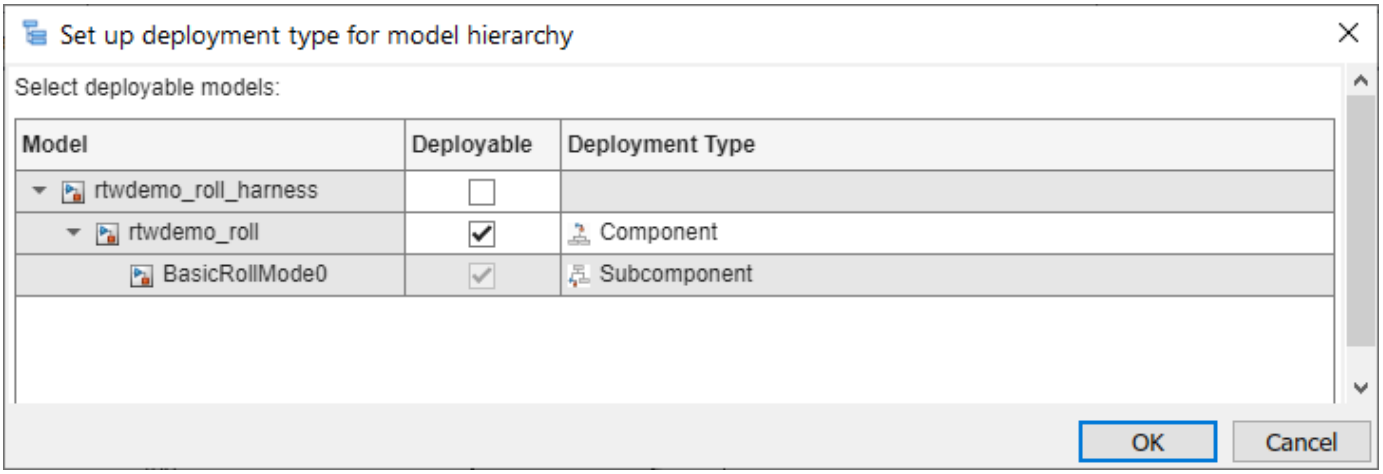
Control code interface generated for models by specifying deployment types

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

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

You can specify these deployment types for your models:

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



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

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

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

For more information, see Configure Deployment Types for Model Hierarchy.

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

Nested namespace support

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

For more information, see Interactively Configure C++ Interface and Programmatically Configure C++ Interface.

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

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

Compatibility Considerations

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

Calibration file customization

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

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

You can further customize the ASAP2 file as follows:

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

For more information, see Customize Generated ASAP2 File.

Memory section mapping for grouped entry-point functions

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

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

```
/* Internal Data Grouped For Same Function */

FuncInternalData0 rtFuncInternalData0; /* '<Root>/Unit Delay' */
```

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

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

```
/* Internal Data Grouped For Same Function */

/* This memory is of moderate speed and cost */
#pragma MEDIUM_MEM(rtFuncInternalData0)

FuncInternalData0 rtFuncInternalData0; /* '<Root>/Unit Delay' */
```

For more information, see Control Data and Function Placement in Memory by Inserting Pragmas.

Code Generation

Regular expression token decorators to modify certain tokens

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

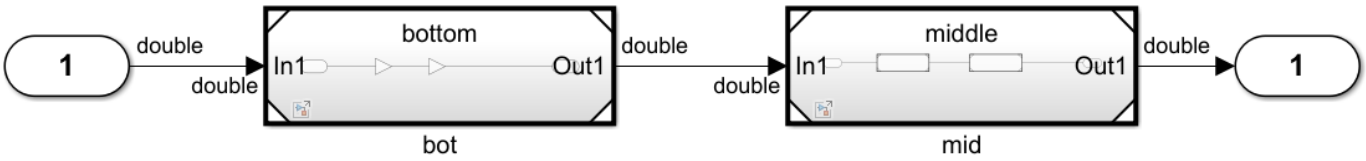
For example, this identifier naming rule takes the root model name `$R` and replaces instances of `a` with `b`: `$R["a/b"]`.

For more information, see Identifier Format Control.

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

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

Consider the following model, `top`.



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

Before R2022a, the code generator produced this code:

