Simulink® Coder™ Release Notes

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

Simulink® 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

R2023a

R2022b

R2022a

R2020b

R2019b

R2019a

R2015aSP1

R2015b

R2013b

R2013a

R2011b

R2011a

R2023a

Model Architecture and Design

Support for continuous states with startup variant blocks

You can generate code for “startup” variant blocks that are connected to continuous state blocks. In the generated code, continuous states are initialized to zeros. Based on the variant condition that evaluates to `true` during code compilation, the associated states are assigned appropriate values.

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)`	`void slexVariantSubsystems_initialize(void)`
`{`	`{`
`  /* Enable for Sin: '<Root>/sine1' */`	`  /* Enable for Sin: '<Root>/sine1' */`
`  slexVariantSubsystems_DW.systemEnable = 1;`	`  slexVariantSubsystems_DW.systemEnable = 1;`
`  /* Enable for Sin: '<Root>/sine2' */`	`  /* Enable for Sin: '<Root>/sine2' */`
`  slexVariantSubsystems_DW.systemEnable_e = 1;`	`  slexVariantSubsystems_DW.systemEnable_e = 1;`
`  /* Enable for Sin: '<Root>/sine3' */`	`  /* Enable for Sin: '<Root>/sine3' */`
`  slexVariantSubsystems_DW.systemEnable_b = 1;`	`  slexVariantSubsystems_DW.systemEnable_b = 1;`
	`  slexVaria_startupVariantChecker();`
`  /* startup variant condition checks */`	`}`
`  utAssert(VSS_LINEAR_CONTROLLER() +`	
`  VSS_NONLINEAR_CONTROLLER() == 1);`	`static void slexVaria_startupVariantChecker(void)`
`}`	`{`
	`  /* startup variant condition checks */`
	`  utAssert(VSS_LINEAR_CONTROLLER() +`
	`  VSS_NONLINEAR_CONTROLLER() == 1);`
	`}`

Protecting dirty models is now supported

Starting in R2023a, you can protect a dirty model, which has unsaved changes.

For information about model protection, see “Protect Models to Conceal Contents”.

Diagnostic for insufficient maximum identifier length now provides shortened identifiers

Starting in R2023a, when the **Maximum identifier length** configuration parameter setting does not provide enough character length to make global identifiers unique across models, a message provides the unique identifiers that the software assigns to satisfy the maximum identifier length setting.

When you manually integrate generated code for multiple models, the shortened identifiers are more likely to clash with other identifiers. Check that these identifiers are unique for code generated separately.

To specify the diagnostic action when the code generator shortens identifiers, see **Insufficient maximum identifier length**.

Functionality being removed or changed

rtwsfcn.tlc system target file option will be removed
Warns

The option to generate an S-function target using the `rtwsfcn.tlc` system target file will be removed in a future release. Create a protected model instead.

Protected models support modeling patterns that S-function targets do not support, such as function-call signals at a component interface. Protected models also support features that S-function targets do not support, such as a read-only web view, password protection, encryption, and digital signing.

By default, the model protection process automatically collects, creates, and packages supporting files in a project that also contains the protected model. For example, the project contains supporting files that define global variables that the protected model requires.

To create a protected model, see:

- “Protect Models to Conceal Contents”
- `Simulink.ModelReference.protect`

To use a protected model, see “Reference Protected Models from Third Parties”.

S-function targets generated using the `rtwsfcn.tlc` system target file will continue to work. For information about their limitations, see “S-Function Target Limitations”.

Simulink.ModelReference.protect function will not support normal mode restrictions
Behavior change in future release

In a future release, when you use the `Simulink.ModelReference.protect` function with `Mode` set to `'Normal'`, the function will create a protected model that supports simulation when the parent model simulates in normal, accelerator, or rapid accelerator mode. This change will standardize simulation support for new protected models. Protected models created before this change will be unaffected.

Currently, when you set `Mode` to `'Normal'`, the function creates a protected model that supports simulation only when the parent model simulates in normal mode.

Simulink.ModelReference.protect function sets Mode to 'Accelerator' by default
Behavior change

Starting in R2023a, the `Simulink.ModelReference.protect` function sets the `Mode` argument to `'Accelerator'` by default. A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

With this change, the Create Protected Model dialog box and `Simulink.ModelReference.protect` function use the same default model protection mode.

Previously, the default value of the `Mode` argument was `'Normal'`. The `'Normal'` model protection mode restricts the use of the protected model such that the protected model supports simulation only when the parent model simulates in normal mode.

Code Interface Configuration and Integration

Calibration file customization

Starting from R2023a, Simulink Coder allows you to merge multiple A2L files by combing the data elements present in multiple A2L files to the data description object of 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`.

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.

Support of coder.asap2.export function for DDS Blockset Models

Starting in R2023a, the `coder.aspa2.export` function can be used to generate an A2L file for DDS Blockset models.

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”.

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_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_col_interpselsubtable`	`SubtableInterpolationCol`	"Interpolation with Subtable Selection Algorithm for Row-Major Array Layout"
`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_inline_invariant_signals`	`InvariantSignalsInline`	"Optimize Generated Code Using Inline Invariant Signals"
`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_secondOrderSystem`	`SecondOrderSystem`	"Generate C Code for a Model"
`rtwdemo_slexprfold`	`FoldBlockComputations`	"Fold Expressions"

In addition to searching in Help Center, you can use the functions `modelfinder` and `openExample` to find models and open examples.

More efficient code generation for models with row-major array layout

In R2023a, when a model with its configuration parameter **Array layout** set to `Row-major` contains a constant multidimensional matrix, the code generator produces more efficient code by inlining constant accesses to the matrix.

For example, the code previously generated for constant access to a constant matrix was similar to this.

```
void expconst1_step(void)
{
  memcpy(&expconst1_OutputParameter2[0], &expconst1_Y0[0], 12U * sizeof(real_T));

  expconst1_OutputParameter2[1] = expconst1_ConstP.Constant_Value[0];
  expconst1_OutputParameter2[2] = expconst1_ConstP.Constant_Value[1];
  expconst1_OutputParameter2[5] = expconst1_ConstP.Constant_Value[2];
  expconst1_OutputParameter2[6] = expconst1_ConstP.Constant_Value[3];
  expconst1_OutputParameter2[9] = expconst1_ConstP.Constant_Value[4];
  expconst1_OutputParameter2[10] = expconst1_ConstP.Constant_Value[5];
}

// where
typedef struct {
  /* Expression: [23, 47; 112, 313; 576, 221]
   * Referenced by: '<Root>/Constant'
   */
  real_T Constant_Value[6];
} ConstP_expconst1_T;

const ConstP_expconst1_T expconst1_ConstP = {
  /* Expression: [23, 47; 112, 313; 576, 221]
   * Referenced by: '<Root>/Constant'
   */
  { 23.0, 47.0, 112.0, 313.0, 576.0, 221.0 }
};
```

In R2023a, the generated code is similar to this.

```
void expconst1_step(void) {
    memcpy(&expconst1_OutputParameter2[0], &expconst1_Y0[0], 12U * sizeof(real_T));
    expconst1_OutputParameter2[1] = 23;
    expconst1_OutputParameter2[2] = 47;
    expconst1_OutputParameter2[5] = 112;
    expconst1_OutputParameter2[6] = 313;
    expconst1_OutputParameter2[9] = 576;
    expconst1_OutputParameter2[10] = 221;
}
```

Support for uint32 data type for enumerations

In R2023a, Simulink supports simulation and code generation for models that use enumerations with a `uint32` data type.

- Create Simulink enumerations with a `uint32` data type by calling `Simulink.defineIntEnumType` with the '`StorageType`' argument set to `uint32`.
- Create Simulink enumerations in a data dictionary with **Storage Type** set to `uint32`.
- Simulate models that use class-based enumerations with a `uint32` base type.
- Generate C and C++ code for ERT, GRT, and AUTOSAR-based targets when the model uses enumerations with a `uint32` base type.
- Support import of AUTOSAR XML descriptions of enumerations with a `uint32` data type.

Enumeration values must be less than or equal to `intmax('int32')`.

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

Code generation support for modeling elements with non-Auto storage class and empty identifier

Before R2023a, if you specified a storage class other than `Auto` for an individual modeling element and did not specify a value for the **Identifier** property, the code generator displayed an error. In R2023a, the code generator automatically generates identifiers for certain types of modeling elements for which you do not specify identifiers. The code generator uses the name of the modeling element when possible. If the modeling element does not have a name, the code generator uses the name of the associated block.

The code generator generates identifiers for root-level inports, root-level outports, signals, and block states. For general information, see Code Mappings Editor – C. For information about how the code generator generates identifiers for specific types of modeling elements, see “Configure Root-Level Inport Blocks for C Code Generation”, “Configure Root-Level Outport Blocks for C Code Generation”, “Configure Signal Data for C Code Generation”, and “Configure Block States for C Code Generation”.

Code generation support for MATLAB Function and Stateflow Chart blocks that call multi-instance Simulink functions

Starting in R2023a, you can generate code for models that contain MATLAB Function and Stateflow® Chart blocks that call multi-instance Simulink functions. For more information, see “Multi-Instance Modeling with Simulink Functions”.

Removal of axis information from A2L files that describe Direct Lookup Table (n-D) blocks

Before R2023a, if you generated an A2L file for a model that contained a Direct Lookup Table (n-D) block, the generated A2L file described the Direct Lookup Table (n-D) block as an *n*-dimensional array with axes. For example, this file represents a 2-dimensional Direct Lookup Table (n-D) as the characteristic type `MAP` with two `AXIS_DESCR` sections:

```
/begin CHARACTERISTIC
  /* Name                   */      tableData_directLookUp
  /* Long Identifier        */      ""
  /* Characteristic Type    */      MAP
  /* ECU Address            */      0x0000 /* @ECU_Address@tableData_directLookUp@ */
  /* Record Layout          */      Lookup2D_FLOAT64_IEEE
  /* Maxdiff                */      0
  /* Conversion Method      */      mg534659_lookupData_CM_double
  /* Lower Limit            */      -1.7E+308
  /* Upper Limit            */      1.7E+308
  /begin AXIS_DESCR
    /* Description of X-Axis Points */
    /* Axis Type            */      FIX_AXIS
    /* Reference to Input   */      NO_INPUT_QUANTITY
    /* Conversion Method    */      mg534659_lookupData_CM_uint32
    /* Number of Axis Pts   */      2
    /* Lower Limit          */      0
    /* Upper Limit          */      1
    FORMAT                          "%10.0"
    FIX_AXIS_PAR_DIST              0 1 2
  /end AXIS_DESCR
  /begin AXIS_DESCR
    /* Description of Y-Axis Points */
    /* Axis Type           */      FIX_AXIS
    /* Reference to Input  */      NO_INPUT_QUANTITY
    /* Conversion Method   */      mg534659_lookupData_CM_uint32
    /* Number of Axis Pts  */      3
    /* Lower Limit         */      0
    /* Upper Limit         */      2
    FORMAT                         "%10.0"
    FIX_AXIS_PAR_DIST              0 1 3
  /end AXIS_DESCR
/end CHARACTERISTIC
```

Starting in R2023a, the generated A2L file describes a Direct Lookup Table (n-D) block as an *n*-dimensional array without axes. For example, this file represents a 2-dimensional Direct Lookup Table (n-D) block as type `VAL_BLK`:

```
/begin CHARACTERISTIC
  /* Name                  */      tableData_directLookUp
  /* Long Identifier       */      ""
  /* Type                  */      VAL_BLK
  /* ECU Address           */      0x0000 /* @ECU_Address@tableData_directLookUp@ */
  /* Record Layout         */      Record_FLOAT64_IEEE
  /* Maximum Difference    */      0
  /* Conversion Method     */      mg534659_lookupData_CM_double
  /* Lower Limit           */      -1.7E+308
  /* Upper Limit           */      1.7E+308
  MATRIX_DIM                       2 3
/end CHARACTERISTIC
```

Additional capabilities for deep learning code generation

In R2023a, if you have a Deep Learning Toolbox™ license, you can generate code from a MATLAB Function block and blocks from the Deep Neural Networks library that take advantage of these new features:

- “Generate code for variable-size dlarray data type”
- “Generate code for dlnetwork objects that accept variable sequence length inputs”
- “Generate code for channel-wise convolution layer”
- “Generate code for Pooling layers with mean padding”
- “Generate code that takes advantage of learnables compression in bfloat16 format”
- “Quantized TensorFlow Lite Models: Configure predict function to accept and return fp32 values”
- “Use newer version of TensorFlow Lite library in simulation and code generation”
- “Improved performance of generated generic C/C++ code”

Deployment

Toolchain definition using Target Framework

Using the Target Framework, define and register custom makefile-based toolchains for the Simulink Coder build process. You can associate the toolchains with your development computer and target hardware. For more information, see “Define Custom Makefile-Based Toolchains Using Target Framework”.

Generic CMake toolchain for algorithm export

R2023a provides the `CMake` option for the **Toolchain** (`Toolchain`) configuration parameter. If you specify this option, the code generator:

- Generates C and C++ source code from the Simulink model without building the code.
- Creates a CMake configuration (`CMakeLists.txt`) file that does not depend on specific build tools.

If you also select the **Package code and artifacts** (`PackageGeneratedCodeAndArtifacts`) check box, the `packNGo` function packages the generated source code and CMake configuration file in a ZIP file. Use the ZIP file to export and compile the generated code in another development environment that supports the CMake build tool.

In previous releases, to create the ZIP file, you need to run the `codebuild` and `packNGo` functions.

For more information, see “Configure CMake Build Process”.

Management of target objects

The `target` package provides two new functions for the management of target objects:

- `target.update` –– Use this function to directly update target object definitions in the internal database. You do not have to remove old definitions from the database.
- `target.clear` –– Use this function to clear the internal database of all target object data from current and previous MATLAB sessions.

For more information, see “Update Modified Target Object in Internal Database” and “Clear Target Object Data From Internal Database”.

ToolchainInfo support for filenames that contain spaces

For the build process that uses `ToolchainInfo` objects, these toolchains support the use of filenames containing spaces:

- `GNU gcc/g++ | gmake (64-bit Linux)` on Linux®
- `MinGW64 | gmake (64-bit Windows)` on Windows®
- `Xcode with Clang | gmake (64-bit Mac)` on Mac

For more information, see “Build Process Support for File and Folder Names”.

Data export for XCP-based external mode simulations

You can configure an XCP-based external mode simulation to export logged signal data and top-model output data.

To export data to the MATLAB workspace:

- For logged signal data, select the **Signal logging** check box.
- For top-model output data, select the **Output** check box and, from the **Format** list, select `Dataset`.

To export the data to a MAT file instead of the MATLAB workspace, also select the **Log Dataset data to file** check box.

For more information, see “XCP External Mode Limitations”.

To Workspace block enhancements for XCP-based external mode simulations

If you set the `StreamToWks` and `MATFileLogging` configuration parameters to `'on'` and `'off'` respectively, XCP-based external mode simulations support:

- To Workspace blocks with the **Save format** block parameter set to `Timeseries`.
- To Workspace blocks in referenced models.

For more information, see To Workspace.

Functionality being removed or changed

RTW.MSVCBuild will be removed

Still runs

Support for the `RTW.MSVCBuild` workflow will be removed in a future release. Use a CMake-based toolchain to generate a Microsoft® Visual Studio® Solution. For more information, see “Compile and Debug Generated C Code with Microsoft Visual Studio”.

Performance

Removed redundant operations that use identical values

In R2023a, the generated code no longer contains some redundant operations that access values that are identical at run time. By eliminating these redundant operations, the generated code shows improved execution speed.

For example, consider this sigmoid function, which uses a variable-size input.

```
function sigm = sigmoidFcn(x)
    sigm = 1 ./ (1 + exp(-x));
end
```

In R2022b, the generated code for `sigmoidFcn` contained this code, which accesses the data in the `sigm_data` array in three separate `for` loops.

```
    int32_T k;
    int32_T loop_ub;
    sigm_size[0] = x_size[0];
    sigm_size[1] = x_size[1];
    loop_ub = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = -x_data[k];
    }

    loop_ub = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = exp(sigm_data[k]);
    }

    for (k = 0; k < loop_ub; k++) {
        sigm_data[k] = 1.0 / (sigm_data[k] + 1.0);
    }
```

In R2023a, the generated code combines the operations and accesses the data in the `sigm_data` array in only one `for` loop.

```
    int32_T k;
    int32_T loop_ub_tmp;
    sigm_size[0] = x_size[0];
    sigm_size[1] = x_size[1];
    loop_ub_tmp = x_size[0] * x_size[1];
    for (k = 0; k < loop_ub_tmp; k++) {
        sigm_data[k] = 1.0 / (exp(-x_data[k]) + 1.0);
    }
```

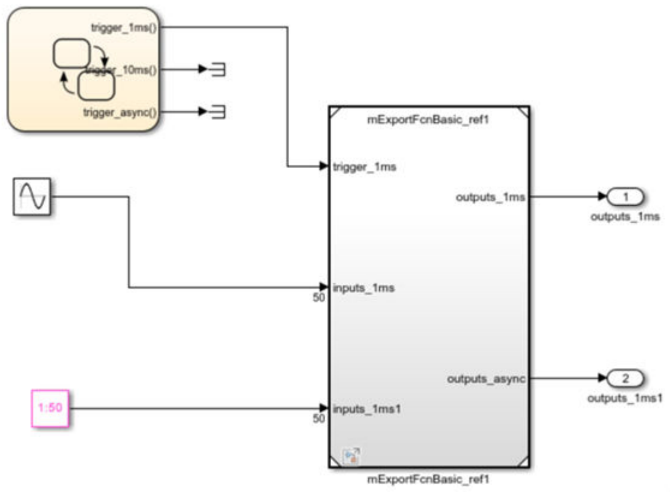
The generated code produces the same output as the code from R2022b, but the R2023a code accesses the array data fewer times and shows improved execution speed.

Control stack size for models containing export-function referenced models

Starting in R2023a, for models containing a referenced export-function model, you can control the number of local and global variables in the generated code by specifying the **Maximum stack size**

(bytes). The generated code now defines local variables until the specified maximum stack space is consumed, after which the code defines the variables as global.

Consider, the model `mExportFcnBasic_Top` that contains the referenced export-function model `mExportFcnBasic_ref1`. The **Maximum stack size (bytes)** is set to `20`.



In R2022b, the code generator produced this code:

```
/* Model step function */
void mExportFcnBasic_Top_step(void)
{
  real_T tmp;
  int32_T i;

  /* Sin: '<Root>/Sine Wave' */
  tmp = sin(1.0 * mExportFcnBasic_Top_M->Timing.t[0] + 0.0);
  for (i = 0; i < 50; i++) {
    /* Sin: '<Root>/Sine Wave' */
    mExportFcnBasic_Top_B.SineWave[i] = tmp * rtCP_SineWave_Amp[i] + 0.0;
  }
```

The code defined `tmp` and `i` as local variables even after the referenced model consumed 20 bytes stack space.

In R2023a, the generated code defines `tmp` and `i` as global variables in `mExportFcnBasic_Top.h`, which reduces the number of local variables and stack usage.

```
/* Block signals (default storage) */
typedef struct {
  real_T SineWave[50];   /* '<Root>/Sine Wave' */
  real_T d;
  int32_T i;
} B_mExportFcnBasic_Top;
```

For more information, see “Customize Stack Space Allocation”.

Use coder.loop.Control objects inside MATLAB Function blocks to improve for loop performance in generated code

In R2023a, you can instruct the code generator to transform specific `for` loops inside your MATLAB Function block in a variety of ways (such as parallelize or vectorize) during code generation. These transforms can often improve the run-time performance of large loops. To specify these transforms, use the following directives inside your MATLAB blocks:

- `coder.loop.interchange`: Interchange nested loops to improve cache performance when accessing array elements.
- `coder.loop.parallelize`: Parallelize loop execution to improve speed by utilizing available threads.
- `coder.loop.reverse`: Reverse the execution order of loop iterations. In some situations, the reversed loop execution can be faster.
- `coder.loop.tile`: Tile loop nests to reduce memory access latency.
- `coder.loop.unrollAndJam`: Unroll and jam loops to improve cache locality.
- `coder.loop.vectorize`: Generate code that uses SIMD instructions to apply an operation simultaneously to multiple instances.

In your MATLAB code, call the directive immediately before the loop you intend to transform. You can combine multiple transforms into a single call by using the dot builder syntax shown in this code snippet. Use the loop index variable name to specify the loop to which you intend to apply a certain transform.

```
coder.loop.parallelize('loopId').interchange('loopId','loopId2');
for loopId = 1:100
    for loopId2 = 1:100
...
end
```

In some situations, you might want to combine multiple transforms in more complex ways than is possible by using this simple dot builder syntax. For example, you might want to apply one of the transforms only if a certain compile-time condition is satisfied. In such situations, use the `coder.loop.Control` objects and the associated object functions. For example:

```
...
loopControl = coder.loop.Control;
loopControl = loopControl.parallelize('loopId');

% You can apply multiple transforms to the same object
if inputVal > threshold
    loopControl = loopControl.interchange('loopId','loopId2');
end

% Call the apply method to inform the code
% generator to affect the loops in the generated code
loopControl.apply;

for loopId = 1:100
    for loopId2 = 1:100
...
end
```

See “Optimize Loops in Generated Code”.

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

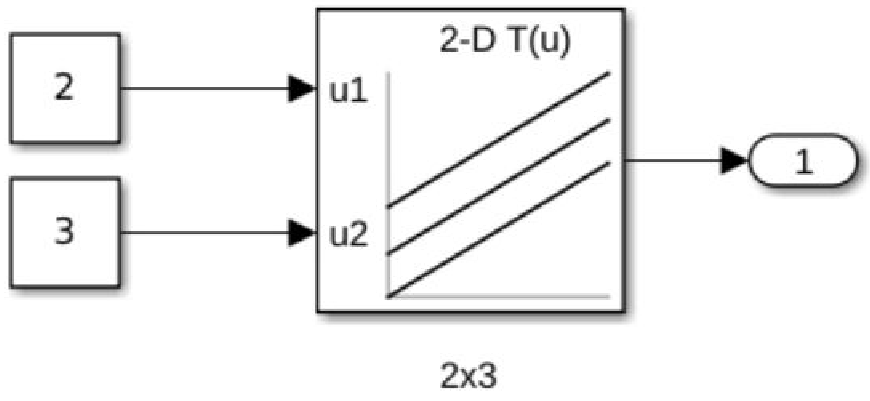
Model Architecture and Design

Configure Simulink.LookupTable object to support differently sized table and breakpoints

Previously, if a model used `Simulink.LookupTable` objects as model arguments, these objects were required to be of the same size as the table and breakpoints.

In R2022b, in the `Simulink.LookupTable` object property dialog box, select the new **Allow multiple instances of this type to have different table and breakpoint sizes** and **Support tunable size** parameters to configure these objects to support differently sized table and breakpoints and generate code for the model. The code generator produces pointers instead of arrays for the table and breakpoint fields in the `struct` definition. These pointers point to arrays that the code generator allocates outside the `struct` definition that have the true data of the table and breakpoints.

For example, consider this model that has an n-D Lookup Table block that is configured with a `Simulink.LookupTable` object, named `LUT1A`. `LUT1A` includes one table with `[2 3]` dimensions and two breakpoint data sets with `[1 2]` and `[1 3]` dimensions.



This is the generated code for the differently sized table and breakpoints in *<model>*`_types.h`:

```
typedef struct {
  uint32_T N1;
  uint32_T N2;
  real_T *BP1;
  real_T *BP2;
  real_T *T;
} LutObjChild_type;
```

The data is stored in the `<model>_data.c` file:

```
   * Expression: LUT1A.Table
   * Referenced by: '<Root>/1-D Lookup Table'
   */
  { 1.0, 4.0, 2.0, 5.0, 3.0, 6.0 },
  /* Expression: LUT1A.Breakpoints(0)
   * Referenced by: '<Root>/1-D Lookup Table'
   */
  { 1.0, 2.0 },

  /* Expression: LUT1A.Breakpoints(1)
   * Referenced by: '<Root>/1-D Lookup Table'
   */
  { 1.0, 2.0, 3.0 }
};
```

If your model uses `Simulink.LookupTable` objects with differently sized table and breakpoints, you can choose storage classes for each object separately in the model code mappings from the **Model Parameters** section on the **Parameters** tab in the Code Mappings editor or by using the code mappings programming interface. For the additional arrays generated from the `Simulink.LookupTable` object, you can choose a default storage class for the objects in the **Model Parameters** section of the **Data Defaults** tab.

With Embedded Coder®, you can replace code from Lookup table blocks that support differently sized table and breakpoint objects by using a code replacement library. For more information, see Code replacement for lookup tables that support differently sized table and breakpoint objects.

Functionality being removed or changed

Option to generate S-function from subsystem will be removed
Still runs

The **C/C++ Code** > **Generate S-Function** option that is available when you right-click a Subsystem block will be removed in a future release. Create a protected model instead.

Protected models support modeling patterns that S-function targets do not support, such as function-call signals at a component interface. Protected models also support features that S-function targets do not support, such as a read-only web view, password protection, encryption, and digital signing.

By default, the model protection process automatically collects, creates, and packages supporting files in a project that also contains the protected model. For example, the project contains supporting files that define global variables that the protected model requires.

To create a protected model, see:

- Protect Models to Conceal Contents
- Simulink.ModelReference.protect

To use a protected model, see Reference Protected Models from Third Parties.

S-function targets generated from subsystems will continue to work. For information about their limitations, see S-Function Target Limitations.

In R2022b, when you right-click a subsystem and select **C/C++ Code** > **Generate S-Function**, the build process immediately starts code generation and compilation of the S-function. Previously, the software opened a window, and you were required to click the **Build** button to initiate the build process. From the window, you were also able to declare parameters tunable and create a software-in-the-loop (SIL) block. For information about how you can now perform these tasks, see C Code

Generation Configuration for Model Interface Elements and SIL or PIL Block Simulation (Embedded Coder), respectively.

Code Interface Configuration and Integration

Calibration File Customization

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

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

The Simulink Coder allows you to add, delete, modify, find, filter, fetch measurements, characteristics, functions, and `compu-methods` 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.

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.

Code Generation

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

Function and file packaging changes for reusable library subsystems

Starting in R2022b, the code generator handles function and file packaging of generated code differently. For reusable library subsystems, the code generator produces subsystem code in the `slprj/`*`target`*`/_sharedutils` folder. The subsystem code is shared across models in a model reference hierarchy for these combinations of subsystem parameters:

Function name options	File name options
`Auto`	`Use subsystem name`
`Use subsystem name`	`Use subsystem name`
`Auto`	`Use function name`
`Use subsystem name`	`Use function name`

For more information, see Generate Reusable Code from Library Subsystems Shared Across Models.

Deployment

CMake toolchain definitions for build process

To build code that you generate from Simulink models, you can now specify CMake toolchain definitions for:

- Microsoft Visual C++® and MinGW® on Windows, GCC on Linux, and Xcode on Mac computers, using Ninja and makefile generators.
- Microsoft Visual Studio and Xcode project builds.

If a supported toolchain is installed on your development computer, you can specify the corresponding CMake toolchain definition for your model. When you run `slbuild`, press **Ctrl+B**, or run a software-in-the-loop (SIL), processor-in-the-loop (PIL), or external mode simulation, CMake:

1. Uses configuration (`CMakeLists.txt`) files to generate standard build files.
2. Runs the compiler and other build tools to create executable code.

For more information, see Configure CMake Build Process and Supported and Compatible Compilers for Windows, Linux, and Mac.

Creation of custom CMake toolchain definitions

Using the `target` package, you can create custom CMake toolchain definitions for building code that you generate from Simulink models. You can:

- Specify CMake parameters, for example, `Generator` and `Toolchain file`.
- Associate the toolchain with operating systems of your development computers.
- Associate the toolchain with your target hardware.
- Add the toolchain definition to an internal database. You can use the toolchain in subsequent MATLAB sessions.

For more information, see Create Custom CMake Toolchain Definition and Supported and Compatible Compilers for Windows, Linux, and Mac.

Toolchain setting Clang v3.1 | gmake (64-bit Mac) renamed

The `Clang v3.1 | gmake (64-bit Mac)` setting for the Toolchain configuration parameter is renamed `Xcode with Clang | gmake (64-bit Mac)`.

When you open a model created by using an earlier release, R2022b automatically updates the **Toolchain** setting. If you export an R2022b model in the format of an earlier release, the **Toolchain** setting reverts to the earlier release setting.

Target Language Complier 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 (Embedded Coder).

Automatic memory configuration for XCP-based external mode simulations

In XCP-based external mode model simulations, the software determines and allocates the required static memory. For models that stream a large amount of data to Simulink or run on low-memory target devices, you do not have to determine and specify memory requirements manually.

If you specify an XCP-based protocol for the **Transport layer** configuration parameter, the Configuration Parameters dialog box does not display the **Static memory allocation** check box. Instead, the dialog box provides two new configuration parameters:

- **Automatically allocate static memory** (`ExtModeAutomaticAllocSize`) — The check box is selected by default. The software automatically allocates the static memory required for the buffers used in external mode communication. The size of memory required depends on the model and the value of the **Maximum duration** parameter. If you clear the check box, you can use the **Static memory buffer size** field to specify memory allocation.
- **Maximum duration** (`ExtModeMaxTrigDuration`) — Use this field to specify the maximum value of `ExtModeTrigDuration` that the software must consider when it determines the required static memory for external mode communication.

For more information, see:

- Memory Allocation for Communication Buffers During XCP External Mode Simulation
- Automatically allocate static memory
- Maximum duration

XCP-based external mode simulations with generated C++ code

You can now run an XCP-based external mode simulation with these configuration parameter settings:

- **Language** — `C++`
- **Code interface packaging** — `C++ class`

Previously, the simulation produced this error:

```
When external mode simulation is chosen, the generated code is not reusable.
Consider setting ‘Code interface packaging’ to ‘Nonreusable function’ on the
Configuration Parameters > Code Generation > Interface pane.
```

For more information, see External Mode Simulation by Using XCP Communication.

XCP external mode support for Simulink.ImageType

XCP-based external mode simulations now support the `Simulink.ImageType` data type. For more information, see External Mode Simulation by Using XCP Communication.

Performance

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.

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.

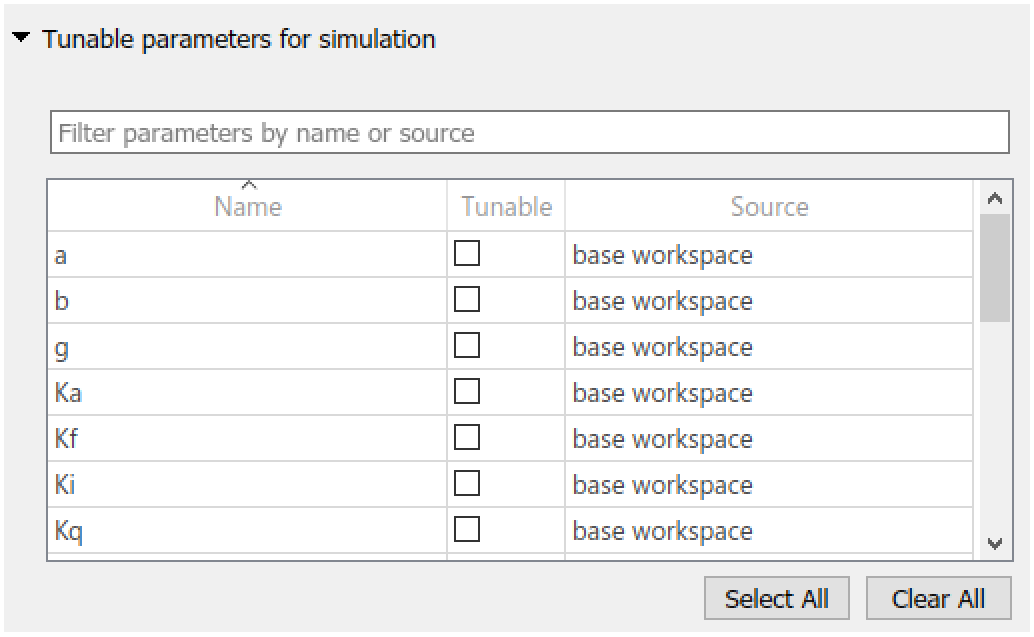
R2022a

Model Architecture and Design

Specify tunable parameters for protected models

When you create a protected model, you can now specify which parameters recipients can tune during simulation.

In the Create Protected Model dialog box, select the tunable parameters individually, or click **Select All**. For more information, see Protect Models to Conceal Contents.


Tunable parameters for simulation
Filter parameters by name or source
Select All
Clear All

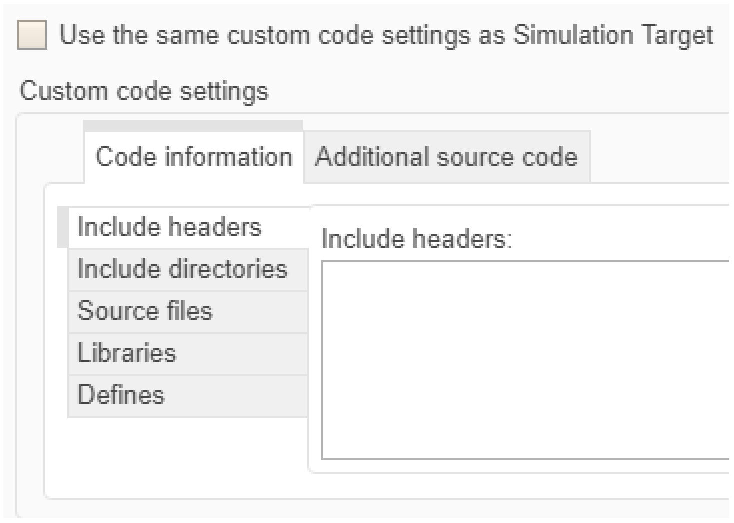
Name	Tunable	Source
a	☐	base workspace
b	☐	base workspace
g	☐	base workspace
Ka	☐	base workspace
Kf	☐	base workspace
Ki	☐	base workspace
Kq	☐	base workspace

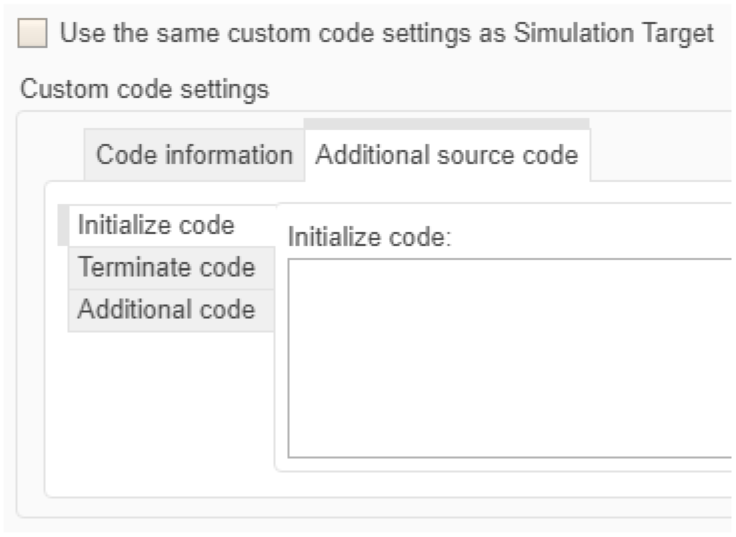
With the `Simulink.ModelReference.protect` function, specify tunable parameters with the new `TunableParameters` name-value argument. By default, no parameters are tunable during simulation.

To get the list of tunable parameters for a protected model, use the `Simulink.ProtectedModel.getTunableParameters` function.

Redesigned Custom Code Pane of Model Configuration Parameters

The Custom Code pane of the Model Configuration Parameters dialog is changed for R2022a. The parameters that pertain to custom code are now together in one section of the dialog, which is organized into two tabs: **Code information** and **Additional source code**. Clicking on a tab shows the parameters that are listed under that tab and hides the contents of the other tab.





Some of the parameters have new names. The following table lists the parameters in the new Custom Code pane dialog that have different names or are in different locations as a result of this change.

Parameter name prior to R2022a	New name in R2022a	New location in R2022a
Insert custom C code in generated > Header file	**Include headers**	**Code information** tab
Insert custom C code in generated > Source file	**Additional code**	**Additional source code** tab
Insert custom C code in generated > Initialize function	**Initialize code**	**Additional source code** tab
Insert custom C code in generated > Terminate function	**Terminate code**	**Additional source code**

Parameter name prior to R2022a	New name in R2022a	New location in R2022a
Additional build information > Source files	**Source files**	**Code information** tab
Additional build information > Include directories	**Include directories**	**Code information** tab
Additional build information > Libraries	**Libraries**	**Code information** tab
Additional build information > Defines	**Defines**	**Code information** tab

Functionality being removed or changed

Tunable parameters for simulation are independent of storage class
Behavior change

Starting in R2022a, when you protect a model with the `Simulink.ModelReference.protect` function, you must specify the parameters of the protected model that you want to be tunable during simulation by using the `TunableParameters` name-value argument. By default, no parameters are tunable during simulation.

Previously, a parameter was tunable when its code generation storage class was set to a value other than `Auto`. By default, the storage class for an individual data element is `Auto`.

Code Interface Configuration and Integration

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.

Code Generation

ARM64 code for Apple platforms

On the **Hardware Implementation** pane, you can specify the device vendor and type `Apple-ARM64`, which enables you to generate ARM64 code for Apple platforms. For more information, see:

- Device vendor
- Device type
- Hardware Implementation Pane

Scoped enum classes for C++11

In R2021b, during C++ code generation, the code generator produced or imported C-style enum blocks in the generated code for exported or imported enumerations. If the **DataScope** property of the enumeration is `Exported`, the code generator produced a C-style enum definition, such as:

```
typedef enum {
    Red = 0,
    Yellow,
    Blue
} BasicColors;
```

or

```
typedef int8_T BasicColors;

#define Red      ((BasicColors)0)
#define Yellow ((BasicColors)1)
#define Blue     ((BasicColors)2)
```

The usage of these enumerations were as follows:

```
void mEnumInitModelClass::initialize()
{
  rtU.In1 = Blue;
  ...
}
```

In R2022a, during C++11 code generation, the code generator produces or imports enumerations only as scoped enum classes in the generated code. The enum definition also specifies the underlying integer type. If the **DataScope** property of the enumeration is `Exported`, the code generator produces this code:

```
enum class BasicColors : int8_T{
    Red = 0,
    Yellow = 1,
    Blue = 2
};
```

The use of these enumerations are qualified by the enum class name:

```
void mEnumInit::initialize()
{
  rtU.In1 = BasicColors::Blue;
  ...
}
```

When you set **Language** as `C++` and **Standard math library** as `C++11 (ISO)`, the code generator produces scoped enum classes that reduce violations of AUTOSAR C++14 Rule A7-2-2 (Polyspace Bug Finder) AUTOSAR C++14 Rule A7-2-3 (Polyspace Bug Finder). For more information, see Use Enumerated Data in Generated Code.