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

  {
    {
      3.0,
      3.0
    },

    {
      3.0,
      3.0
    }
  }                                    /* instance parameters for '<Root>/mid' */
};
```

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

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

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

    /*
     * top/bot
     *   bottom:Q
     */
    3.0
  },

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

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

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

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

For more information about generating comments, see Configure Code Comments.

New parentheses level for MISRA standard compliance and code readability

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

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

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

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

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

Before R2022a	After R2022a

`if((int32_T)reproMissingSuffix_U.In1 < 100)`
`/* Outport: '<Root>/Out1' */`
`reproMissingSuffix_U.Out1 = 100U;`
	`if( reproMissingSuffix_U.In1 < 100U )`
	`/* Outport: '<Root>/Out1' */`
	`reproMissingSuffix_U.Out1 = 100U;`

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

Changes to initialization

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

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

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

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

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

In R2021b, the code generator produced this code:

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

In R2022a, the code generator produces this code:

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

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

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

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

In R2021b, the code generator produced this code:

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

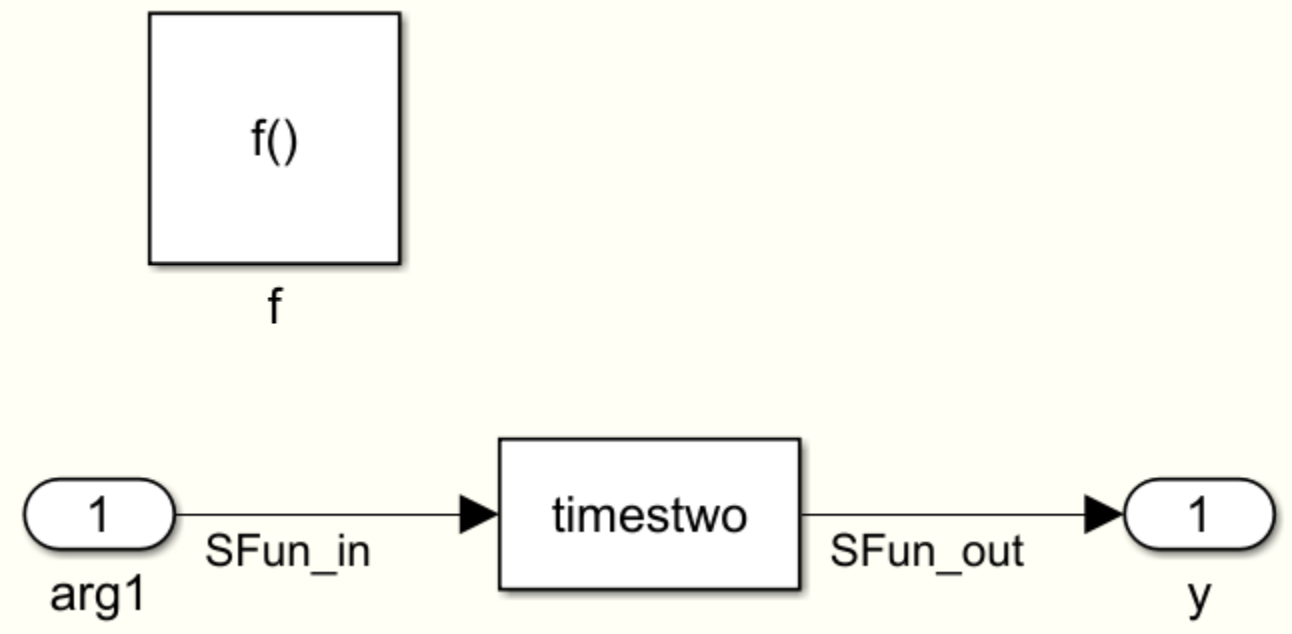
In R2022a, the code generator produces this code:

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

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

Removed redundant S-function output buffer

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



Before R2022a, the code generator produced this code:

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

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

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

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

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

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

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

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

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

C++ Code Generation for client-server interfaces

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

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

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

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

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

CustomSymbolStrUtil parameter available for C++ and AUTOSAR code generation

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

⚠Functionality being removed or changed

Include guards required in imported header files

Behavior change

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

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

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

Deployment

TLC function STRNREP for string replacement

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

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

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

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

Texas Instruments C2000: Support for Texas Instruments F28003x processor

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

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

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

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

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

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

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

Performance

SIMD code for reduction operations

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

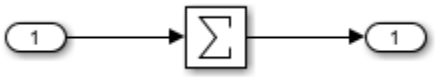
You can generate SIMD code for these blocks:

- Sum
- Product
- Minimum
- Maximum

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

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

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



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

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

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

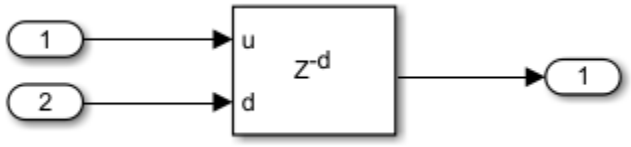
```
__m128d tmp;
real_T tmp_0[2];
int32_T i;
tmp = _mm_set1_pd(0.0);
for (i = 0; i <= 42; i += 2) {
  tmp = _mm_add_pd(tmp, _mm_loadu_pd(&sumElements_U.In1[i]));
}

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

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

Code replacement for circular buffer index for Delay blocks

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

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



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

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

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

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

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

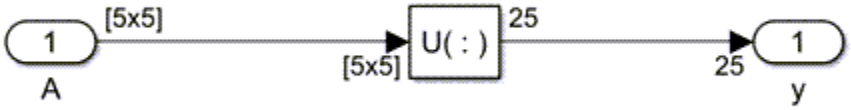
Code replacement for lookup tables by using index search algorithm parameter

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

Code generation by inlining redundant function calls

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

For example, consider the model `mRTWCGLateInlineReshapeRowmajor`.