Compatibility Considerations

If you import enumerations that contain C-style blocks (`typedef enum` syntax) and generate C++11 code, the generated code fails to compile. Update the imported enumeration definition in the header file to be scoped enum classes (`enum class` syntax).

GRT-based system target files no longer support SimStruct

Errors

Starting in R2022a, in GRT-based system target files, you must set the TLC variable `GenRTModel` to 1. That variable setting instructs the code generator to use the real-time model data structure `rtModel` instead of the simulation data structure `SimStruct`. The `rtModel` data structure encapsulates model-specific information in a more compact form. If you set `GenRTModel` to 0, the code generator returns an error. For more information, see Data Structures in the Generated Code.

Generate code for dlnetwork workflows that use deep learning arrays

In R2022a, you can generate code for `dlnetwork` (Deep Learning Toolbox) and `dlarray` (Deep Learning Toolbox) that you use to run inference with `dlnetwork`. The `dlnetwork` object code generation supports the Intel® Math Kernel Library for Deep Neural Networks (MKL-DNN) library for CPUs.

You can use MATLAB Function block or the Predict or Image Classifier block from the **Deep Neural Networks** library to import the `dlnetwork` into Simulink.

Deployment

Files modelsources.txt and defines.txt not generated

When you build your model, the `modelsources.txt` and `defines.txt` files are not generated. Previously, the build process created the files in the build folder. For more information, see Manage Build Process Folders and Manage Build Process Files.

Reduced bandwidth contiguous signal streaming by using XCP DAQ packed mode

For XCP-based external mode simulations, the External Signal & Triggering dialog box has a **Send multiple contiguous samples in same packet** check box. If you select the check box, the XCP server uses the XCP DAQ packed mode for streaming signals from the target hardware to Simulink. The mode improves the ratio of useful data to control data in streamed signals, sending data from multiple time steps in each packet. For more information, see XCP External Signal & Triggering Dialog Box.

Build process support for Unicode characters

Previously, if a build folder path contained Unicode® characters that did not belong to the system locale, the build process might produce an error. In R2022a, a build process that uses a Microsoft Visual C++ compiler produces compilation artifacts (including the `nmake` makefile) that are UTF-8 encoded. You can build code in a folder where the path contains Unicode characters that do not belong to the system locale. For more information, see Build Process Support for Folder Names.

Microsoft Visual C++ 2022 toolchain support for Windows

On Windows, you can compile generated code by using the Microsoft Visual C++ 2022 product family. For more information, see Supported Compilers.

Performance

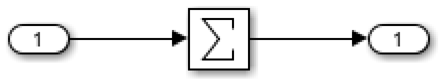
SIMD code for reduction operations

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

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.

Improved performance of generated generic C/C++ code

In R2022a, the generated generic C/C++ code performance (that does not depend on third-party libraries) for the following layers in your deep neural network has improved:

- `bilstmLayer` (Deep Learning Toolbox)

- `convolution2dLayer` (Deep Learning Toolbox)
- `fullyConnectedLayer` (Deep Learning Toolbox)
- `gruLayer` (Deep Learning Toolbox)
- `lstmLayer` (Deep Learning Toolbox)

In addition, you can generate generic C/C++ code that uses SIMD intrinsics for these layers. Use of SIMD intrinsics is likely to further improve the performance the generated code. To generate code that uses SIMD intrinsics, do one of the following:

- Specify a code replacement library that supports SIMD, for example, GCC ARM Cortex-A. To specify a code replacement library, set the code generation configuration parameter **Code replacement library**.
- Specify SIMD instruction set for the target hardware by setting the code generation configuration parameter **Leverage target hardware instruction set extensions**.

Verification

MATLAB Coder Interface for Visual Studio Code Debugging

If you install the support package MATLAB Coder Interface for Visual Studio Code Debugging, you can use Visual Studio Code as the graphical user interface for these debuggers:

- MinGW GDB on Windows
- GDB on Linux
- LLDB on macOS

For information about installing the support package, in MATLAB Central™ File Exchange, search for `MATLAB Coder Interface for Visual Studio Code Debugging`.

For information about debugger support, see Debug Generated Code During SIL Simulation (Embedded Coder).

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

Model Architecture and Design

New ID for check "Check for relative execution order change for Data Store Read and Data Store Write blocks"

Starting in R2021b, the ID for Model Advisor check Check for relative execution order change for Data Store Read and Data Store Write blocks is changed to `mathworks.design.datastoresimrtwcmp`. The previous ID was `com.mathworks.sorting.datastoresimrtwcmp`.

Protection for models that use noninlined S-functions

Starting in R2021b, you can protect models that use noninlined S-functions. Previously, you could not protect models that used noninlined S-functions directly or indirectly. For more information, see Protect Models to Conceal Contents.

Code Interface Configuration and Integration

Changes to model hierarchy requirements

Starting in R2021b, the code generator allows a model reference hierarchy to have different specifications for model configuration parameter **Generate C API for: root-level I/O**.

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

Calibration file customization

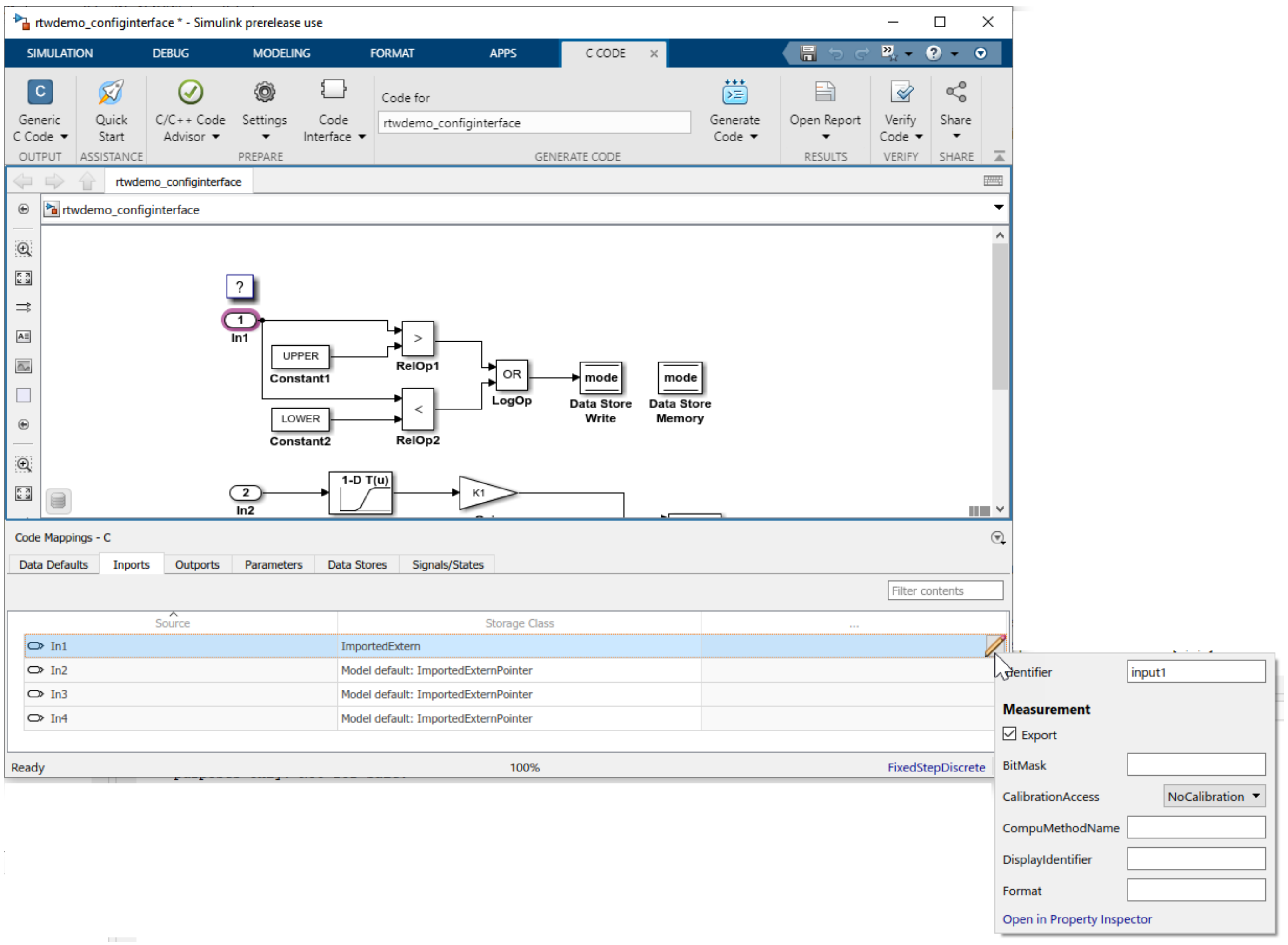
Starting in R2021b, you can customize the ASAP2(a2l) file. The **Code Mappings editor** enables you to customize the calibration properties of measurement and characteristic objects. For example, you can set the properties **`Calibration Access`** and add a **`Display Identifier`** by using the Code 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.

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.



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

Identifier	input1

Measurement

☑ Export

BitMask	
CalibrationAccess	NoCalibration
CompuMethodName	
DisplayIdentifier	
Format	

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

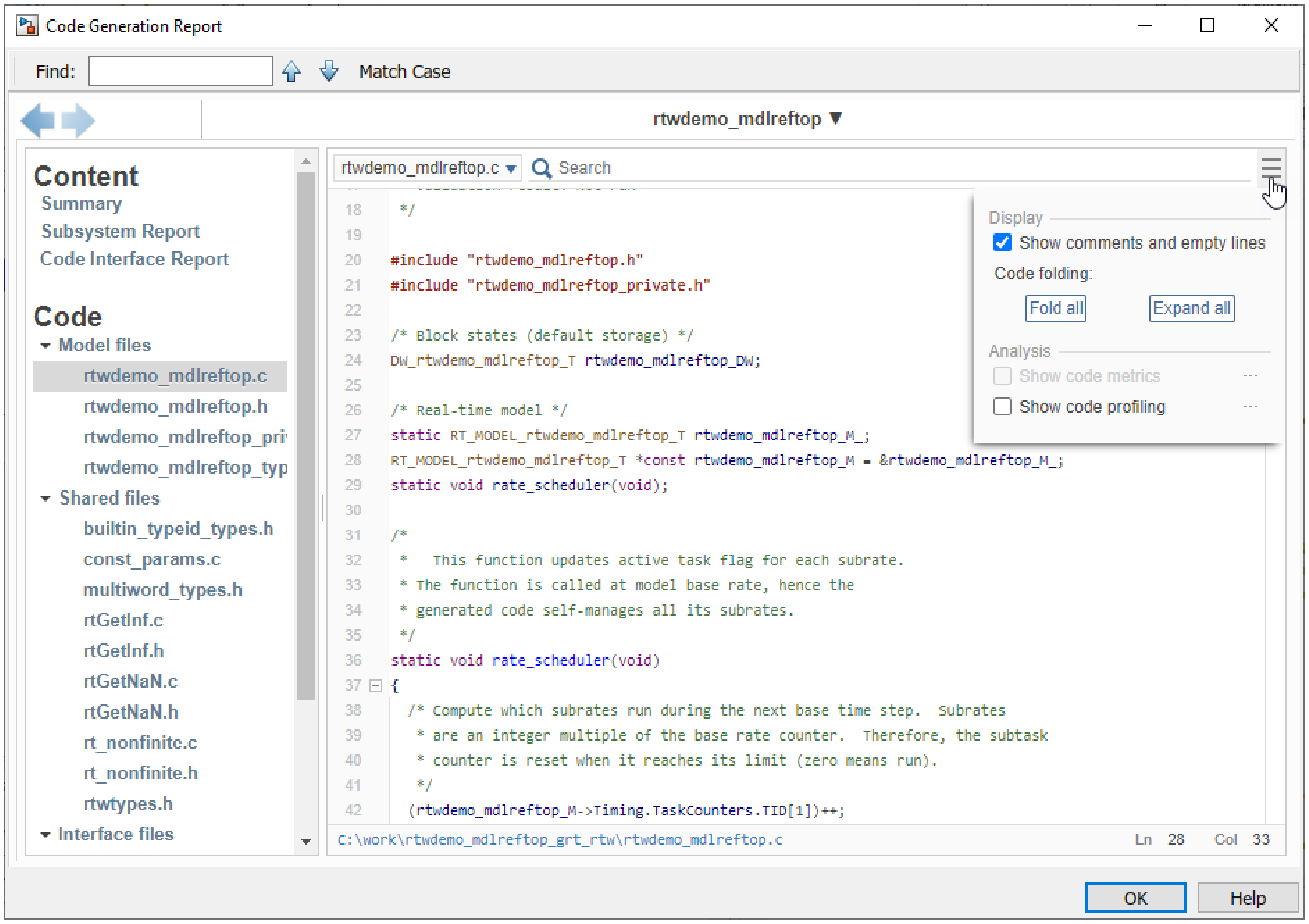
Beginning in R2021b, the Code Mappings editor displays data related to In Bus Element and Out Bus Element blocks in a hierarchical view. In previous releases, this data displayed as a flat list in the Code Mappings editor.

Code Generation

Code generation report enhanced with new functionalities

Starting in R2021b, the code generation report has been enhanced to include new functionalities. In the new code generation report, you can:

- View your model hierarchy and quickly navigate to reports for other models in the hierarchy.
- For a model, search the generated code files simultaneously.
- Customize the code display by controlling comments, code folding, and code metrics information.



Show comments and empty lines ✓

Show code metrics ☐

Show code profiling ☐

```
 */

#include "rtwdemo_mdlreftop.h"
#include "rtwdemo_mdlreftop_private.h"

/* Block states (default storage) */
DW_rtwdemo_mdlreftop_T rtwdemo_mdlreftop_DW;

/* Real-time model */
static RT_MODEL_rtwdemo_mdlreftop_T rtwdemo_mdlreftop_M_;
RT_MODEL_rtwdemo_mdlreftop_T *const rtwdemo_mdlreftop_M = &rtwdemo_mdlreftop_M_;
static void rate_scheduler(void);

/*
 *   This function updates active task flag for each subrate.
 * The function is called at model base rate, hence the
 * generated code self-manages all its subrates.
 */
static void rate_scheduler(void)
{
  /* Compute which subrates run during the next base time step.  Subrates
   * are an integer multiple of the base rate counter.  Therefore, the subtask
   * counter is reset when it reaches its limit (zero means run).
   */
  (rtwdemo_mdlreftop_M->Timing.TaskCounters.TID[1])++;
```

C:\work\rtwdemo_mdlreftop_grt_rtw\rtwdemo_mdlreftop.c

For more information, see Reports for Code Generation.

In the code generation report, you can still generate a web view of your model and generate additional report sections, such as the subsystem report. If you collect code coverage results for your model, the code generation report does not include the coverage results. Instead, use the coverage report generated by the coverage tool that you use to collect results.

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 - `rtmodel.c` or `rtmodel.cpp` and `rtmodel.h`
- For models configured for concurrent execution - `rt_main.c` or `rt_main.cpp`

For more information, see Manage Build Process Files and Analyze the Generated Code Interface.

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 uses the wrapper function, at least temporarily:

- For models configured for multitasking, you can update the `#include` statement to specify `rtmodel.h` instead of `*model*.h`.
- For models configured for concurrent execution, you can copy the wrapper function from the generated example `main` program `rt_main.c` or `rt_main.cpp` and paste it into your application `main` program.

Target hardware data management

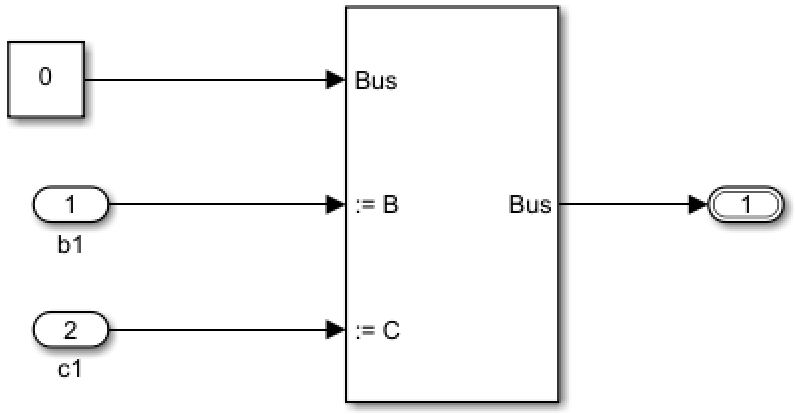
The `target` package provides these enhancements for target hardware data management:

- `NumberOfCores`, `NumberOfThreadsPerCore`, and `NumberOfLogicalCores` properties for `target.Processor` objects, which you can use to describe multicore processor architectures. For more information, see `target.Processor`.
- `NonStandardDataTypes` property for `target.DataTypes` objects, which you can use to capture nonstandard data type implementations. For more information, see `target.LanguageImplementation`.

File size reduction by using memset function for zero initialization

Starting in R2021b, the code generator uses the `memset` function to initialize bus objects and enumerations to zero when you select the model configuration parameter **Use memset to initialize floats and doubles to 0.0**.

Consider this model. The Bus Assignment block receives several inputs, which initialize to zero.



In R2021a, the code generator initialized the bus object by using a ground constant, even if you selected the **Use memset to initialize floats and doubles to 0.0** parameter.

```
const BusObject2 test_model1_rtZBusObject2 = {
   {
      {
         0.0,                          /* a */
         FPL_MAG_VAR_SOURCE_LOCAL,     /* b */
         0.0                           /* c */
      }, {
         0.0,                          /* a */
         FPL_MAG_VAR_SOURCE_LOCAL,     /* b */
         0.0                           /* c */
      }, {
         0.0,                          /* a */
         FPL_MAG_VAR_SOURCE_LOCAL,     /* b */
         0.0                           /* c */
      }
   }
}

/* Model step function */
void test_model1_step(void)
{
   BusObject2 rtb_BusAssignment;

   /* BusAssignment: '<Root>/Bus Assignment' incorporates:
    *  Constant: '<Root>/Constant'
    *  Inport: '<Root>/b1'
    *  Inport: '<Root>/c1'
    */
   rtb_BusAssignment = test_model1_rtZBusObject2;
   rtb_BusAssignment.b1 = test_model1_U.b1;
   rtb_BusAssignment.c1 = test_model1_U.c1;

   /* Outport: '<Root>/Outport' */
   test_model1_Y.Outport = rtb_BusAssignment;
}
```

In R2021b, the code generator initializes the bus object by using the `memset` function:

```
/* Model step function */
void mGndCnstWithEnum_step(void)
{
   BusObject2 rtb_BusAssignment;

   /* BusAssignment: '<Root>/Bus Assignment' incorporates:
    *  Inport: '<Root>/b1'
    *  Inport: '<Root>/c1'
    */
   memset(&rtb_BusAssignment, 0, sizeof(BusObject2));
   rtb_BusAssignment.b1 = mGndCnstWithEnum_U.b1;
   rtb_BusAssignment.c1 = mGndCnstWithEnum_U.c1;

   /* Outport: '<Root>/Outport' */
   mGndCnstWithEnum_Y.Outport = rtb_BusAssignment;
}
```

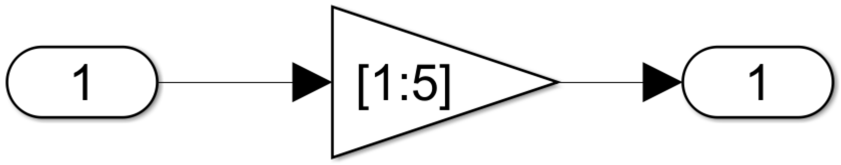
This change reduces the size of the generated source file. For more information, see Optimize Generated Code Using `memset` Function.