In R2021b, the code generator produced this code:

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

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

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

In R2022a, the code generator produces this code:

```
void mRTWCGLateInlineReshapeRowmajor_step(void)
{
  int32_T i;

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

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

Stack usage profiling for code generated from Simulink models

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

Identification of performance bottlenecks in generated code

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

Code execution profiling for multiple Model blocks

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

Verification

Unit-testing atomic subsystem code in AUTOSAR software component

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

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

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

⚠Functionality being removed or changed

SIL/PIL support for BullseyeCoverage will be removed
Still runs

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

Check bug reports for issues and fixes

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

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

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

R2021b

Code Generation from MATLAB Code

Communication I/O information display during SIL or PIL execution

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

Visualization of task scheduling

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

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

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

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

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

Model Architecture and Design

Built-in storage class for multi-instance data

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

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

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

For more information, see Implement Dimension Variants for Array Sizes in Generated Code.

Code Interface Configuration and Integration

Storage class with pointer data access in Embedded Coder Dictionary

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

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

For more information, see Embedded Coder Dictionary.

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

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

For more information, see Embedded Coder Dictionary.

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

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

For more information, see Embedded Coder Dictionary.

Changes to model hierarchy requirements

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

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

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

- Inports

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

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

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

Calibration file customization

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

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

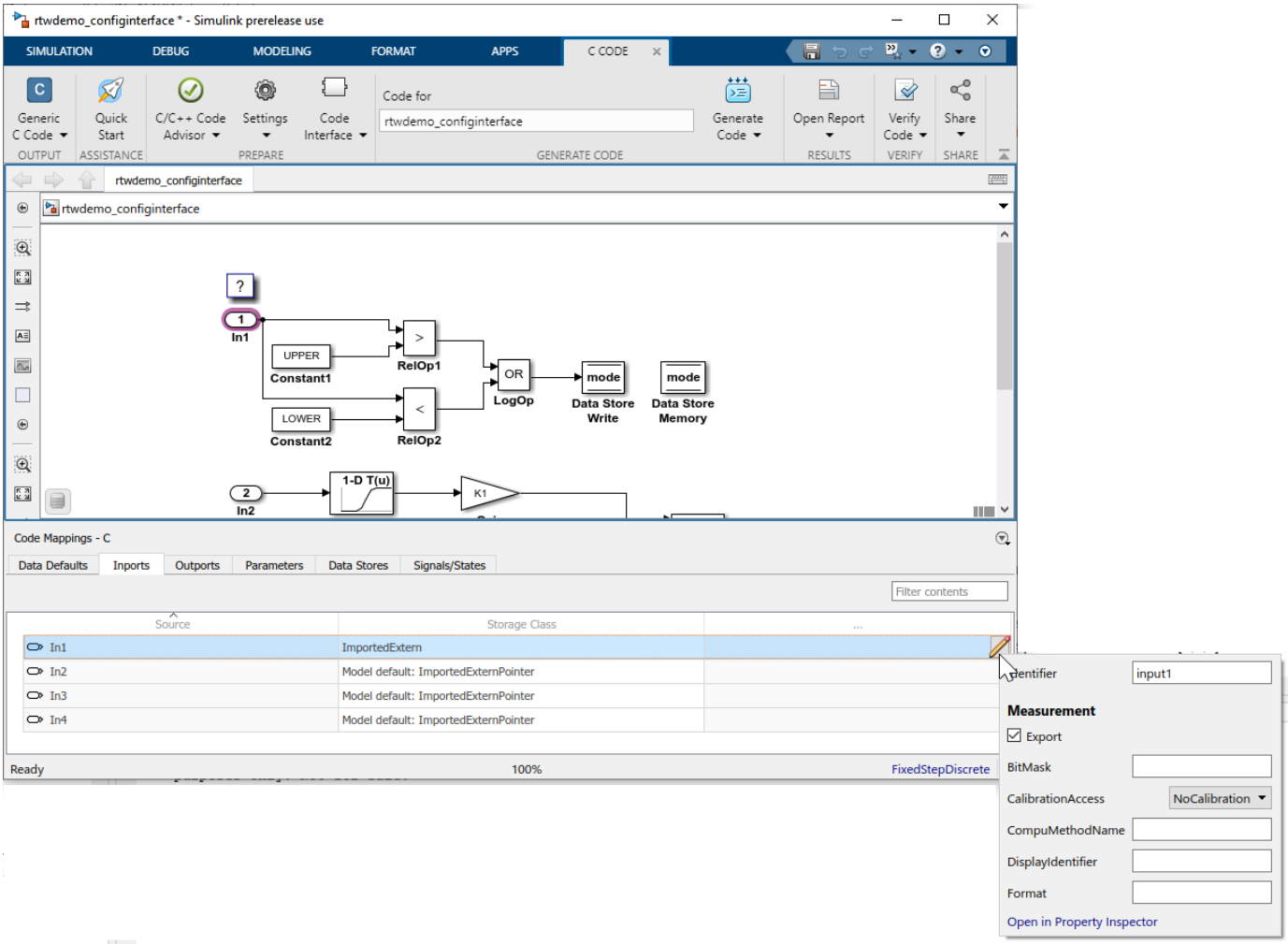
TLC code storage classes in default mapping

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

Configure additional properties from the Code Mappings editor

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

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



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

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

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

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

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

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

Reusable storage class in Code Mappings editor

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

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

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

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

For more information, see Specify Buffer Reuse for Signals in a Path.

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

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

Code Generation

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

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

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

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

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

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

Compatibility Considerations

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

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

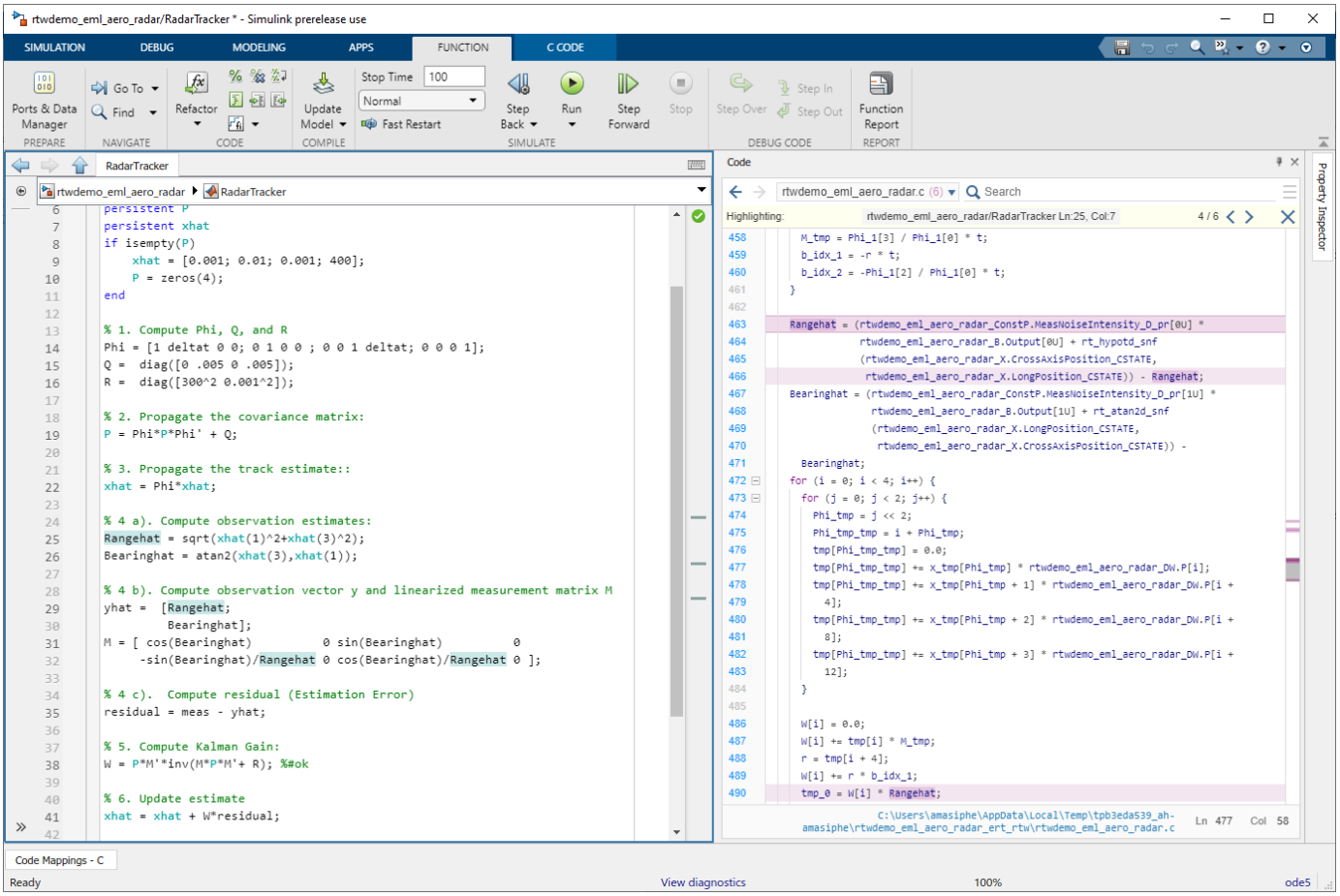
- To use the generated static `main` program on bare board target hardware (**Generate an example main program** is selected and **Target operating system** is set to `BareBoardExample`), you can update the `#include` statement in the `main` program to specify `rtmodel.h` instead of *`model`*`.h`.
-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

For more information, see Deploy Generated Standalone Executable Programs To Target Hardware.

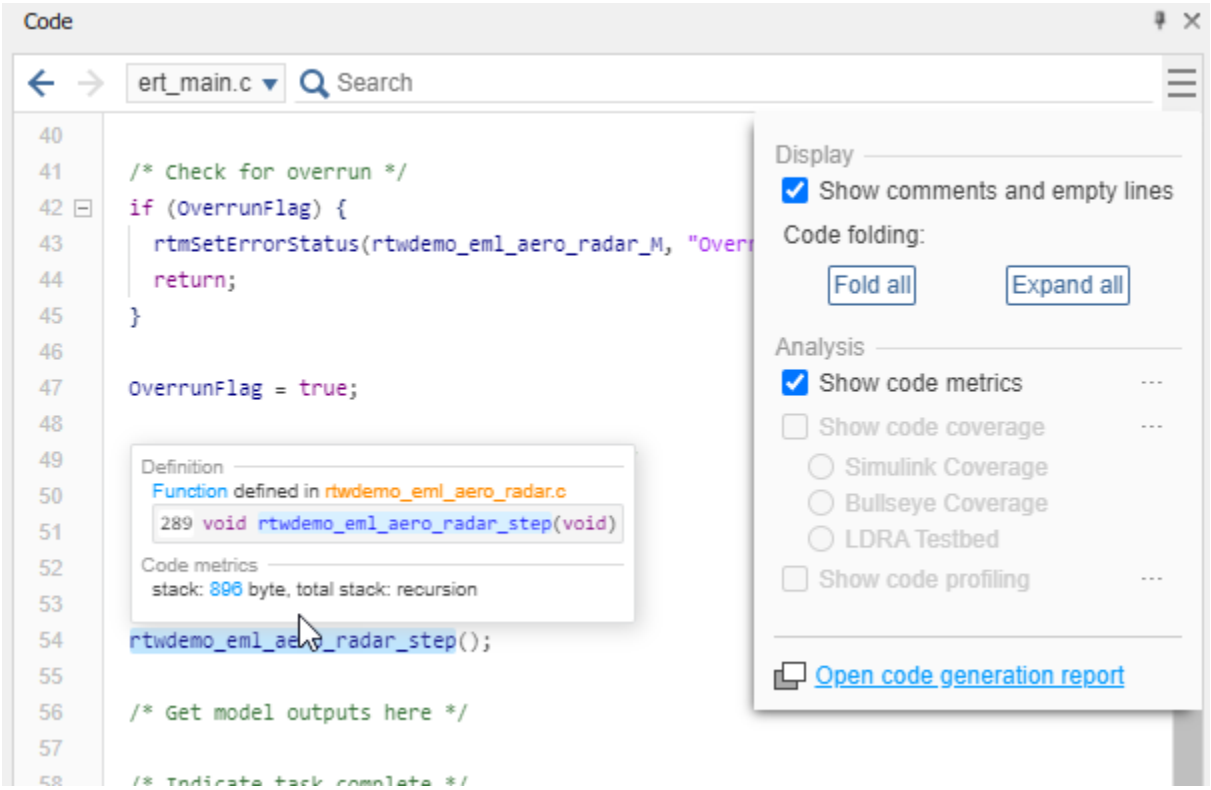
Code view for MATLAB Function block

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

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



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



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

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

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

Changes to generated C++ header files

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

const member functions for C++ class interface

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

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

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

- Calls a TLC function through fully inlined S-functions
- Is `getRTM()`
- Is *`model`*`_derivatives`

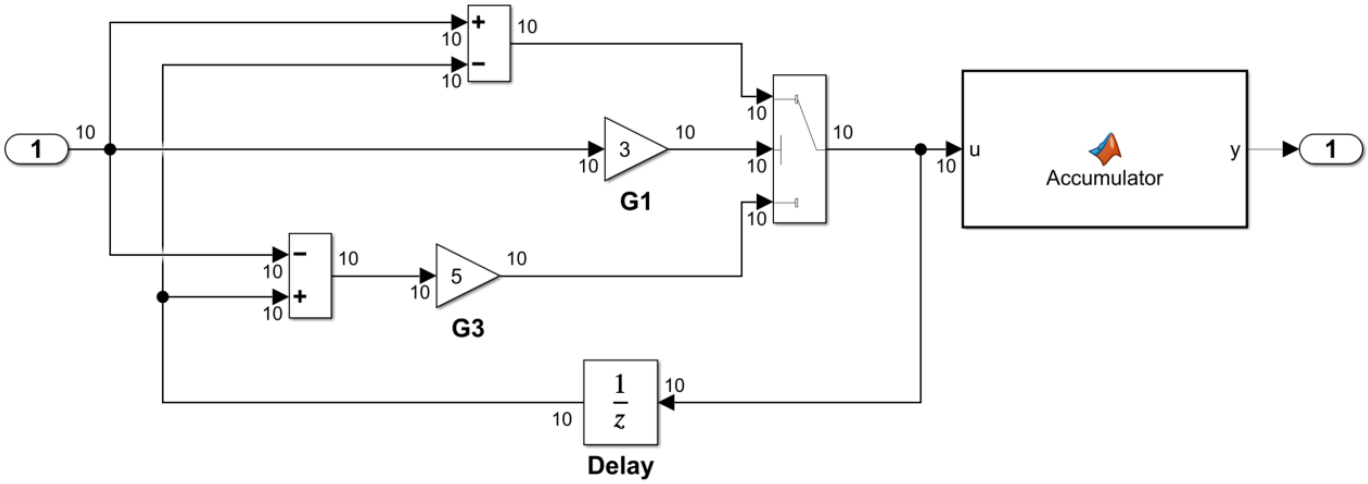
Generating member functions as `const` reduces MISRA C++ 2008 Rule 9-3-3 violations. For more information, see MISRA C++:2008 Rule 9-3-3 (Polyspace Bug Finder).

Minimized variable visibility for C++ code

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

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

Consider the model `mMinimizeVariableScopeBasic`.



In R2021a, the code generator produced this code:

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

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

In R2021b, the code generator produces this code:

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

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

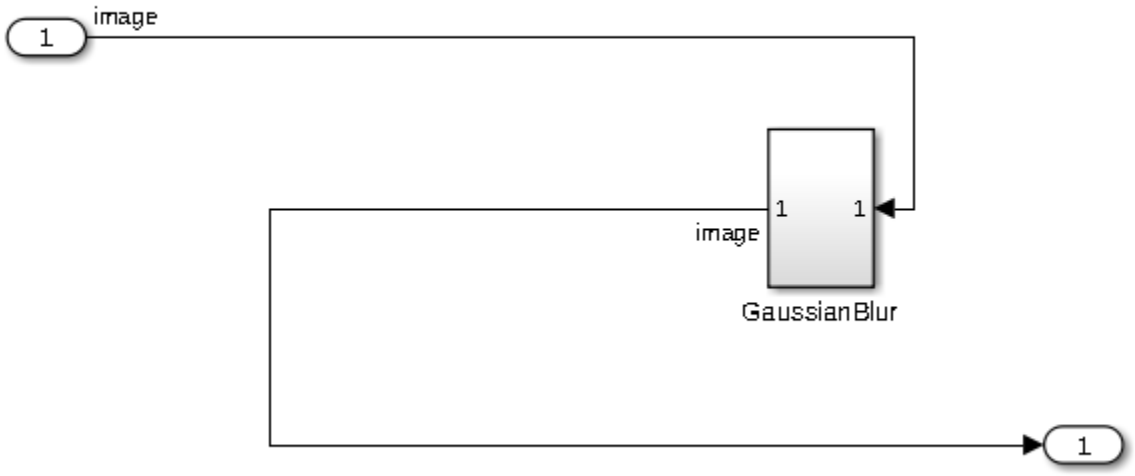
When you set the parameter value of `AdvancedOptControl` to `-SLCI` for compatibility with code inspection, the code generator does not generate minimally scoped variables.

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

Image data by using OpenCV class cv::Mat

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

For example, consider this model:



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

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