Data reference of model parameter in model.rtw

In R2021a, by default, if you did not specify a value for the model configuration parameter `RTWDataReferencesMinSize`, the code generator assigned the default value `10` to the parameter.

In R2021b, by default, if you do not specify a value for the model configuration parameter `RTWDataReferencesMinSize`, the code generator assigns the value `-1` to the parameter. In this case, the code generator writes a data reference of a model parameter to the *`model`*`.rtw` file in place of a data vector. This optimization reduces the file read/write operations, which improves the code generation time. When you specify a positive value for the `RTWDataReferencesMinSize` parameter, the code generator uses the value as the threshold size based on which it writes data references to the *`model`*`.rtw` file in place of a data vector.

For example, consider this model:



In R2021a, if you did not specify any value for the `RTWDataReferencesMinSize` parameter, the code generator wrote the data vector to the *`model`*`.rtw` file:

```
Parameter {
      Identifier          "Gain_Gain"
      LogicalSrc          P0
      Protected         no
      Value            [1.0, 2.0, 3.0, 4.0, 5.0]
      CGTypeIdx         19
      ContainerCGTypeIdx     18
```

In R2021b, if you did not specify any value for the `RTWDataReferencesMinSize` parameter, the code generator writes the data reference of the parameter to the *`model`*`.rtw` file:

```
Parameter {
      Identifier          "Gain_Gain"
      LogicalSrc          P0
      Protected         no
      Value             SLData(0)
      CGTypeIdx         20
      ContainerCGTypeIdx     19
```

For more information, see Data References in the *`model`*`.rtw` File.

Model parameter value in TLC

In R2021a, you could access the data vector of the model parameter written to the `ModelParameters` record in the *`model`*`.rtw` by fetching the `Value` record in your TLC script. For example, you used:

```
%assign numRowValues = SIZE(RowIndex.Value, 1)
```

In R2021b, you can access the data vector of the model parameter by using the new TLC function `LibGetParameterValue`. The `LibGetParameterValue` function is in the `paramlib.tlc` file. For example, you can use:

```
%assign numRowValues = SIZE(LibGetParameterValue(RowIndex), 1)
```

For more information, see LibGetParameterValue.

Language standard parameter for configuring C/C++ language standard

Previously, you configured the C/C++ language standard by using the **Standard math library** parameter located on the **Interface** pane in the Configuration Parameters dialog box. You can now use the **Language standard** parameter located on the **Code Generation** pane. For more information, see Language standard.

Deployment

CMake ships with MATLAB

The executable file for CMake, a third-party, open-source tool for build process management, is included as part of MATLAB. After generating code from a model, you can use `codebuild` to create CMake configuration (`CMakeLists.txt`) files, and then invoke CMake to build the generated code. For more information, see Approaches for Building Code Generated from Simulink Models and Compile Code in Another Development Environment.

XCP external mode simulation through concurrent execution

On Windows and Linux computers, you can run external mode simulations that generate, build, and concurrently execute model code for the native threads example.

For more information, see Build on Desktop and XCP External Mode Limitations.

Simplified creation of XCP target connectivity objects

Previously, multiple `target.create` function calls were necessary to create the target connectivity objects required for XCP-based external mode simulation. In R2021b, you can create the objects through a single call to `target.create`. For more information, see:

- Step 6 in Customise Connectivity for XCP External Mode Simulations
- `target.XCPPlatformAbstraction`
- `target.XCP`
- `target.XCPExternalModeConnectivity`
- `target.ExternalMode`

Parameter upload for external mode simulations

If you set `DefaultParameterBehavior` to `'Inlined'`, the code generator embeds numeric model parameter values instead of symbolic parameter names in the generated code. You can use `Simulink.Parameter` objects to remove parameters from inlining and declare the parameters tunable. Previously, for XCP external mode simulations, when you connected Simulink to the target application, Simulink did not upload the numeric values of the tunable parameters from the target application. In R2021b, Simulink uploads the parameter values to the model.

For XCP, TCP/IP, and serial external mode simulations, previously, Simulink did not upload workspace parameter values that had enumerated data types. In R2021b, Simulink uploads the parameter values.

For more information, see:

- Create Tunable Calibration Parameter in the Generated Code
- Automatic Parameter Uploading on Host/Target Connection
- XCP External Mode Limitations
- TCP/IP and Serial External Mode Limitations

Hold Updates button for Run on Custom Hardware app

In the Run on Custom Hardware app for external mode simulations, the **Hold Updates** toggle button replaces the **Batch Mode** button. If you toggle on the **Hold Updates** button, you can defer the update of block parameters until you toggle off the button or click the **Update All Parameters** button.

For more information, see External Mode Simulation by Using XCP Communication and External Mode Simulation with TCP/IP or Serial Communication.

removeSourceFiles function for RTW.BuildInfo object

The `RTW.BuildInfo` class provides the `removeSourceFiles` function for the removal of source files from a build information object. For more information, see Remove Source File Names from Build Information.

Performance

Generation of SIMD code for Intel hardware

In R2021b, you can generate single instruction, multiple data (SIMD) code from Simulink models by using Intel SSE technology.

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 platforms.

To generate code that uses SIMD intrinsics, set the new configuration parameter **Leverage target hardware instruction set extensions** to SSE2. If you have Embedded Coder, you can generate code that uses additional SIMD instruction sets. For more information, see Generate SIMD Code from Simulink Blocks.

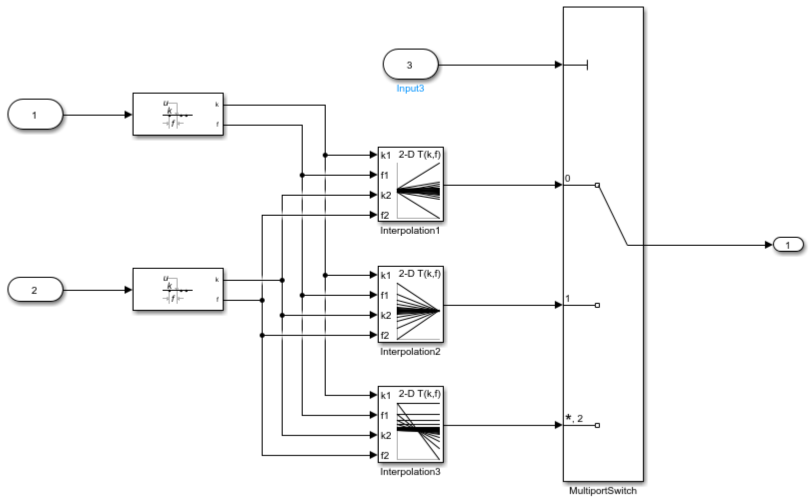
Optimized code for models containing multiple Interpolation Using Prelookup blocks

Starting in R2021b, the code generator eliminates redundant data copies from the code generated for models containing multiple Interpolation Using Prelookup blocks when these conditions are true:

- The inputs of the Interpolation Using Prelookup blocks are from the same sources.
- The Interpolation Using Prelookup blocks have identical block parameter settings except they have different **Table data** values specified, on the **Main** tab, in the **Value** text box.
- The data type of the specified table data matches the **Table data** type specified on the **Data Types** tab.
- The blocks are of the same variant and have the same sample times.
- No lookup table object is used in the blocks.
- The output signals of the blocks are not branched, logged, or resolved to signal objects.
- The model configuration parameter **Use memcpy for vector assignment** is selected, and the table size (in bytes) exceeds the specified **Memcpy threshold (bytes)**.

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

Consider the model `mCommonSrceInterp`.



The model contains three Interpolation Using Prelookup blocks that take inputs from the `Prelookup1` and `Prelookup2` blocks. The outputs of the Interpolation Using Prelookup blocks connect to a Multiport Switch block. This model satisfies the preceding the conditions.

In R2021a, the code generator produced this code:

```
/* Model step function */
void mCommonSrcInterp_step(void)
{
  real_T frac[2];
  real_T rtb_Prelookup1_o2;
  real_T rtb_Prelookup2_o2;
  uint32_T bpIndex[2];
  uint32_T rtb_Prelookup1_o1;
  uint32_T rtb_Prelookup2_o1;

  /* PreLookup: '<Root>/Prelookup1' incorporates:
   *  Inport: '<Root>/Input1'
   */
  rtb_Prelookup1_o1 = plook_binx(mCommonSrcInterp_U.Input1,
    mCommonSrcInterp_ConstP.Prelookup1_BreakpointsData, 1U, &rtb_Prelookup1_o2);
    .....
  switch ((int32_T)mCommonSrcInterp_U.Input3) {
   case 0:
    /* Interpolation_n-D: '<Root>/Interpolation1' */
    frac[0] = rtb_Prelookup1_o2;
    frac[1] = rtb_Prelookup2_o2;
    bpIndex[0] = rtb_Prelookup1_o1;
    bpIndex[1] = rtb_Prelookup2_o1;
    .....
    mCommonSrcInterp_Y.Output8 = intrp2d_l_pw(bpIndex, frac,
      mCommonSrcInterp_ConstP.Interpolation1_Table, 2U);
    break;
    ....
```

The code contained temporary variables `rtb_Prelookup1_o2`, `rtb_Prelookup2_o2`, `rtb_Prelookup1_o1`, and `rtb_Prelookup2_o1` to hold the outputs of the `Prelookup1` and

Prelookup2 blocks that passed to the individual Interpolation Using Prelookup blocks as inputs through assignment operations.

In R2021b, the code generator produces this code:

```
/* Model step function */
void mCommonSrcInterp_step(void)
{
  real_T frac[2];
  const real_T *rtb_MultiportSwitch_TableData_0;
  uint32_T bpIndex[2];
  /* Interpolation_n-D: '<Root>/Interpolation3' incorporates:
    ...
   */
  bpIndex[0] = plook_binx(mCommonSrcInterp_U.Input1,
    mCommonSrcInterp_ConstP.Prelookup1_BreakpointsData, 1U, &frac[0]);
  bpIndex[1] = plook_binx(mCommonSrcInterp_U.Input2,
    mCommonSrcInterp_ConstP.Prelookup2_BreakpointsData, 20U, &frac[1]);
   .....
  switch ((int32_T)mCommonSrcInterp_U.Input3) {
   case 0:
    rtb_MultiportSwitch_TableData_0 =
      &mCommonSrcInterp_ConstP.Interpolation1_Value[0];
    break;
     ....
  mCommonSrcInterp_Y.Output8 = intrp2d_l_pw(bpIndex, frac,
    rtb_MultiportSwitch_TableData_0, 2U);
```

The code generator does not generate the temporary variables rtb_Prelookup1_o2, rtb_Prelookup2_o2, rtb_Prelookup1_o1, and rtb_Prelookup2_o1 to hold the outputs of the Prelookup1 and Prelookup2 blocks. The code generator eliminates the unnecessary assignment operations that took place previously to pass the outputs to individual Interpolation Using Prelookup blocks. Now, the outputs of Prelookup1 and Prelookup2 blocks pass as function arguments in the highlighted code line.

Check bug reports for issues and fixes

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

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

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

R2021a

Model Architecture and Design

Code reuse across models for S-function inside library subsystems

In R2020b, you could place S-functions inside a reusable subsystem within a model. For S-functions that met certain requirements, you specified the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the `ssSetOptions` function. This flag indicated that your S-function met the requirements for subsystem code reuse.

In R2021a, you can place S-functions inside a reusable library subsystem and reuse the subsystem within a model and across the model reference hierarchy. The code generator produces the code for the S-functions in the `slprj\ert\_sharedutils` folder.

To reuse S-functions within a model, set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag in the `ssSetOptions` function.

To reuse S-functions across models, in the `ssSetOptions` function, set these flags in the S-function file:

- In the `ssSetOptions` function, set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag.
- Set `ssSetSupportedForCodeReuseAcrossModels` to `1` or `true`

For example:

```
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetOptions(S,
                 SS_OPTION_WORKS_WITH_CODE_REUSE |
                 SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_DISALLOW_CONSTANT_SAMPLE_TIME);
    ssSetSupportedForCodeReuseAcrossModels(S, 1);
}
```

If you use the `legacy_code` function to configure your S-function for code reuse, use the new S-function option `supportCodeReuseAcrossModels`.

If your S-function uses custom functions defined in your external header files, add the `LibAddtoSystemCustomIncludes(system, incFileName)` function in the `.tlc` file of your S-function.

For more information, see S-Functions for Code Reuse.

Parameter name updated to copy code mappings

In the Model Reference Conversion Advisor, the name of the **Copy code mapping information to the new converted model** option is now **Copy code mappings**. For more information, see Convert Subsystem to Referenced Model and Generate Code.

Code Interface Configuration and Integration

Invalid values detection after changing enum definitions in custom system target file

Previously, changing an `enum` definition in a custom system target file sometimes caused models to have invalid settings and revert to the default values for those settings. A model had an invalid setting if you:

- Saved a model with a specific value for the `enum` in your custom system target file.
- Edited your custom system target file by changing or removing the value from the `enum` definition.
- Loaded the model using the new version of the system target file.

In R2021a, when you change an `enum` definition in a custom system target file and load a model, a warning indicates if the model used a value that is no longer valid for the `enum`. The enhanced warning shows the invalid value and the corrected value. To enable the enhanced warning, when you change an `enum` definition, change the property `rtwgensettings.Version` to a different value. For example, change `rtwgensettings.Version` from `'1'` to `'1.1'`. For more information about custom TLC files, see Target Language Compiler Basics.

Calibration file generation

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

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

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

Code configuration for data dictionary defaults

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

ASAP2 system target file being removed
Warns

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

Functionality being removed or changed

You can no longer use the **ASAP2 interface** configuration parameter to generate code for the ASAP2 data interface. Use the **Generate Calibration Files** tool to generate ASAP2 related code and ASAP2 file.

Code Generation

simstruc_types.h not generated in rtwtypes.h

In R2020b, for non-ERT system target files, the code generator included the `simstruc_types.h` file in the generated `rtwtypes.h` file.

In R2021a, for non-ERT system target files, the code generator does not include the `simstruc_types.h` file in the generated `rtwtypes.h` file.

For more information, see Manage Build Process File Dependencies.

Target hardware data management

R2021a provides these `target` package enhancements for target hardware data management.

Function	Enhancement
`target.remove`	If you specify the name-value argument, `'IncludeAssociations', true`, the function removes the specified target object and associated objects from an internal database. The function does not remove an associated object if it is referenced by other target objects. The function displays information about the removed objects, which you can suppress by using the name-value argument, `'SuppressOutput', true`. For more information, see `target.remove`.
`target.add`	The function displays information about the objects that it adds to an internal database. The function also returns a vector that contains the added objects. You can suppress the text output by using the name-value argument, `'SuppressOutput', true`. For more information, see `target.add`.
`target.export`	When you run the function generated by `target.export`, it returns the registered target object and associated target objects. Previously, the function did not return associated target objects. For more information, see `target.export`.
`target.create`	Using name-value arguments, you can create an object and specify properties in a single step for these classes: • `target.Timer` • `target.Command` • `target.TargetConnection` Previously, multiple steps were required.

Functionality being removed or changed

Compatibility Considerations

rtwreport function will be removed
Warns

The function rtwreport will be removed in a future release. Use coder.report.generate instead.

To update your code, change instances of the function name rtwreport to coder.report.generate. You do not need to change the input arguments.

Unlike the rtwreport function, the coder.report.generate function provides additional input options that you can use to configure the generated report. However, the coder.report.generate function does not include snapshots of the model or the block execution order list.

RTW.HWDeviceRegistry is not supported
Errors

Simulink Coder no longer supports the RTW.HWDeviceRegistry class. If you use the RTW.HWDeviceRegistry class in rtwTargetInfo.m and sl_customization.m files to register hardware devices, the software produces an error. To upgrade existing definitions of hardware devices, you can use the target.upgrade function. For more information, see Upgrade Data Definitions for Hardware Devices.

Deployment

Target connectivity customization for external mode simulations

Using the `target` package, you can customize target connectivity for external mode simulations. MATLAB registration of the customized target connectivity is simple. You can customize the XCP transport layer, the XCP platform abstraction layer, and tools for automatic deployment. You can reuse the hardware board definition for processor-in-the-loop (PIL) simulations.

R2021a provides these new classes:

- `target.ExternalMode`
- `target.ExternalModeConnectivity`
- `target.XCP`
- `target.XCPExternalModeConnectivity`
- `target.XCPPlatformAbstraction`
- `target.XCPSerialTransport`
- `target.XCPTCPIPTransport`
- `target.XCPTransport`

For more information, see Set Up Connectivity Between Simulink and Target Hardware.

XCP external mode simulation on big endian target hardware

You can run XCP-based external mode simulations on target hardware that uses a big endian architecture. For more information, see External Mode Simulation by Using XCP Communication.

Build commands slbuild and rtwbuild unified

R2021a unifies the `slbuild` and `rtwbuild` commands. Use `slbuild` to:

- Build models and subsystems.
- Build multiple subsystems by specifying subsystem names in a cell array.

The `rtwbuild` command is now an alias for `slbuild`.

Compiler default language standards to compile code

Previously, when you used a Linux GCC compiler, the software added compiler flags to enforce the specified language standard in the parameter **Standard math library** on the compilation process. This enforcement restricted building custom code that did not conform with the version of the language standard. For instance, if you set this parameter to `C++03`, while building the generated code that integrated custom code with C++11 features, you got a compilation error.

In R2021a, when you use a Linux GCC compiler, the software does not enforce the language standard specified in the parameter **Standard math library** on the compilation process. Instead, during compilation, it uses the compiler default language standard. Depending on the version of the GCC compiler, now you can build the generated code that integrates custom code or libraries that uses higher language standard features compared to the language standard specified in the parameter. For more information, see Standard math library.

packNGo for CMake configuration files

In R2021a, you can use the `packNGo` function to package CMake configuration files (`CMakeLists.txt`) that are generated by the `codebuild` function. When you run the function, you must specify the name-value argument `'packType','hierarchical'`. For example:

```
packNGo(buildFolder, ...
        'packType', 'hierarchical', ...
        'nestedZipFiles', false);
```

After unpacking the ZIP file in another development environment, you can use CMake to:

- Generate makefiles or projects.
- Build binary files.

For more information, see Compile Code in Another Development Environment.

Functionality being removed or changed

Rapid simulation (RSim) system target file will be removed
Still runs

The rapid simulation (RSim) system target file (`rsim.tlc`) will be removed in a future release. The RSim system target file provides capabilities for characterizing model behavior as described in Accelerate, Refine, and Test Hybrid Dynamic System on Host Computer by Using RSim System Target File. Instead of using the RSim system target file:

- To speed up simulations, use the Simulink Rapid Accelerator simulation mode. Rapid Accelerator simulation mode provides the same functionality as the RSim system target file, but is easier to use. See Design Your Model for Effective Acceleration.
- To deploy standalone simulations outside of the MATLAB and Simulink environment, use the Simulink Compiler™. The Simulink Compiler provides a turnkey solution for building and sharing simulations as standalone executables that package a compiled Simulink model with MATLAB code that sets up, runs, and analyzes model simulations. See Comparing Simulink Coder and Simulink Compiler (Simulink Compiler).

Use of Scope Viewers and Floating Scope blocks in referenced models during XCP external mode simulations
Errors

You cannot use Scope Viewers and Floating Scope blocks to monitor signals in referenced models during XCP-based external mode simulations. To monitor referenced model signals, enable signal logging and use the Simulation Data Inspector. For more information, see External Mode Simulation by Using XCP Communication.

Performance

Reduced zero initialization code

In R2020b, when you defined large and complex structures that needed to be initialized to zero, the code generator initialized a `rtz*` global variable that had zero values. This variable increased the memory footprint of the generated code and code generation time. For example:

```
const Y testModel_rtZY = {
   {
      {
         0.0,                          /* a */
         0.0                           /* c */
      }, {
         0.0,                          /* a */
         0.0                           /* c */
      }, {
         0.0,                          /* a */
         0.0                           /* c */
      }
    ...
   ,                                   /* A */
   0.0,                                /* B */
   0.0                                 /* C */
} ;
...
/* Model step function */
void testModel_step(void)
{
   Y rtb_BusAssignment;

   /* BusAssignment: '<Root>/Bus Assignment' incorporates:
    *  Constant: '<Root>/Constant'
    *  Inport: '<Root>/b1'
    *  Inport: '<Root>/c1'
    */
   rtb_BusAssignment = testModel_rtZY;
   rtb_BusAssignment.B = testModel_U.b1;
   rtb_BusAssignment.C = testModel_U.c1;

   /* Outport: '<Root>/Outport' */
   testModel_Y.Outport = rtb_BusAssignment;
}
```

In R2021a, when you define complex data structures that need to be initialized to zero, the code generator uses the `memset` function to initialize a variable that has zero values in the *`model`*`_step` function. The code generator does not use the `memset` function when you define types that do not have a representation for a zero value, for example, `int16`, a fixed-point data type that has slope and bias configured by using nonzero values.

```
/* Model step function */
void testModel_step(void)
{
   Y rtb_BusAssignment;

   /* BusAssignment: '<Root>/Bus Assignment' incorporates:
    *  Inport: '<Root>/b1'
    *  Inport: '<Root>/c1'
    */
   (void)memset(&rtb_BusAssignment, 0, sizeof(Y));
   rtb_BusAssignment.B = testModel_U.b1;
   rtb_BusAssignment.C = testModel_U.c1;

   /* Outport: '<Root>/Outport' */
   testModel_Y.Outport = rtb_BusAssignment;
}
```

For more information, see Optimize Generated Code Using `memset` Function.

Check bug reports for issues and fixes

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

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

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

R2020b

Model Architecture and Design

Automatically package protected models with their dependencies

When you create a protected model, you can now automatically package it with its dependencies and a harness model in a project archive. When the recipient extracts the contents of the project archive and opens the harness model, they should be able to simulate the protected model without needing to define missing variables or objects. Before sharing the project archive, check whether the project contains all of the necessary supporting files, and update the harness model as needed.

In the Create Protected Model dialog box, set **Contents** to `Protected model (.slxp) and dependencies in a project`. For **Name of project archive (.mlproj)**, use the default name or specify a name. The project inside the project archive uses the same name.

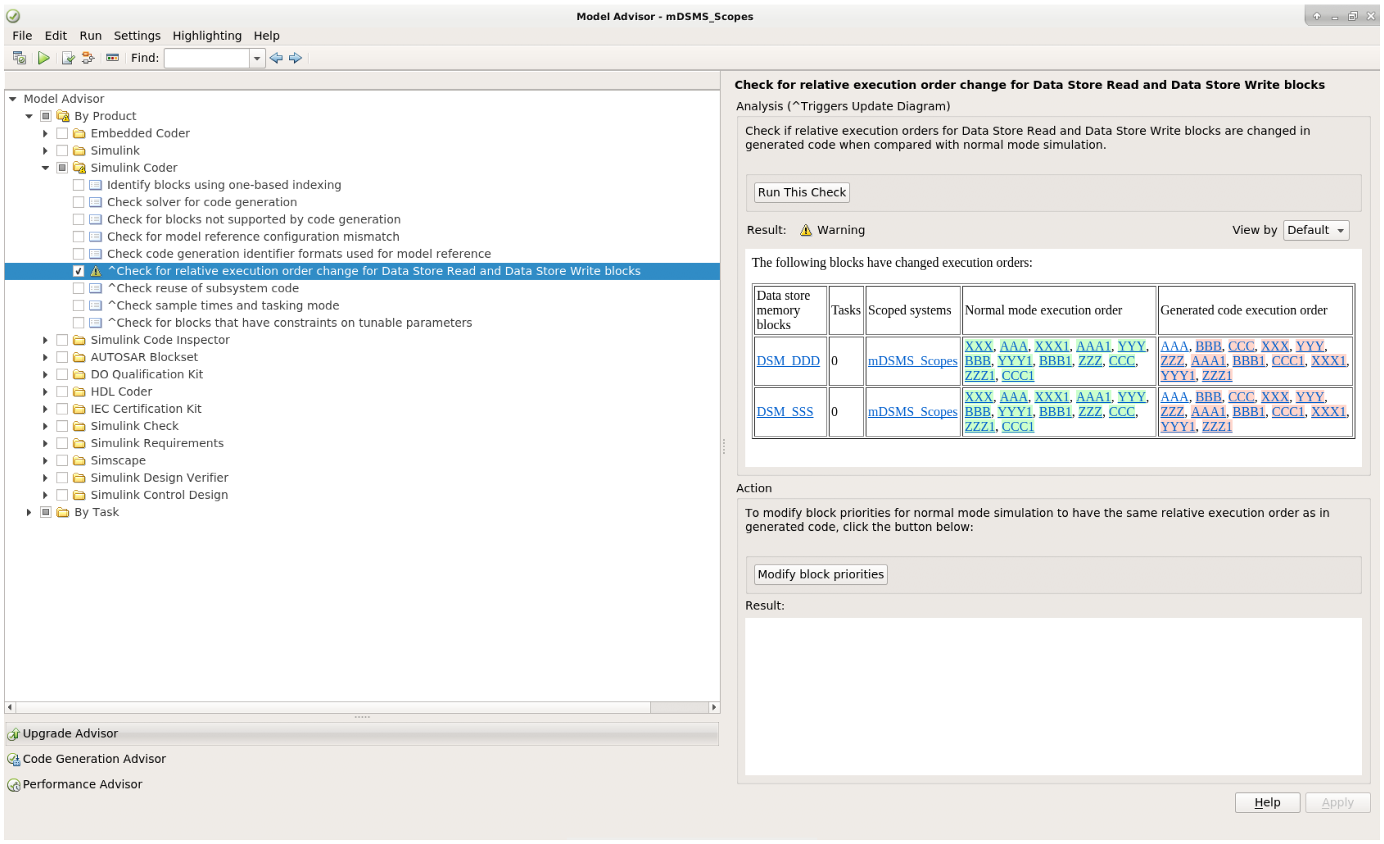
Alternatively, use the `Simulink.ModelReference.protect` function with the comma-separated pair consisting of `'Project'` and `true`. To specify a project name, also use the comma-separated pair consisting of `'ProjectName'` and the desired name specified as a character vector. If you do not specify a project name, the function uses the default name.

For more information, see Package and Share Protected Models.

Execution order check for Data Store Memory blocks

Use the Model Advisor check Check for relative execution order change for Data Store Read and Data Store Write blocks (check ID `com.mathworks.sorting.datastoresimrtwcmp`) to verify that the execution order of the Data Store Read and Data Store Write blocks in normal (simulation) mode does not change when the model is compiled for code generation.

When there are differences in the execution order, the check issues a **Warning** and identifies the discrepancies in the results. You can correct the issue by clicking the Model Advisor action button **Modify block priorities**. The Model Advisor updates the blocks so that the execution order in the normal (simulation) mode matches the order in the code generation mode.


Model Advisor - mDSMS_Scopes
File Edit Run Settings Highlighting Help
Find:
Model Advisor
By Product
Embedded Coder
Simulink
Simulink Coder
Identify blocks using one-based indexing
Check solver for code generation
Check for blocks not supported by code generation
Check for model reference configuration mismatch
Check code generation identifier formats used for model reference
^Check for relative execution order change for Data Store Read and Data Store Write blocks
^Check reuse of subsystem code
^Check sample times and tasking mode
^Check for blocks that have constraints on tunable parameters
Simulink Code Inspector
AUTOSAR Blockset
DO Qualification Kit
HDL Coder
IEC Certification Kit
Simulink Check
Simulink Requirements
Simscape
Simulink Design Verifier
Simulink Control Design
By Task
Upgrade Advisor
Code Generation Advisor
Performance Advisor
Run This Check
View by Default
Modify block priorities
Help
Apply

Check for relative execution order change for Data Store Read and Data Store Write blocks

Analysis (^Triggers Update Diagram)

Check if relative execution orders for Data Store Read and Data Store Write blocks are changed in generated code when compared with normal mode simulation.

Result: ⚠ Warning

The following blocks have changed execution orders:

Data store memory blocks	Tasks	Scoped systems	Normal mode execution order	Generated code execution order
DSM_DDD	0	mDSMS_Scopes	XXX, AAA, XXX1, AAA1, YYY, BBB, YYY1, BBB1, ZZZ, CCC, ZZZ1, CCC1	AAA, BBB, CCC, XXX, YYY, ZZZ, AAA1, BBB1, CCC1, XXX1, YYY1, ZZZ1
DSM_SSS	0	mDSMS_Scopes	XXX, AAA, XXX1, AAA1, YYY, BBB, YYY1, BBB1, ZZZ, CCC, ZZZ1, CCC1	AAA, BBB, CCC, XXX, YYY, ZZZ, AAA1, BBB1, CCC1, XXX1, YYY1, ZZZ1

Action

To modify block priorities for normal mode simulation to have the same relative execution order as in generated code, click the button below:

Result:

To execute the check, open the Model Advisor and browse to the **By Product** > **Simulink Coder** folder.

Code Interface Configuration and Integration

Streamlined model data configuration for code generation

R2020b simplifies how you configure model data, such as block data and signal lines, for code generation. Without affecting your model design configuration, from the Code Mappings editor in the Simulink Coder app and code mappings API, you can configure default settings for categories of data. Then, override those settings, as needed, for specific data elements. When producing code for data, the code generator uses storage classes that you specify to determine properties, such as whether to generate code that reads from and writes to a global variable or global variable pointer defined by external code.

You can use the Code Mappings editor to map an individual model data element to:

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

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

Code mappings also enable you to associate a model with code configurations for multiple platforms.

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

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

Migration of Preexisting Models

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

- Inport blocks
- Outport blocks
- Signal lines
- Block states

- Data stores
- Parameter objects in the model workspace

Simulink does not migrate data that is configured for code generation in external sources, such as the base workspace or a data dictionary.

For information about code mappings, see C Code Generation Configuration for Model Interface Elements, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

Compatibility Considerations

The code mappings interfaces for configuring data are compatible withh common data configuration scenarios from previous releases of Simulink Coder software.

You can work around many of the incompatibilities by developing MATLAB scripts to run in Simulink Coder. For more information, see “Functionality being removed or changed” on page 6-5 and Migration of Model Data Configurations to Code Mappings.

Functionality being removed or changed

The new code mappings interfaces streamline how you configure model data elements for code generation. These interfaces introduce:

- Incompatibilities with uncommon data configuration scenarios from previous releases of Simulink Coder software.
- Changes for the use of other Simulink interfaces for configuring data, such as the Model Data Editor, the Model Explorer, and the Signal Properties dialog box.

Simulink interface changes for data configuration
Still runs

In R2020b, the Code Mappings editor is the primary location to configure model data elements for code generation.

- In the Model Data Editor, the `Code` view has been removed. The editor does not display a **Code** section in the Property Inspector.
- You can no longer configure code generation properties in the Signal Properties dialog box.
- For `Simulink.Signal` objects in the model workspace, you can no longer configure code generation properties in the Model Explorer or in the property dialog box. To configure these elements for code generation, use the Code Mappings editor or code mappings API.
- For data objects in the model workspace other than `Simulink.Signal` objects, where previously you could configure code generation properties in the Model Explorer or in the property dialog box, links or buttons take you to the Code Mappings editor instead.
  - In the Model Explorer, in the **Contents** pane, click the **Configure** link in the **Storage Class** column.
  - In the Model Explorer **Dialog** pane and in the property dialog box, on the **Code Generation** tab, click **Configure in Coder App**.

For more information see, C Code Generation Configuration for Model Interface Elements, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

TypeQualifier property for built-in storage classes no longer used for data objects

You can no longer use the `TypeQualifier` property for built-in storage classes, such as `ExportedGlobal` and `ImportedExtern`, is no longer supported for use with data associated with data objects because more robust mechanisms are available for achieving the same results. In previous releases, when you specified the property, the code generator added C qualifiers, such as `const` and `volatile`, to the beginning of data declarations and definitions. You might have set this property as:

- `CoderInfo.TypeQualifier` property for data objects in a workspace or data dictionary
- Port parameter `RTWStorageTypeQualifier`
- Block parameter `RTWStateStorageTypeQualifier` for Data Store Memory, Discrete Filter, Discrete State-Space, Discrete-Time Integrator, Discrete Transfer Fcn, Discrete Zero-Pole, and Memory blocks

To address this change in an existing model that uses the `TypeQualifier` property, open the model in a release before R2020b. Create and run a MATLAB script that loads the data for the model from a workspace or data dictionary, finds data objects that have the `TypeQualifier` property set to a nonempty string value, and changes the storage class setting to a storage class predefined with the required type qualifier (for example, storage class `Const` includes qualifier `const` in data declarations and definitions). For an example, see Migration of Model Data Configurations to Code Mappings.

Starting in R2020b, if you have Embedded Coder software, use the Code Mappings editor or code mappings API to associate data elements with a storage class that specifies a C qualifier (see Choose Storage Class for Controlling Data Representation in Generated Code). If none of the available storage classes meets your application requirements, define a new storage class by using the Embedded Coder Dictionary (see Define Storage Classes, Memory Sections, and Function Templates for Software Architecture (Embedded Coder)) . Then, use the **Code Mappings Editor** or code mappings API (`coder.mapping.api.CodeMapping`) to associate the model data to the new storage class.

Code configuration for parameter objects initialized in model workspace from external data sources moved to code mappings

Starting in R2020b, the code mappings interface enables you to associate a model with multiple code generation configurations for data. When you load a model created in a previous release of Simulink and the model workspace is initialized from an external data source, such as a MAT-file, Simulink moves the code configuration for the parameter object to the code mappings for that model.

Once the configuration for data elements in a model has been converted to code mappings, use the Code Mappings editor or the code mappings API to get and set parameter code configuration settings. See C Code Generation Configuration for Model Interface Elements, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

Copies of blocks or signal lines between models no longer include code configuration

Starting in R2020b, when you use the Simulink Editor to copy a block or signal line to another model, Simulink does not copy the code configuration associated with the copied modeling element. The contents of the `Simulink.CoderInfo` object for the copied modeling element is removed. This change:

- Eliminates unnecessary copies of code configuration information for data configured within the model.
- Supports unique code configuration of data elements for a model and its active system target file.
- Promotes reuse of modeling patterns across models that have different code configurations.

To copy the code configuration information associated with a block or signal line, use the code mappings API. For an example, see Migration of Model Data Configurations to Code Mappings. For information about the API, see `coder.mapping.api.CodeMapping`.

Code configuration for data configured within a model removed for library models

Starting in R2020b, for you open a model created in a previous release, Simulink ignores the code configuration for data elements for library models. Reconfigure code generation for data in the context of models that use the library (see C Code Generation Configuration for Model Interface Elements).

This change does not apply to data objects saved in the base workspace or a data dictionary.

To avoid losing the code configuration for data, in an earlier release, create and run a MATLAB script that migrates the model to use external data objects. For an example, see Migration of Model Data Configurations to Code Mappings.

Default (Custom) storage class removed

To prevent confusion with the concept of default code configurations that you can set up by using the Code Mappings editor or code mappings API, you can no longer use the `Default (Custom)` storage class for data configured within a model. The storage class is not recommended and will not be in a future release for global data (data configured in the base workspace or a data dictionary).

For models created in R2020a or earlier, the storage class for a data element is set to `Default (Custom)` when these conditions exist:

- The `StorageClass` property for the `Simulink.CoderInfo` object is set to `Custom`.
- The `CustomStorageClass` property for the `Simulink.CoderInfo` object is not modified or is explicitly set to `Default`.

For an Outport block, signal line, block state, data store, or model workspace parameter set to `Default (Custom)` , when you load the model, Simulink converts the storage class setting to `ExportedGlobal` and displays a warning about the change. `ExportedGlobal` is equivalent to `Default (Custom)`.

Starting in R2020b, use the Code Mappings Editor or code mappings API to specify default code generation configurations for categories of data elements. See C Code Generation Configuration for Model Interface Elements, **Code Mappings Editor**, and `coder.mapping.api.CodeMapping`.

Changing between GRT-based and ERT-based system target file
Behavior change

Starting in R2020b, when you change the system target file setting for a model between a GRT-based and ERT-based system target file, Simulink applies an alternative code configuration for each system target file.

A change between system target files can occur if you:

- Alternate between the Simulink Coder and Embedded Coder app.
- Change the active configuration set for a model.
- Change the setting of model configuration parameter **System target file**.

It is a best practice to configure data differently for a model depending on whether you are generating rapid-prototyping (GRT) or production (ERT) code. Simulink associates the code configuration with the system target file so that you can set up multiple code configurations for a model.

To copy code mappings when you switch system target files, create and run a MATLAB script that uses the code mappings API to copy relevant code mappings. For an example, see Migration of Model Data Configurations to Code Mappings. For information about the API, see `coder.mapping.api.CodeMapping`.

Simulink.CoderInfo object Alignment property for data configured within a model removed

The `Simulink.CoderInfo` object property `Alignment` for data configured for code generation within a model has been removed, including data represented by:

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

To use the `Alignment` property, represent data by using data objects outside of the model. For an example, see Migration of Model Data Configurations to Code Mappings.

APIs for controlling data interfaces
Still runs

In R2020b, code generation information for data objects configured within a model migrates from data objects to the mapping infrastructure. This change might affect existing scripts that you use to manage the code configuration for these data objects.

- You do not need to update scripts that use existing functions to interact with data objects configured for code generation within a model. When you get and set code generation information by using one of these functions, data objects now communicate with the mapping to maintain the mapping as the single source for this information. This information includes functions such as `assignin`, `evalin`, `getVariable`, `get_param`, `set_param`, and `isequal`.

  For example, these workflows do not require updates:

  - Getting the handle of a data object in the model workspace.

    ```
    mws = get_param('modelname', 'modelworkspace');
    objHandle = mws.getVariable('Param');
    ```
  - Evaluating an expression in the context of the model workspace.

```
mws.evalin("param=Simulink.Parameter; param.CoderInfo.StorageClass='ExportedGlobal';")
```

  - Specifying code generation settings for a signal object stored on a port.

    ```
    portHandles = get_param('blkPath', 'PortHandles');
    get_param(portHandles.Outport, 'StorageClass');
    set_param(portHandles.Outport, 'StorageClass', 'ExportedGlobal');
    ```
  - Specifying code generation settings for a root outport block.

    ```
    get_param('blkPath', 'StorageClass');
    set_param('blkPath', 'StorageClass', 'ExportedGlobal');
    ```
  - Comparing data objects.

    ```
    p1 = Simulink.Parameter;
    p1.CoderInfo.StorageClass = 'Custom';
    p1.CoderInfo.CustomStorageClass = 'ExportToFile';
    p2 = copy(p1);
    isequal(p1, p2); % Returns true
    p1.CoderInfo.StoragClass = 'ExportedGlobal';
    isequal(p1, p2); % Returns false
    ```
- The function `copy` can no longer copy the code generation properties of data objects configured for code generation within a model. You can instead use the new `clone` function to create a copy of an object with its code generation properties. You can use the `clone` function only to create a copy in the same workspace as the source object.