```
};

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

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


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

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

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

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

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

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

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


/* Model step function */
void mCVCodegenModelClass::step()
{

  /* S-Function Block: '<Root>/GaussianBlur' */
opencv::GaussianBlur(mCVCodegen_Y.Out1, &mCVCodegen_U.In1);

}
```

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

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

Shared types and parameters storage in same header file

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

- `Simulink.Alias`

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

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

```
#ifndef RTW_HEADER_myHeader_h_
#define RTW_HEADER_myHeader_h_
#include "rtwtypes.h"

typedef real_T myAlias;
typedef creal_T cmyAlias;

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

#endif                                 // RTW_Header_myHeader_h
```

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

Bidirectional traceability in Code view by default

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

Deployment

New TLC variable OverrideSampleERTMain for disabling generation of example main program

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

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

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

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

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

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

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

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

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

Support for STMicroelectronics STM32F4xx-based Boards

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

Performance

Generation of SIMD code by using new configuration parameter

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

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

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

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

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

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

- `Intel SSE`
- `Intel AVX`
- `Intel AVX-512`

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

Image Processing Toolbox functions enhanced with multithreading and algorithm improvements

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

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

The optimized functions that have multithreading capabilities are:

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

The functions that have algorithm improvements are:

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

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

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

The loop executed sequentially.

In R2021b, the code generator produces this code snippet:

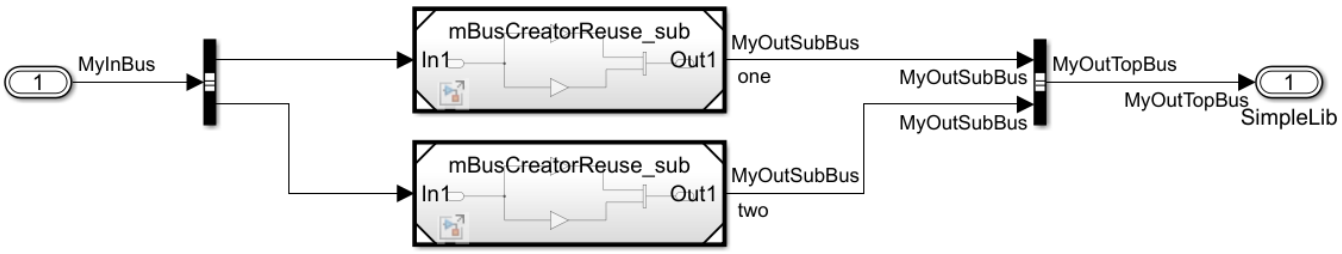
```
...
#pragma omp parallel for num_threads(omp_get_max_threads())