You can still use of existing command-line functions for configuring data objects for code generation within a model continues, but it is recommended that you use the new code mappings API instead. The code mappings API:

- Provides one programming interface for configuration of default code generation settings for categories of data and individual data elements.
- Supports multiple configuration mappings for a model.
- Eliminates the need to create data objects to configure model data elements for code generation.

For information about the new code mappings API, see `coder.mapping.api.CodeMapping`.

ASAP2 system target being removed
Still runs

Support for **asap2.tlc** system target will be removed in future release. Use grt.tlc system target file with model configuration parameter **ASAP2 Interface** selected instead. For more information, see Generate an ASAP2 File.

Code Generation

Code descriptor information for tunable breakpoint set data in Lookup table blocks

In R2020a, the `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects were created only if parameters in the lookup table data and breakpoint set data were tunable.

In R2020b, the `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects are created if these conditions are true:

- Lookup table data is tunable.
- One of these conditions is true:
  - Breakpoint set data is tunable.
  - Breakpoint set data is nontunable and the block does not use a `Simulink.LookupTable` object.
  - The block uses a `Simulink.LookupTable` object.

If the breakpoint set data is nontunable, you can obtain additional information regarding whether the breakpoint set data is evenly spaced or not by accessing a new object under the `coder.descriptor.FixAxisMetadata` class and these subclasses:

- `coder.descriptor.EvenSpacingMetadata`
- `coder.descriptor.NonEvenSpacingMetadata`

Code generation using C++11 standard math library

In R2020b, you can generate C++ libraries and executables for Simulink models that use the C++11 (ISO) or ISO®/IEC 14882:2011(E) standard math library. To generate code by using C++11 (ISO) math libraries, set the configuration parameter **Language** to `C++` and set the configuration parameter **Standard math library** to `C++11 (ISO)`.

For more information, see "Configure Standard Math Library for Target System" .

Clearer pattern of ordering of local variable declarations

In R2020a, the order of local variable declarations in the generated C/C++ code did not follow an obvious pattern. In R2020b, the grouping of local variable declarations in the generated C/C++ code is by type and in order of decreasing array size. For variables and array of the same type and size, the declarations are in alphabetical order.

For example, this table shows a sample of local variable declaration groupings in R2020a and R2020b. The R2020b declarations follow a clear pattern making it easier to locate a variable declaration.

Sample Local Variable Declarations in R2020a	Sample Local Variable Declarations in R2020b
`static boolean_T imgEdge[307200];`	`static real32_T b_img[307200];`
`static uint8_T b_img2[307200];`	`static real32_T img2[307200];`
`static uint8_T label[307200];`	`static real32_T imgRect[230400];`
`static real32_T imgRect[230400];`	`static uint8_T b_img2[307200];`
`static real32_T b_img[307200];`	`static uint8_T label[307200];`
`static real32_T img2[307200];;`	`static boolean_T imgEdge[307200];`
`int32_T b_r;`	`visioncodegen_BlobAnalysis_1 *obj;`
`int32_T c;`	`real_T cor[9];`
`int32_T i;`	`real_T b_B[2];`
`uint32_T q;`	`real_T ex;`
`real32_T thresh;`	`real_T pos;`
`real_T ex;`	`int32_T boundingbox_data[240];`
`real_T pos;`	`int32_T c_boundingbox_data[60];`
`boolean_T b_boundingbox_data[60];`	`int32_T b_r;`
`int32_T c_boundingbox_data[60];`	`int32_T c;`
`int32_T boundingbox_data[240];`	`int32_T i;`
`uint8_T v_data[1];`	`real32_T e_BW[400];`
`uint8_T label_data[58564];`	`real32_T f_BW[400];`
`uint32_T points1[8];`	`real32_T b_x[20];`
`real32_T tmp_data[8];`	`real32_T tform_T[9];`
`real32_T tform_T[9];`	`real32_T tmp_data[8];`
`real32_T b_x[20];`	`real32_T thresh;`
`real32_T e_BW[400];`	`uint32_T points1[8];`
`real32_T f_BW[400];`	`uint32_T q;`
`real_T b_B[2];`	`uint8_T label_data[58564];`
`real_T cor[9];`	`uint8_T v_data[1];`
`visioncodegen_BlobAnalysis_1 *obj;`	`boolean_T b_boundingbox_data[60];`

Enhanced status messages for code generation

In R2020b, when you build a model, you get more information about code generation status. You can observe the status messages at the bottom of the Simulink window and see the progress of the entire code generation process.

Removal of space in #define

In previous releases, the code generator inserted a space between `#` and `define` in the generated code. For example:

```
# define rtwdemo_comments_COMMON_INCLUDES_
```

In R2020b, the code generator removes the space between `#` and `define` in the generated code. For example:

```
#define rtwdemo_comments_COMMON_INCLUDES_
```

Query capability for target.get function

Use the `target.get` function to obtain a list of target feature objects that are saved in memory. You can refine the query to list only objects with properties that match specified name-value pairs.

Previously, you used this function only to retrieve a specified target feature object from memory.

Model Advisor Updates

A new Model Advisor check **Check reuse of subsystem code** (ID mathworks.codegen.SubsysCodeReuse) identifies CodeReuse Subsystem blocks that do not reuse code.

Functionality being removed or changed

rtwbuild and slbuild no longer generate model reference simulation targets by default
Behavior change

Starting in R2020b, the `rtwbuild` and `slbuild` functions do not generate model reference simulation targets by default. Excluding the model reference simulation targets allows for faster code generation for model hierarchies.

You can continue to generate the model reference simulation targets with the `rtwbuild` and `slbuild` functions by using the `IncludeModelReferenceSimulationTargets` argument.

Models referenced in normal mode that are loaded during code generation are closed
Behavior change

Starting in R2020b, models referenced in normal mode are closed after their code is generated unless they were loaded before code generation began. If a `CloseFcn` model callback deletes workspace objects needed by other models in the model hierarchy, you receive an error. To avoid errors during code generation, use data dictionaries, modify model callbacks, or load the affected referenced models before starting code generation. For more information, see “Model Callbacks”.

RTW.HWDeviceRegistry support will be removed
Warns

The `RTW.HWDeviceRegistry` class support will be removed in a future release. If you use the `RTW.HWDeviceRegistry` class in `rtwTargetInfo.m` and `sl_customization.m` files to register hardware devices, the software produces a warning. To upgrade existing definitions of hardware devices, you can use the `target.upgrade` function. For more information, see “Upgrade Data Definitions for Hardware Devices”.

Deployment

codebuild function for independent compilation of generated code

In R2020b, you can use the `codebuild` function to compile generated code on a different operating system or by using a different compiler. As compilation is independent of code generation, custom toolchain and template makefile testing is simpler. You can create custom toolchains and template makefiles more quickly.

Use this workflow:

1. Generate code on your development computer and use `packNGo` to package the code.
2. Relocate and unpack the code in the new environment.
3. Run `codebuild`, specifying the compiler through the `BuildMethod` argument. The function builds the generated code by using the generated code description and build configuration from the relocated `buildInfo.mat` file.

For more information, see Compile Code in Another Development Environment.

Configuration files for CMake build system

R2020b supports CMake, an open-source tool for build process management. After generating code from a Simulink model or MATLAB code, use the `codebuild` function to generate `CMakeLists.txt` files, which are configuration files for the CMake build system. When you run CMake, it uses the configuration files to produce standard build files for your build environment, for example, makefiles, NinjaScript files, or Microsoft Visual Studio projects.

For more information, see Compile Code in Another Development Environment.

slbuild builds multiple models

In R2020b, you can build multiple models with one invocation of the `slbuild` command. If parallel building of referenced models is enabled for top models, `slbuild` performs the build process without the need for manual reinitialization of the parallel pool of MATLAB workers.

For more information, see Build Multiple Top Models and Reduce Build Time for Referenced Models by Using Parallel Builds.

Build Summary for Top Model and Referenced Models

When you build models, the build process provides a summary that helps you to find out for each model whether and why code is generated and compiled. The summary provides this information:

- List of models for which code is generated and compiled
- Reasons for rebuilding models
- Number of rebuilt models
- Duration of build

The build process displays the summary in the:

- Command Window, if you use line commands, for example, `slbuild` or `rtwbuild`.
- Diagnostic Viewer, if you use the Simulink Editor or press **Ctrl+B**.

For more information, see Build Multiple Top Models.

Parallel build continues after MATLAB worker stoppage

If you start a pool of MATLAB workers by running `parpool` (Parallel Computing Toolbox) with `'SpmdEnabled'` set to `false`, then if a worker stops working during parallel building of referenced models, the build process continues to run on the remaining workers in the parallel pool. Previously, when a worker stopped working, the entire pool of workers stopped. For information about building referenced models in parallel, see Reduce Build Time for Referenced Models by Using Parallel Builds.

Dynamic signal selection and triggering for XCP external mode simulations

For XCP external mode simulations, you can use the External Signal & Triggering dialog box to:

- Select logged signals that you want to monitor. Changing your selection of logged signals does not require rebuilding and redeployment of the target application.
- Configure a trigger that starts uploading of data from the target application. You can specify the trigger signal as a logged signal in the top model or a referenced model.

For more information, see:

- Triggered Signal Monitoring for XCP External Mode Simulations
- XCP External Signal & Triggering Dialog Box

XCP external mode simulation supports half-precision data type

In R2020b, through an XCP external mode simulation, you can monitor and tune half-precision data types within a real-time application. For example, you can:

- Use emulated half-precision data types in the application.
- Stream signals from the application to the Simulation Data Inspector and scope blocks and viewers.

For information about running an external mode simulation, see External Mode Simulation by Using XCP Communication.

Intel C and C++ toolchain use with Windows

You can compile generated code by using Intel C and C++ compilers for Windows. In R2020b, you can use:

- Intel Parallel Studio XE 2020 with Microsoft Visual Studio 2017, 2019
- Intel Parallel Studio XE 2019 with Microsoft Visual Studio 2015, 2017, 2019
- Intel Parallel Studio XE 2018 with Microsoft Visual Studio 2015, 2017, 2019

Support for Intel Parallel Studio XE 2017 is removed.

For more information, see Supported Compilers.

Simulink Coder Support Package for NVIDIA Jetson CPUs: Generate, build, and deploy Simulink models on Jetson CPU

The Simulink Coder Support Package for NVIDIA® Jetson™ CPUs is available from release R2020b onwards. You can use the support package to generate, build, and deploy Simulink models on the Jetson CPU.

Functionality being removed or changed

Passing RTW.BuildInfo to a hook function

Behavior change

If you pass an `RTW.BuildInfo` object as an argument to a hook function, for example, `before_make`, the object content in R2020b differs from the object content in R2020a. The name of the link object no longer includes the library extension. This table shows two examples.

R2020a	R2020b

`>> buildInfo.ModelRefs(1).Name`
 	
`ans =`
 	
`    'refmodel_rtwlib.lib'`
	`>> buildInfo.ModelRefs (1).Name`
	
	`ans =`
	
	`    'refmodel_rtwlib'`

`>> buildInfo.LinkObj(1).Name`
 	
`ans =`
 	
`    'rtwshared.lib'`
	`>> buildInfo.LinkObj(1).Name`
	
	`ans =`
	
	`    'rtwshared'`

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

Performance

Faster code generation without generating model reference simulation targets

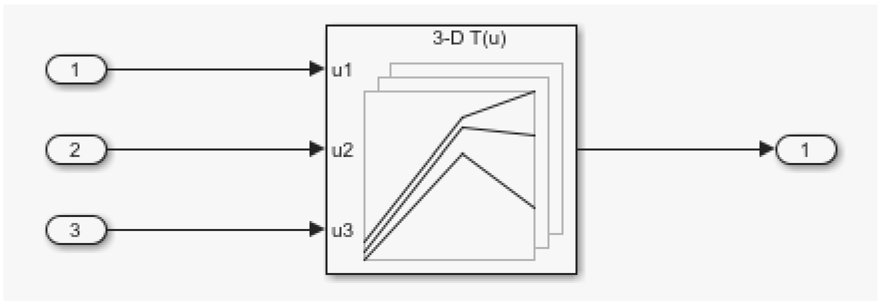
Code generation for model hierarchies is faster than in previous releases.

By default, code generation no longer generates the model reference simulation targets. For compatibility considerations, see “rtwbuild and slbuild no longer generate model reference simulation targets by default” on page 6-12.

Unused variables elimination for n-D Lookup Table blocks in generated code

In R2020a, the generated code for n-D Lookup Table blocks contained unused DWork vectors. In R2020b, you can generate code with some of the unnecessary DWork vectors eliminated. This optimization improves the efficiency of the generated code.

Consider this model `mLUT` that has a 3-D Lookup Table block.



In R2020a, the code generator produced this C code where the header file `mLUT.h` contained the DWork vectors `m_bpLambda` and `m_bpIndex` which were never used.

```
typedef struct {
  real_T m_bpLambda[3];              /* '<Root>/3-D Lookup Table' */
  uint32_T m_bpIndex[3];             /* '<Root>/3-D Lookup Table' */
} DW_mLUT_T;
```

In R2020b, the code generator produces code without the unused DWork vectors.

Verification

Check bug reports for issues and fixes

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

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

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

R2020a

Model Architecture and Design

Digital certificate signing for protected models

In R2020a, you can attach a digital signature to a protected model. To sign your protected model, use a PFX format digital certificate. When you share your protected model with a third party, they can verify that the model was signed by you and was not changed after you signed it. To sign your protected model, use these new functions.

Function	Description
`Simulink.ProtectedModel.sign`	Attach digital signature to protected model
`Simulink.ModelReference.ProtectedModel.setPasswordForCertificate`	Input password for certificate file

You can also use the new `'Sign'` argument of the `Simulink.ModelReference.protect` function to sign the model when you protect it. For more information, see Sign a Protected Model.

Rate Transition block deterministic mode support for concurrent execution

R2020a adds deterministic mode support to the Rate Transition block for models partitioned and configured for concurrent execution. You can configure a model for concurrency and configure Rate Transition blocks in that model so that the code generator produces code that transfers data predictably. To enable concurrent task execution, select the model configuration parameter **Allow tasks to execute concurrently on target**. To enable deterministic mode for a Rate Transition block, select the block parameter **Ensure deterministic data transfer (maximum delay)**. For example, if you are checking for a match between simulation and code generation numeric results for a model that uses concurrent task execution on multicore target hardware, select model configuration parameter **Allow tasks to execute concurrently on target** and Rate Transition block parameter **Ensure deterministic data transfer (maximum delay)**.

For more information, see Rate Transition block and Multicore Processor Targets (Simulink).

C/C++ message-based communication provides length argument for service functions

C/C++ message support now generates an additional parameter to specify message payload length in service functions. For more information, see Generate C or C++ Code for Message-Based Communication in Simulink.

C message-based communication defines service data types in one location in generated code

C code generated for message-based communication now defines service data types once in a shared folder accessible across multiple models. Previously, service types were defined in multiple locations in the generated code (the header file of each model in the message chain). For more information, see Generate C or C++ Code for Message-Based Communication in Simulink.

C/C++ message-based communication available for reusable subsystems

C/C++ message support is now available for reusable subsystems that contain Send or Receive blocks. Previously, message-based communication code generation was not supported for models that contained reusable subsystems with Send or Receive blocks. For more information, see Generate C or C++ Code for Message-Based Communication in Simulink.

Protect models for use with a Simulink license

In R2020a, when you protect a model that contains blocks that require another product license in addition to Simulink, using the protected model in a separate MATLAB session requires only a Simulink license. For more information about protecting models, see Protect Models to Conceal Contents.

Computer Vision Toolbox Interface for OpenCV in Simulink: Import OpenCV code into Simulink

The Computer Vision Toolbox™ Interface for OpenCV in Simulink support package enables you to import OpenCV code into a Simulink model. To install the support package, first click **Add-Ons** on the MATLAB **Home** tab. In the **Add-On** Explorer window, find and click the support package, and then click **Install**. This support package requires Computer Vision Toolbox. After installing the support package, you can import your OpenCV code and create Simulink library by using the **OpenCV Importer** app. The importer uses two OpenCV conversion blocks ToOpenCV and FromOpenCV. You can generate C++ code from the created Simulink model and deploy the code into your target hardware. For more information, see Install and Use Computer Vision Toolbox OpenCV Interface for Simulink (Computer Vision Toolbox).

Data, Function, and File Definition

Constant parameters outside const_params.c file

In R2019b, when the model configuration parameter **Generate shared constants** (`GenerateSharedConstants`) was cleared or set to `off`, the code generator defined the constant parameters in a nonshared area, in the `model_ert_rtw` folder in the `model_data.c` file.

In R2020a, when the model configuration parameter **Generate shared constants** is cleared or set to `off`, the code generator generates the constant parameters shared across the model reference hierarchy in a shared location, in the `slprj/`*`target`*`/_sharedutils` folder, but outside the `const_params.c` file. The code generator defines the individual constant parameters in the reusable library subsystem files individually.

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

Code Generation

Lookup Table blocks code description in generated code by using Code Descriptor API

To access information about lookup table blocks that have tunable parameters and tunable breakpoint set data, you can now retrieve a `coder.descriptor.LookupTableDataInterface` and a `coder.descriptor.BreakpointDataInterfacecoder.descriptor.BreakpointDataInterface` object. Use the `getDataInterfaces` method to retrieve information about these lookup table blocks in the generated code:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Interpolation Using Prelookup
- Direct Lookup Table (n-D)
- Sine
- Cosine

For example, create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescObj = coder.getCodeDescriptor(model)
```

Retrieve properties of the Lookup Table block and breakpoint set in the generated code.

```
params = getDataInterfaces(codeDescObj, 'Parameters')
```

`params` is an array of `coder.descriptor.LookupTableDataInterface` and `coder.descriptor.BreakpointDataInterface` objects.

The `coder.descriptor.LookupTableDataInterface` class has these methods:

- `isLookupTableDataInterface`
- `getAllParameters`

The `coder.descriptor.BreakpointDataInterface` class has these methods:

- `isBreakpointDataInterface`

Documentation for tlc() function being removed

Starting in R2020a, documentation for the `tlc` function is removed. Support for this function in Simulink Coder will be removed in a future release. The `tlc` function was used to invoke the Target Language Compiler (TLC) at the MATLAB command prompt. The TLC converted the model description file, *`model`*`.rtw`, into target-specific code or text.

Data interface type name changes in Code Descriptor API

In the Code Descriptor API, to match the parameter categories in the Code Mappings editor, these changes were made to the data interface type names:

- `GlobalParameters` is renamed to `ExternalParameterObjects`.
- `LocalParameters` is renamed to `ModelParameters`.

For more information, see `coder.descriptor.DataInterface`.

Deployment

Intel C and C++ toolchain support for Windows

You can compile generated code by using Intel C and C++ compilers for Windows. R2020a supports:

- Intel Parallel Studio XE 2017 with Microsoft Visual Studio 2015, 2017
- Intel Parallel Studio XE 2018 with Microsoft Visual Studio 2015, 2017
- Intel Parallel Studio XE 2019 with Microsoft Visual Studio 2015, 2017, 2019

For more information, see Supported Compilers.

Parallel pool automatically starts for parallel building of referenced models

If you have Parallel Computing Toolbox™ software and you select the **Enable parallel model reference builds** check box for the top model of a model reference hierarchy, you can run the build process without checking the status of the parallel pool of MATLAB workers. If the parallel pool is not running, the build process starts the parallel pool by using the default cluster profile.

Previously, you were required to check the status of the parallel pool and start the parallel pool if it was not running.

The build process no longer creates parallel build log files in reference model subfolders within the build folder.

- If you start the build process from the command line, the build process displays build log messages in the Command Window.
- If you start the build process from the Simulink Editor Code perspective, the build process displays build log messages in the Diagnostic Viewer.

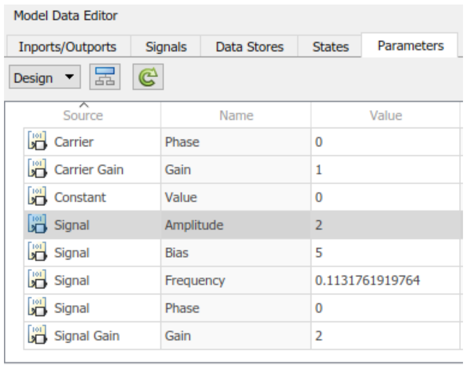
For more information, see Reduce Build Time for Referenced Models by Using Parallel Builds.

Simplified workflow for external mode Run on Custom Hardware app

The Run on Custom Hardware app, through the **Hardware** tab, provides enhancements that simplify the workflow for running external mode simulations and tuning parameters.

On the **Hardware** tab, the **Run on Hardware** section now displays the **Stop Time** field. Previously, the field was accessible only through a drop-down list.

During an external mode simulation, you can tune parameters by modifying their values in the **Parameters** tab of the Model Data Editor. To open the **Parameters** tab, in the **Prepare** section of the **Hardware** tab, click the **Tune Parameters** button.



Source	Name	Value
Carrier	Phase	0
Carrier Gain	Gain	1
Constant	Value	0
Signal	Amplitude	2
Signal	Bias	5
Signal	Frequency	0.1131761919764
Signal	Phase	0
Signal Gain	Gain	2

When you modify a value, Simulink immediately downloads the value to the target application. If you select **Batch Mode** in the **Prepare** section of the **Hardware** tab, you can modify multiple values, and then download the values simultaneously by clicking the **Update All Parameters** button.

In the **Prepare** section of the **Hardware** tab, you can download values of block parameters that are variables in the MATLAB workspace by clicking the **Update All Parameters** button. Previously, this functionality was provided only by pressing **CTRL+D**.

For more information, see External Mode Simulation by Using XCP Communication and External Mode Simulation with TCP/IP or Serial Communication.

Limit data quantity logged during XCP external mode simulation

Previously, XCP external mode simulations did not support controls that limited the amount of signal data logged by Simulink on your development computer. In R2020a, in the Signal Properties dialog box, on the **Logging and accessibility** tab, an XCP external mode simulation supports these options:

- **Limit data points to last**
- **Decimation**

For more information, see Logging and Accessibility Options (Simulink).

XCP external mode simulations on Mac computers

The XCP slave platform abstraction layer supports the Apple macOS operating system. You can now run host-based XCP external mode simulations on Mac computers.

For more information, see External Mode Simulation by Using XCP Communication.

Checksums determine whether object code is up to date

When you build a model, the software uses checksums to determine whether object code is up to date and recompilation of generated code is required. Previously, the software used file timestamps. For more information, see Use of Checksums in Recompilation of Generated Code .

Performance

Verification

Check bug reports for issues and fixes

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

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

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

R2019b

Model Architecture and Design

Generate C++ Code for Software Compositions with Message-Based Communication

R2019b introduces C++ and C code generation for message-based communication between Simulink model components using Messages & Events library Send, Receive, and Queue blocks. This release also introduces C++ code generation for message-based communication between Simulink top models and external message protocol services (middleware or operating systems).

For more information about code generation for message-based communication, see Generate C or C ++ Code for Message-Based Communication in Simulink and Generate C++ Code from Top Models for Message-Based Communication By Using External Message Protocols. For more information about the blocks, see Send, Receive, and Queue block descriptions.

Model integration with row-major data for MATLAB Function block and Stateflow charts

In R2019b, you can use row-major array layout in a MATLAB Function block to more easily and efficiently integrate your model with row-major data and algorithms. When you specify row-major array layout in the block, the layout applies to simulation and C/C++ code generation. To specify the array layout for the block, use the `coder.rowMajor` and `coder.columnMajor` functions. For more information, see Generate Row-Major Code for Model That Contains a MATLAB Function Block.

Row-major layout is now supported in Stateflow charts, state transition tables, and truth table blocks, including:

- Charts that use MATLAB as the action language.
- Charts that contain truth table functions and MATLAB functions.
- Charts that use custom C code where custom variables and arguments to custom functions are scalars or vectors.

For more details, see the “Enhanced support of row-major data in Stateflow blocks” (Stateflow) release note in Stateflow.

TLC block files and rtwmakecfg files for S-functions

Previous versions of the Legacy Code Tool generated a TLC block file and an `rtwmakecfg.m` file that checked the Simulink version number. These files issued an error when run in certain older releases. In R2019b only, using these files might result in an error when building the S-functions for which the files were generated.

Compatibility Considerations

To detect the affected files, use the Upgrade Advisor check **Check model for S-function upgrade issues**. To fix the affected files, click **Modify Files**. Or, regenerate the files by using the Legacy Code Tool.

For more information, see Import Calls to External Code into Generated Code with Legacy Code Tool.

Data, Function, and File Definition

Duplicate enumeration member names in generated code

In R2019a, Simulink Coder produced an error if you had duplicate enumeration member names in the generated code. In R2019b, you can control duplicate enumeration member names in the generated code. Use the configuration parameter **Duplicate enumeration member names** to generate an error or warning message or allow duplicate enumeration member names in different enumeration types during code generation. **Duplicate enumeration member names** is set to `error` by default. You can use duplicate enumeration member names only if two enumerations have the same `StorageType` and have these specifications:

- `DataScope` set to `'Imported'`
- `StorageType` set to `'int8'`, `'int16'`, `'int32'`, `'uint8'`, or `'uint16'`
- `Value` is the same

For example, consider these enumerations:

```
typedef int32_T enumA;
#define a       (0)
#define p       (1)

typedef int32_T enumB;
#define b       (0)
#define p       (1)
```

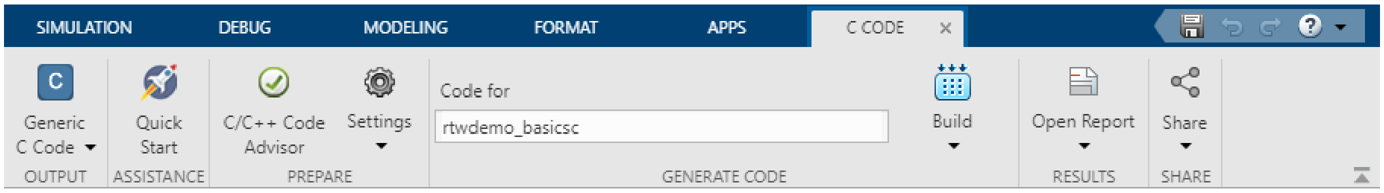
These enumerations have the same `int32` storage type. The enumeration member `p` with value `1` is the same for `enumA` and `enumB`. The configuration parameter **Duplicate enumeration member names** enables you to generate an error or warning message or allow code generation for duplicate enumeration member names. For more information, see Control Use of Duplicate Enumeration Member Names.

Code Generation

Simulink Coder contextual tabs on the Simulink Toolstrip

To assist you in your code generation workflow, use the Simulink Coder contextual tabs.

To access the **C Code** or the **C++ Code** tab, open the Simulink Coder app from the **Apps** gallery tab on the Simulink Toolstrip. To support common code generation workflow tasks, the tab provides Simulink Coder functionality corresponding to each task. The Simulink Coder app places the model in the Simulink Editor Code perspective. You can apply code generation settings for model elements in the Model Data Editor. The integrated help pane provides quick access to tools, video tutorials, and links to more information.



The Simulink Coder app supports models configured with GRT-based system target files. If you have not configured your model or model hierarchy with a GRT-based system target file, Simulink Coder prompts you to either open an app that supports your model's system target file or change your model's system target file to `grt.tlc`.

For more information, see “Simulink Toolstrip: Access and discover Simulink capabilities when you need them”.

Model configuration and code generation by using Simulink Coder Quick Start

The new Simulink Coder Quick Start tool helps you quickly generate code for rapid prototyping from your Simulink model.

First select preferences about your code generation output and objective. Then, the tool presents the parameter changes required to generate code. If you choose to generate code, the tool executes the changes to your configuration set and generates the code.

When code generation is complete, links to the documentation present possible next steps, such as preserving data in the generated code. To use the Simulink Coder Quick Start tool, open the Simulink Coder app and click **Quick Start**. For more information, see Generate Code by Using the Quick Start Tool.

Export of hardware device data

R2019b provides the `target.export` function, which enables you to share hardware device data across computers and users. For more information, see Export Hardware Device Data.

Data validation for hardware device features

R2019a introduced a new mechanism for registering hardware devices, which uses the target feature classes, `target.Processor` and `target.LanguageImplementation`. R2019b exposes `target.Object`, which is the base class for target feature classes. To validate the data integrity of objects that belong to target feature classes, use the `IsValid` property or `validate` method.

Consider an example where you create a `target.Processor` object and associate an existing language implementation with the object.

```
myProcessor = target.create('Processor');
myProcessor.LanguageImplementations = target.get('LanguageImplementation', ...
                                                 'ARM Compatible-ARM Cortex');
```

To validate the created object, run `myProcessor.IsValid` or `myProcessor.validate()`.

```
myProcessor.IsValid
```

```
ans =
  logical
  0
```

```
myProcessor.validate()
```

```
Error using target.Processor/validate
Target data validation failed.
* Undefined property "Name" in "Processor" object.
* Undefined identifier in "Processor" object.
```

The validation fails because these `target.Processor` properties are not specified:

- `Name` — Processor name
- `Id` — Object identifier

You can specify a processor name, which also specifies the object identifier.

```
myProcessor.Name = 'MyProcessor';
```

Check the validity of `myProcessor`.

```
myProcessor.IsValid
```

```
ans =
  logical
  1
```

The validity of the object is established.

When you use the `target.add` function to register a target feature object, the software also checks the validity of the object.

For more information, see Register New Hardware Devices.

Upgrade of hardware device definitions

R2019b provides the `target.upgrade` function, which enables you to upgrade existing definitions of hardware devices. The function uses a specific upgrade procedure to create objects from definitions in current data artifacts. By default, the function also creates the file, `registerUpgradedTargets.m`. To register the upgraded definitions, run `registerUpgradedTargets.m`.

For more information, see Upgrade Data Definitions for Hardware Devices.

Compatibility Considerations

Support for the use of `rtwTargetInfo.m` and `sl_customization.m` files to register hardware devices through the `RTW.HWDeviceRegistry` class will be removed in a future release. To update the registration mechanism, use the `target.upgrade` function.

Model Configuration Parameters Symbols pane renamed to Identifiers

In the Model Configuration Parameters dialog box, the **Code Generation** > **Symbols** pane has been renamed to **Code Generation** > **Identifiers**. For more information, see Model Configuration Parameters: Code Generation Interface.

Simulink cache file support for code generation artifacts

Simulink cache files now support code generation artifacts. The cached artifacts can reduce the time required for successive simulation and code generation. Caching occurs automatically when you simulate models in accelerator or rapid accelerator mode, or generate code for models. When Simulink cache files are available in the Simulation cache folder (Simulink), simulation and code generation automatically extract the relevant artifacts from the Simulink cache file. To share Simulink cache files with team members, you can store them in a network location.

For more information, see Simulink Cache Files for Incremental Code Generation.

Retrieve information about shared local data stores in the generated code by using Code Descriptor API

You can now retrieve a `coder.descriptor.DataInterface` object for shared local data stores in the generated code, in addition to inports, outports, parameters, global data stores, and internal data. Use the `getDataInterfaces` method to retrieve information about shared local data stores in the generated code.

For example, create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescriptor = coder.getCodeDescriptor(model)
```

Retrieve properties of the shared local data store in the generated code.

```
dataInterface = codeDescriptor.getDataInterfaces('SharedLocalDataStores')
```

`dataInterface` is an array of `coder.descriptor.DataInterface` objects.

For more information, see Get Code Description of Generated Code.

Deployment

Run on Custom Hardware app for external mode simulations

The Run on Custom Hardware app enables you to run external mode simulations on your development computer or target hardware that is not supported by MathWorks® support packages. To run an external mode simulation, you:

1. Build the target application on your development computer.
2. Deploy the target application to the target hardware.
3. Connect Simulink to the target application that runs on the target hardware.
4. Start execution of generated code on the target hardware.

With the app, you can:

- Perform the steps separately or with one click.
- Register custom launchers that deploy the target application.

The app is in the **Hardware** tab. To open the tab, on the Simulink toolstrip **Apps** tab, click **Run on Custom Hardware**. Or, on the Embedded Coder app **C Code** tab, select **Verify Code** > **Run on Custom Hardware**.

For more information, see:

- Host-Target Communication with External Mode Simulation
- External Mode Simulation by Using XCP Communication
- External Mode Simulation with TCP/IP or Serial Communication

Specify CacheFolder or CodeGenFolder separately

In R2019b, you can use `Simulink.fileGenControl` to specify separately nondefault values for these Simulink preferences:

- `CacheFolder`
- `CodeGenFolder`

This table shows R2019a and R2019b behavior.

Preference Setting	`CodeGenFolder` -- Default	`CodeGenFolder` -- Nondefault
`CacheFolder` -- Default	Supported in R2019a and R2019b	Not supported in R2019a Supported in R2019b
`CacheFolder` -- Nondefault	Not supported in R2019a Supported in R2019b	Supported in R2019a and R2019b

Removal of support for STF_par_cfg_chk

R2019b does not support the *`STF`*`_par_cfg_chk` function for checking the configuration of parallel workers.

packNGo packages buildInfo.mat files for folder hierarchy

In R2019b, for the `packNGo` function, if you specify `'hierarchical'` for the `'packType'` argument, the function packages the `buildInfo.mat` files for the top model, referenced model, and shared utility folders. Previously, the function packaged the `buildInfo.mat` file only for the top model.

Check bug reports for issues and fixes

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

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

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

R2019a

Model Architecture and Design

Functionality being removed or changed

Compatibility Considerations

Previously, if a protected model referenced a model that shared a name with either a different protected model or a different model within the hierarchy of another protected model, you could simulate and generate code for a model that referenced both protected models. In R2019a, attempting to generate code for such a top model, or attempting to simulate such a top model in software-in-the-loop (SIL), processor-in-the-loop (PIL), or rapid accelerator modes, results in an error. For more information, see Protect Models to Conceal Contents.

Data, Function, and File Definition

Code Generation

Register new hardware devices

Extend the range of supported hardware by using the `target.Processor` and `target.LanguageImplementation` classes to register new devices.

For details, see Register New Hardware Devices.

Japanese translation for code generation report

When running Simulink Coder in a Japanese locale, the code generation report is in Japanese.

Simulink Coder contextual tabs on the Simulink Toolstrip Tech Preview

In R2019a, you have the option to turn on the Simulink Toolstrip. For more information, see "Simulink Toolstrip Tech Preview replaces menus and toolbars in the Simulink Desktop".

The Simulink Toolstrip includes contextual tabs, which appear only when you need them. The Simulink Coder contextual tabs include options for completing actions that apply only to Simulink Coder.

- To access the **C Code** tab, open the Simulink Coder app from the App gallery tab on the Simulink Toolstrip. If the **C++ Code** tab opens, select C code from the **Output** section of the gallery.
- To access the **C++ Code** tab, open the Simulink Coder app from the App gallery tab on the Simulink Toolstrip. If the **C Code** tab opens, select C++ code from the **Output** section of the gallery.
- To access the **Hardware** tab, in the **C Code** tab, select **Verify** > **Run on Custom Hardware**.

Documentation does not reflect the addition of the Simulink Coder contextual tabs.

Retrieve information about internal data in the generated code by using Code Descriptor API

You can now retrieve a `coder.descriptor.DataInterface` object for internal data in the generated code, in addition to inports, outports, parameters, and global data stores. Use the `getDataInterfaces` method to retrieve information about these internal data structures in the generated code:

- DWork vectors
- Block I/O
- Zero-crossings

For example, build the `rtwdemo_counter` model.

```
rtwbuild('rtwdemo_counter')
```

Create a `coder.codedescriptor.CodeDescriptor` object for the model.

```
codeDescriptor = coder.getCodeDescriptor('rtwdemo_counter')
```

Retrieve a list of all data interface types in the generated code.

```
dataInterfaceTypes = codeDescriptor.getDataInterfaceTypes()
```

`dataInterfaceTypes` has these values:

```
    {'Inports'     }
    {'Outports'    }
    {'InternalData'}
```

Retrieve properties of the internal data in the generated code.

```
dataInterface = codeDescriptor.getDataInterfaces('InternalData')
```

`dataInterface` is an array of `coder.descriptor.DataInterface` objects. Obtain the details of the first internal data of the model by accessing the first location in the array.

```
dataInterface(1)
```

The first `coder.descriptor.DataInterface` object with properties is returned.

```
              Type: [1×1 coder.descriptor.types.Integer]
               SID: 'rtwdemo_counter:15'
     GraphicalName: 'X_g'
       VariantInfo: [0×0 coder.descriptor.VariantInfo]
    Implementation: [1×1 coder.descriptor.StructExpression]
            Timing: [1×1 coder.descriptor.TimingInterface]
```

Deployment

Separate makefile for shared code and simplified template makefiles

R2019a provides a simplified process for compiling and linking generated code. The build process:

- Creates a separate makefile for shared utility code, for example, `slprj/ert/_sharedutils/rtwshared.mk`.
- No longer uses these template makefile tokens:
  - `|>S_FUNCTIONS <|`
  - `|>S_FUNCTIONS_OBJ <|`
  - `|>SHARED_SRC <|`
  - `|>SHARED_SRC_DIR <|`
  - `|>SHARED_BIN_DIR <|`
  - `|>SHARED_LIB <|`
  - `|>MODELREF_INC_PATH <|`

  If you specify a template makefile that contains any of those tokens, the build process produces a warning. If you want to suppress the warning, in the template makefile, insert the text `# NO_WARN_LEGACY_TOKENS`.
- Performs faster parallel builds for some models, for example, models that use many shared utility files.

For more information, see Customize Template Makefiles.

New configuration property in shared utility checksum hash table

In the shared utility checksum hash table, `toolchainOrTMF` replaces the `tmfName` and `toolchainName` properties.

If your build process uses a toolchain or a template makefile that is associated with a toolchain, the `toolchainOrTMF` value provides the name of the toolchain.

If your build process uses a template makefile, the `toolchainOrTMF` value provides the name of the template makefile.

For more information, see Manage the Shared Utility Code Checksum.

Read and write data over serial port using BeagleBone Blue SCI blocks

The Simulink Coder Support Package for BeagleBone® Blue Hardware supports the SCI Read and SCI Write blocks. You can use these blocks to read and write serial data to the Universal Asynchronous Receiver Transmitter (UART) on the BeagleBone hardware.

Measure data using BeagleBone Blue sensor blocks

The Simulink Coder Support Package for BeagleBone Blue Hardware supports the Barometer and MPU9250 blocks. These blocks add the following functionality:

- Barometer: Measure barometric air pressure around the BMP280 sensor.
- MPU9250: Measure acceleration, angular rate, and magnetic field along the axes of the MPU9250 sensor. You can also calculate fusion values, such as Euler angles and quaternions.

XCP slave memory allocation for external mode signal logging

For a GRT or ERT system target file, if you select **External mode** and set **Transport layer** to `XCP on TCP/IP` or `XCP on Serial`, you enable **Static memory allocation** and **Static memory buffer size**. In the **Static memory buffer size** field, you can specify the size of XCP slave memory allocated for signal logging.

For more information, see:

- External Mode Simulation with XCP Communication
- Static memory allocation

.