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

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

Reduced data copies for models that have Bus Creator blocks

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

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



In R2021a, the code generator produced this code:

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

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

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

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

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

In R2021b, the code generator generates this code:

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

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

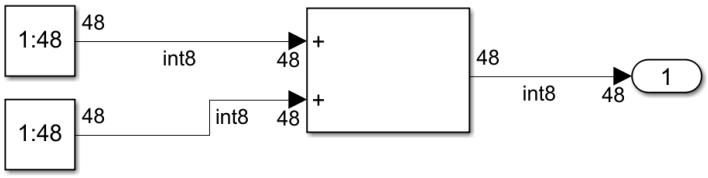
The generated code does not contain the local variables `rtb_one`, `rtb_two`, and the data copies. Now, the code generator stores the input elements of the Bus Creator block into the root output structure fields `rtY.SimpleLib.one` and `rtY.SimpleLib.two` directly.

SIMD optimization for more integer data types

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

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

Consider the model `mAddInt8`, which has an Add block.



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

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

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

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

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

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

```
  }
}
```

The loop increments by 16 because the input data type is `int8`. Incrementing by 16 instead of one occurs because the SIMD functions in the loop body process data in parallel. If the input data type is `int16`, the loop increments by 8. This optimization increases the execution speed of the generated code. For more information, see Generate SIMD Code from Simulink Blocks.

Root outport initialization code performance improvements

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

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

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

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

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

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

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

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

/* Model initialize function */
void mStringInBusCGpatterns_initialize(void)
{
   /* external outputs */
```

```
  (void) memset((void *)&mStringInBusCGpatterns_Y, 0,
                sizeof(ExtY_mStringInBusCGpatterns_T));
  mStringInBusCGpatterns_Y.Out3 = mStringInBusCGpatterns_rtZbusObj;
}
```

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

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

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

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

```
const botBus mAoSIndirectMask_rtZbotBus = {
  {
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    0.0 }
  ,                                   /* a */
  0.0                                 /* b */
} ;                                   /* botBus ground */

/* external outputs */
(void) memset(&mAoSIndirectMask_Y.Out1[0], 0,
              2U*sizeof(botBus));
```

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

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

Using the `memset` function on individual outports results in smaller object files, reducing ROM consumption. For more information, see Optimize Generated Code Using `memset` Function.

Readability improvement for root outport initialization code

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

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

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

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

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

For more information, see Customize Generated Identifier Naming Rules.

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

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

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

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

This is consistent with code generation for external inputs.

Optimize code by unrolling parallel for-loops

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

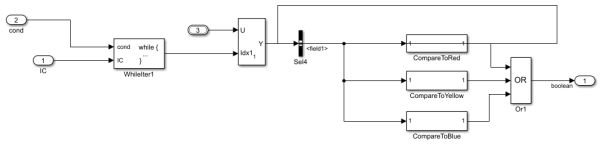
When the code generator unrolls a `parfor-loop`, it produces a copy of the loop body for each iteration. For a small number of loop iterations that perform some simple calculation, parallelization is inefficient as it introduces overheads, which includes time taken for thread creation, data synchronization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

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

Improved common subexpression elimination

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

Consider the model `mWhileBusAccess`.



In R2021a, the code generator produced this code:

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

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

In R2021b, the code generator produces this code:

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

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

Optimized SIMD code that performs fused multiply add operations

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

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

Redundant data copies elimination by reusing S-function block buffers

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

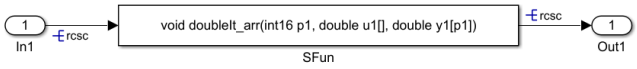
- The block has signals with buffer reuse specifications by using `Simulink.Signal` objects. For more information, see Specify Buffer Reuse for Signals in a Path.
- The block has function prototype control specifications to use the same buffer for input ports and output ports. For more information, see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

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

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

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

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



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

```
void doubleIt_arr_inplace(int dim, double* inOutVal) {
    int i = 0;
    for (i = 0;i< dim; i++){
        inOutVal[i] *= 2;
    }}
void doubleIt_arr(int dim, double* inVal, double* outVal)
{
// This check is required for Simulation Correctness because TLC code is ignored in Simulation
 if (inVal != outVal){
     int i = 0;
```

```
      for (i=0;i<dim;i++){
            outVal[i] = inVal[i];
        }}
    doubleIt_arr_inplace(dim, outVal);
}
```

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

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

In R2021a, the code generator produced this code:

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

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

In R2021b, the code generator produces this code:

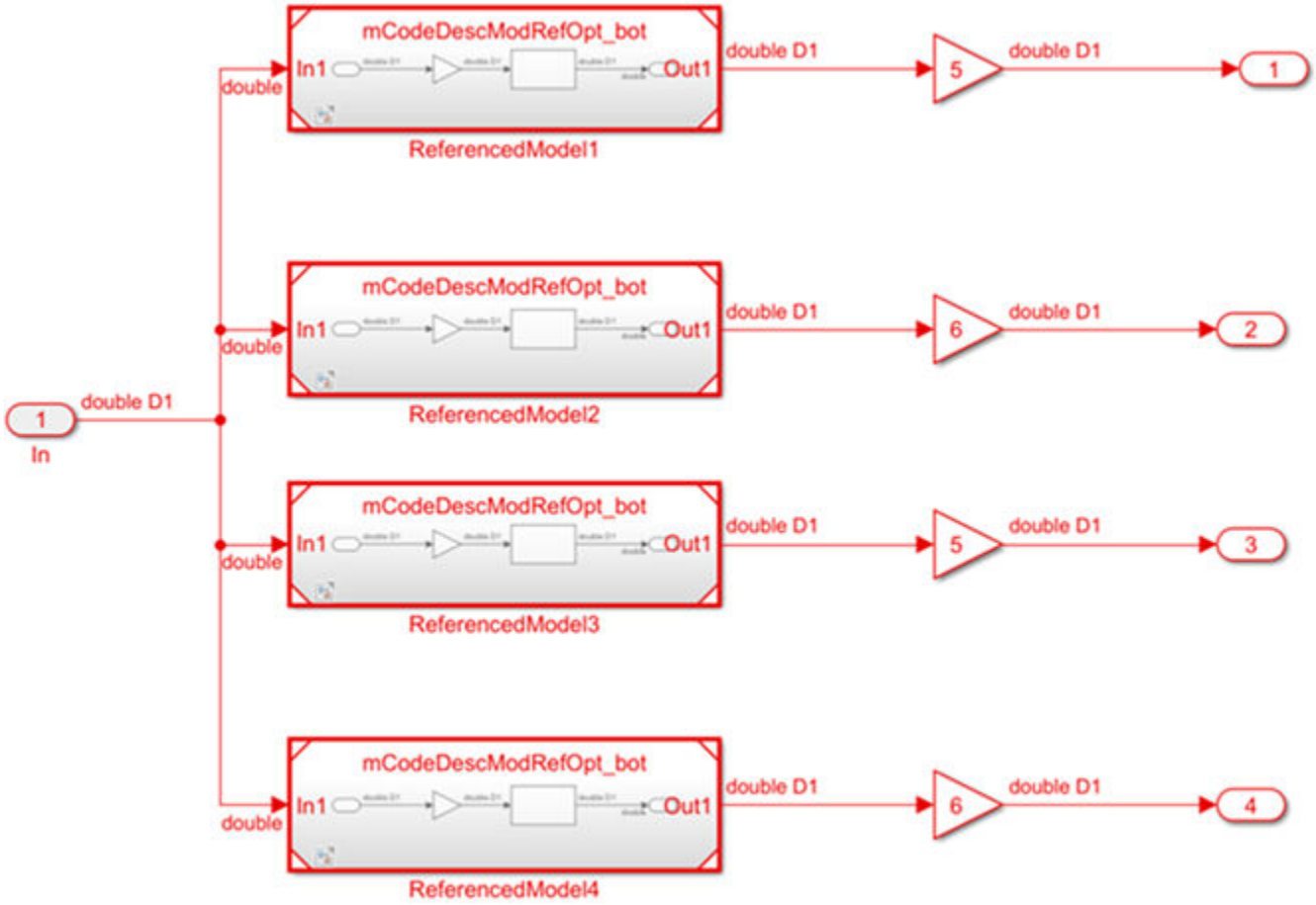
```
/* Model step function */
void mRCSC_RootInOut_step(void)
{
  /* S-Function (ex_sfun_doubleit_arr): '<Root>/SFun' */
  doubleIt_arr(2, (real_T*)(&(rcsc[0])), (&(rcsc[0])));
}
```

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

Optimized code for models containing referenced models

Starting in R2021b, the generated code has fewer local variables for modeling patterns that have referenced models. The unnecessary local variables are eliminated, which improves the efficiency of

generated code. For example, consider a model `mCodedescNodRefOpt_top` that has a referenced model `mCodedescNodRefOpt_bot`.



In R2021a, the code generator produced this code:

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

In R2021b, the code generator produced this code:

```
void mCodeDescModRefOpt_top_step(void)
{
  real_T rtb_ReferencedModel1;
```

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

The generated code contains lesser local variables in R2021b.

For more information on model references, see Generate Code for Model Reference Hierarchy.

Nonstatic data class member initialization of instance-specific parameters

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

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

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

  3.0
};

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

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

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

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

Code replacement for trigonometric functions that use lookup table approximation

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

Verification

Communication I/O information display during SIL or PIL simulation

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

Signal and state logging for SIL and PIL simulations

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

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

For information about current limitations, see SIL and PIL Limitations.

LDRA tool suite code coverage analysis

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

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

Check bug reports for issues and fixes

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

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

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