Performance

Inplace updates for Assignment and Bus Assignment blocks

In R2019a, the name of the parameter **Perform inplace updates for Bus Assignment blocks** on the **Optimization** tab has been updated to **Perform inplace updates for Assignment and Bus Assignment blocks** to reflect the reuse of input and output variables of Bus Assignment and Assignment blocks if possible. For further information refer to Perform in-place updates for Assignment and Bus Assignment blocks .

New location for optimization configuration parameter

Previously, in the Configuration Parameters dialog box, the **Signal storage reuse** parameter was on the **Simulation Target** pane. In R2019a, the **Signal storage reuse** parameter is on the **Code Generation > Optimization** pane.

Verification

Check bug reports for issues and fixes

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

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

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

R2018b

Model Architecture and Design

Data, Function, and File Definition

Code Generation

Row-Major Array Layout: Simplify integration with external C/C++ code for Lookup Table and other blocks

The code that you generate can store array elements in column-major or row-major array layout. MATLAB uses column-major array layout by default, whereas the C/C++ languages use row-major layout by default.

For example, consider matrix `A` which is a 4x3 matrix:

```
A =
    1    2    3
    4    5    6
    7    8    9
    10   11   12
```

In column-major array layout, the elements of the columns are contiguous in memory. `A` is represented in the generated code as:

```
1    4    7    10    2    5    8    11    3    6    9    12
```

In row-major array layout, the elements of the rows are contiguous. `A` is represented in the generated code as:

```
1    2    3    4    5    6    7    8    9    10    11    12
```

In previous releases, the code generator produced C/C++ code that used column-major array layout. In R2018b, you can choose to generate code that uses column-major or row-major array layout. Row-major layout can improve performance for certain algorithms and ease integration with other code that also uses row-major layout.

For more information, see Code Generation of Matrices and Arrays.

Model Configuration

To support row-major code generation, these are the new model configuration settings in the Configuration Parameters dialog box.

Parameter Name	Command-Line Name	Default	Location
Array layout	`ArrayLayout`	`Column-major`	**Code Generation > Interface**
Use algorithms optimized for row-major array layout (Simulink)	`UseRowMajorAlgorithm`	Off	**Math and Data Types**
External functions compatibility for row-major code generation	`UnsupportedSFcnMsg`	`error`	**Code Generation > Interface**
Default function array layout (Simulink)	`DefaultCustomCodeFunctionArrayLayout`	`Not specified`	**Simulation Target**

To prepare your model for row-major code generation, set the model configuration parameter **Array layout** to `Row-major`. You can also preserve array dimensions for parameters in the generated code. For more details, see "Multi-Dimensional Arrays: Preserve array dimensions for parameters and lookup tables in generated code" (Embedded Coder).

When **Array layout** is set to `Row-major`, the code generator uses algorithms to maintain consistency of numeric results between simulation and the generated code. Sometimes, the generated code for these algorithms can be inefficient. You can enable the **Use algorithms optimized for row-major array layout** configuration parameter to enable efficient algorithms, which might result in numeric differences between simulation and generated code. Use this configuration parameter to enable the lookup table, sum, and product blocks for efficient row-major code generation. For more information about lookup tables, see "Row-Major Array Layout: Simplify integration with external C/C++ code for Lookup Table and other blocks".

When you integrate your external C functions by using the C Caller block, you can specify the **Default function array layout** configuration parameter to `Row-major` to enable row-major code generation. For details about the C Caller block, see "C Caller Block: Call external C functions directly from the model".

If you have Embedded Coder, you can use Code Replacement Tool for creating row-major code replacement table entries. For more information, see "Code Replacement: Optimize generated code with SIMD and row-major order support and code replacement enhancements" (Embedded Coder).

For details about S-functions, see "Row-major code generation support for S-functions".

Model Advisor Checks

In Embedded Coder, these Model Advisor checks have been added to support row-major code generation:

- **Identify blocks generating inefficient algorithms** (Embedded Coder) – A precompile check enabled by default. This check flags the blocks that generate inefficient algorithms. To avoid inefficient algorithms, enable the **Use algorithms optimized for row-major array layout** configuration parameter.
- **Check for blocks not supported for row-major code generation** (Embedded Coder) – Flags all blocks that are not supported for row-major code generation.
- **Identify TLC S-Functions with unset array layout** (Embedded Coder) – Flags all S-function blocks that have `SSArrayLayout` set to `SS_UNSET`.

Miscellaneous Support

New TLC function `LibIsRowMajor` returns true when the current model uses the row-major array layout.

New method `getArrayLayout` in Code Descriptor API returns the array layout of the generated code.

Verification

You can verify the row-major generated code by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

Limitations

In R2018b, row-major code generation is not supported for any of the toolboxes. For complete list of blocks unsupported for row-major code generation, see Unsupported Blocks for Row-Major Code Generation.

Hardware Implementation Parameters: ProdHWDeviceType and TargetHWDeviceType are case-insensitive

In R2018b, the values for the `ProdHWDeviceType` and `TargetHWDeviceType` command-line parameters are case-insensitive. For example, these commands specify the same value for `ProdHWDeviceType`:

- `set_param(`*`modelOrConfigurationSet`*`, 'ProdHWDeviceType', 'atmel->avr')`
- `set_param(`*`modelOrConfigurationSet`*`, 'ProdHWDeviceType', 'Atmel->AVR')`

Deployment

XCP External Mode Simulation: Animate Stateflow charts and run pure integer code

XCP external mode simulations support uploading DWork data during model execution, which enables you to:

- Animate Stateflow charts.
- View state activity through the Simulation Data Inspector.

For more information, see:

- Animate Stateflow Charts (Stateflow)
- View State Activity by Using the Simulation Data Inspector (Stateflow)

You can run XCP external mode simulations with code that is generated for target hardware that does not support floating-point arithmetic, that is, code generated with `PurelyIntegerCode` set to `'on'`.

If `PurelyIntegerCode` is set to `'on'`, you must also specify `FixedStep` with a resolution that is greater or equal to 1 microsecond. For example, you can specify 1.000001, but not 1.0000001.

For more information, see:

- Support: floating-point numbers
- XCP External Mode Limitations

ST Nucleo Tuning and Monitoring: Perform external mode simulation on ST Nucleo for parameter tuning and signal monitoring by using XCP over TCP/IP or UART (Serial)

The Simulink Coder Support Package for STMicroelectronics® Nucleo Boards supports external mode simulation for parameter tuning and signal monitoring using Universal Measurement and Calibration Protocol (XCP) over TCP/IP or UART as the transport layer. XCP-based external mode enables signal monitoring using Simulation Data Inspector and Dashboard blocks.

STMicroelectronics Nucleo Boards: Support for TCP/IP and UDP Blocks

The Simulink Coder Support Package for STMicroelectronics Nucleo Boards supports TCP/IP and UDP blocks. For more information, see TCP Receive, TCP Send, UDP Receive, and UDP Send.

Build Process: Library and header files for model reference hierarchy are not copied

Previously, the build process copied:

- Model reference library files to the build folder for the parent model
- Model reference header files to the `referenced_model_includes` subfolder of the build folder for the parent model.

In R2018b, the build process does not copy model reference library or header files. The build process creates a response file for the header file paths.

If you want the build process to copy model reference header files to the …`/parentModel/referenced_model_includes` subfolder, set these new custom toolchain attributes to `true`:

- `NoCompilerCommandFile`
- `CopyReferencedModelHeaders`

For more information, see `addAttribute`.

The build argument `MODELREF_LINK_LIBS` is not supported. For example, the `getBuildArgs` function does not extract the `MODELREF_LINK_LIBS` argument identifier and value from a build information object.

The `MODELREF_LINK_LIBS` template makefile (TMF) token is still supported.

If you run a MATLAB script that uses the `getBuildArgs` function to extract an argument identifier and value for `MODELREF_LINK_LIBS`, the script might fail.

Build Process: MATLAB_INCLUDES is not required in custom template makefiles

The `MATLAB_INCLUDES` macro is not required in custom template makefiles. In R2018b, the build process extracts the required include paths from a build information object. You do not have to remove the macro from existing template makefiles.

Simulink Coder Support Package for VEX EDR V5 Robot Brain: Generate, build, and deploy Simulink models on VEX EDR V5 Robot Brain

The Simulink Coder Support Package for VEX® EDR V5 Robot Brain is available from release R2018b onwards. You can use the support package to generate, build, and deploy Simulink models on the VEX EDR V5 Robot Brain.

The support package includes a library of Simulink blocks for programming the VEX EDR V5 Robot Brain to work with sensors (analog and digital), actuators (V5 Smart Motor, DC motor, and Servo motor), and gamepad inputs (from the VEX V5 Controller).

Support for BeagleBone Blue hardware available on Mac OS

You can use the Simulink Coder Support Package for BeagleBone Blue Hardware on Mac OS.

Extended list of blocks in BeagleBone Blue support package

The Simulink Coder Support Package for BeagleBone Blue Hardware is enhanced to support these additional blocks in the Actuators, Basic, Communication, and Video libraries of the support package.

Library	Block	Usage
Actuators	Servo Motor	Sets the shaft position of a standard servo motor that is connected to the hardware. The position can vary from `0` to `180` degrees.
Basic	Analog Input	Measures the voltage of the analog input pin relative to the analog reference voltage (1.8 volts) on the hardware. If the measured voltage equals the ground voltage, the block outputs `0`. If the measured voltage equals the analog reference voltage, the block outputs `1023`.
	Digital Read	Reads the logical value of a digital input pin on the hardware. If the logical value of the digital pin is LOW (0 volts), the block outputs `0`. If the logical value of the digital pin is HIGH (3.3 volts), the block outputs `1`.
	Digital Write	Sets the logical value of a digital output pin on the hardware. An input of `1` sets the logical value of the digital pin HIGH to 3.3 volts. An input of `0` sets the logical value of the digital pin LOW to 0 volts.
	PWM	Generates square waves on the specified analog output pin. The input value controls the duty cycle of the waveform. An input of `0` produce waves with 0% duty cycle. An input of `255` produce waves with 100% duty cycle.
Communication	I2C Master Read	Reads data from an I2C slave device or an I2C slave device register connected to the hardware.
	I2C Master Write	Writes data to an I2C slave device or an I2C slave device register connected to the hardware.

Library	Block	Usage
	SPI Master Transfer	Writes data to and reads data from a slave device connected to the hardware over the serial peripheral interface (SPI).
	SPI Register Read	Reads data from registers of a slave device over the SPI.
	SPI Register Write	Writes data to registers of a slave device over the SPI.
Video	SDL Video Display	Displays video data using the Simple DirectMedia Layer (SDL) multimedia library. The block accepts video data in YCbCr 4:2:2 and RGB formats.
	V4L2 Video Capture	Captures video from USB cameras that are supported by the Video for Linux Two (V4L2) API driver framework. The camera must be Universal Video Class (UVC) compatible.

Performance

Faster generated code for matrix operations in the MATLAB Function block

To improve the simulation speed of MATLAB Function block algorithms that perform certain low-level vector and matrix computations (such as matrix multiplication), the simulation software can call BLAS functions. In R2018b, if you use Simulink Coder to generate C/C++ code for these algorithms, you can specify that the code generator produce BLAS function calls. If you specify that you want to generate BLAS function calls and the input arrays for the matrix operations meet certain criteria, the code generator produces calls to relevant BLAS functions. The code generator uses the CBLAS C interface.

BLAS is a software library for numeric computation of basic vector and matrix operations that has several highly optimized machine-specific implementations. MATLAB uses this library for basic matrix computations. Simulink uses the BLAS library that is included with MATLAB. Simulink Coder uses the BLAS library that you specify. If you do not specify a BLAS library, the code generator produces code for the matrix operation instead of generating a BLAS call.

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

Verification

Check bug reports for issues and fixes

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

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

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

R2018a

Model Architecture and Design

Protected Models: Use concurrent tasking

In R2018a, you can protect a model when you select the parameter **Allow tasks to execute concurrently on target** (`ConcurrentTasks`). Concurrent tasking support includes simulation and code generation for the protected model.

Model Advisor Check: Enhance check for blocks in your model that are not supported by code generation

Model Advisor check Check for blocks not supported by code generation now analyzes content of library linked blocks and content in all masked subsystems.

Configuration Reference in Data Dictionary: Quickly select code generation target for model reference hierarchy

In R2018a, to select a code generation target for a model reference hierarchy without modifying the individual models, use a configuration reference in the data dictionary.

1. In the data dictionary, create a configuration set for each of your code generation targets.
2. In the data dictionary, create a configuration reference.
3. For each model in the hierarchy, point the configuration reference to the configuration reference in the data dictionary.

Once configured, switch the code generation target for your model reference hierarchy by selecting the appropriate configuration set in the configuration reference that you created in the data dictionary.

For more information, see Use Configuration Reference to Select Code Generation Target.

Data, Function, and File Definition

SimulinkGlobal Name Change: Storage class renamed to Model default

In R2018a, the storage class `SimulinkGlobal` has a new name, `Model default`.

If you have Embedded Coder, you can control the meaning of `Model default` by configuring settings in **Code Mappings** > **Data Defaults**. For more information, see Default Code Configurations for Data and Functions: Apply default code generation configurations for categories of model data and functions across a model (Embedded Coder).

Compatibility Considerations

Existing scripts that assign `SimulinkGlobal` to data items continue to work. Consider replacing `SimulinkGlobal` with `Model default`.

Scripts that query the storage classes of data items and expect `SimulinkGlobal` now encounter `Model default` instead. You must modify these scripts so they expect `Model default`.

Convert to Parameter Object: Apply storage classes to numeric variables

Previously, when you inspected a numeric workspace variable in the Model Explorer or in the Model Data Editor **Parameters** tab, you could not use the **Storage Class** column to apply a storage class. In R2018a, the column offers a single option, `Convert to parameter object`, which converts the variable to a parameter object such as `Simulink.Parameter`. After the conversion, you can use the column again to apply a storage class. To convert many variables, you can take advantage of the existing batch-editing abilities of the Model Explorer and the Model Data Editor.

For more information, see Apply Storage Class When Block Parameter Refers to Numeric MATLAB Variable.

Code Generation

Code Descriptor: Retrieve meta information about generated code by using MATLAB API

You can use the code descriptor API once the code is generated to obtain meta-information about the generated code. Use the code descriptor API to describe these items in the generated code:

- Data Interfaces: inports, outports, parameters, and global data stores.
- Function Interfaces: initialize, output, update, and terminate.
- Run-time information of the data and function interfaces, such as timing requirements of each interface entity.
- Model hierarchy information and code description of referenced models.

The function `getCodeDescriptor` takes a single argument `modelName` or `buildDirectory` and returns the `coder.codedescriptor.CodeDescriptor` object.

The code descriptor API has these methods:

Name	Description
`getAllDataInterfaceTypes`	Returns all the data interface types.
`getDataInterfaceTypes`	Returns all the data interface types in the generated code.
`getDataInterfaces(dataInterface)`	Returns the properties of the specified data interface. `dataInterface` can take any of the values returned by `getAllDataInterfaceTypes`.
`getAllFunctionInterfaceTypes`	Returns all the function interface types.
`getFunctionInterfaceTypes`	Returns all the function interface types in the generated code.
`getFunctionInterfaces(functionInterface)`	Returns the properties of the specified function interface. `functionInterface` can take any of the values returned by `getAllFunctionInterfaceTypes`.
`getReferencedModelNames`	Returns the names of the referenced models.
`getReferencedModelCodeDescriptor(refModelName)`	Returns the code descriptor object for the referenced model specified in `refModelName`.

For more information, see Get Code Description of Generated Code.

Code Generation Advisor: Updates to parameter recommendations for objectives

In R2018a, when the Code Generation Advisor checks your model configuration settings against execution efficiency objectives, it does not consider the parameter **Reuse buffers of different sizes and dimensions** (`DifferentSizesBufferReuse`).

Code Obfuscation: Protect intellectual property by using rtwbuild option

When you use the `rtwbuild` function, you can now specify whether to generate obfuscated C code. Use the `ObfuscateCode` option. If `true`, the code generator produces obfuscated code that you can share with third parties with reduced likelihood of compromising intellectual property.

Hardware Implementation Settings: Inaccurate values corrected

R2018a provides the correct values for these **Hardware Implementation** pane settings.

Device vendor	Device type	Device detail	R2018a value	Previous value
`Texas Instruments`	`C5000`	**Number of bits** per **pointer**	`16`	`32`
`Texas Instruments`	`C5000`	**Number of bits** per **ptrdiff_t**	`16`	`32`
`Texas Instruments`	`TMS570 Cortex-R4`	**Byte ordering**	`Big Endian`	`Little Endian`

When you open a saved model from a previous release, R2018a updates the incorrect values.

For more information, see Hardware Implementation Pane.

Deployment

Build Process: Specify toolchain for template makefile

To build code generated from Simulink models, you can specify a process that uses a template makefile that is associated with a toolchain.

You can still use the template makefile approach that you used with previous releases, that is, you can use a template makefile build process that is not associated with a toolchain.

For more information, see Choose Build Approach and Configure Build Process.

External Mode Simulation: Use XCP communication protocol

Run external mode simulations where the communication between your development computer and the target processor is based on XCP, the Universal Measurement and Calibration Protocol. R2018a provides:

- An XCP slave communication stack for a range of target processors and transport layers, for example, TCP/IP and Sxl.
- An external mode XCP transport layer for your development computer, which supports:
  - Parameter tuning.
  - Signal monitoring through the Simulation Data Inspector, Dashboard blocks, and Scope blocks.

For more information, see:

- Host-Target Communication with External Mode Simulation
- External Mode Simulation with XCP Communication
- Customize XCP Slave Software

External Mode Simulation: EXT_MODE is not required in template makefile

For external mode simulations that use the template makefile (TMF) approach to build model code, the code generation option, `EXT_MODE`, is not required in the template makefile.

If you currently use a custom template makefile, you can update the template makefile:

1. Specify a toolchain for the build process by adding the `TOOLCHAIN_NAME` macro to the template makefile.
2. Remove `EXT_MODE` statements from the template makefile.

When you run an external mode simulation, the build process uses the `RTW.BuildInfo` object to specify the required transport layer source files in the generated makefile.

To specify target-specific source files for the transport layer, you must use `addSourceFiles` to add the required files to the `RTW.BuildInfo` object.

You do not have to update your custom template makefile. You can continue to use your current template makefile without the `TOOLCHAIN_NAME` and `EXT_MODE` changes.

For more information, see:

- Create a Transport Layer for TCP/IP or Serial External Mode Communication
- Associate the Template Makefile with a Toolchain
- Customize Post-Code-Generation Build Processing
- Customize Build Process with STF_make_rtw_hook File
- Customize Build Process with sl_customization.m

Build Process Status for Parallel Builds: View and interact with build process status for parallel builds of referenced model hierarchies

You can now view and interact with build process status for parallel builds through the **Build Process Status** window. In the window, you see the status of referenced model builds, the elapsed time for builds, and a **Cancel** button that you can use to end the build process without creating incomplete build artifacts. For more information, see View Build Process Status.

BeagleBone Blue Support Package: Generate, build, and deploy Simulink models on BeagleBone Blue hardware

The Simulink Coder Support Package for BeagleBone Blue Hardware is available from release R2017b onwards. You can use the support package to generate, build, and deploy Simulink models on the BeagleBone Blue hardware. You can also tune parameter values in the model and receive data from the model when it is running on the hardware.

The block library of the BeagleBone Blue support package includes these blocks.

Block	Usage
Button	Read logical state of button
DC Motor	Set power, direction, and stopping action of a DC motor
Encoder	Measure incremental position and direction of a rotating motor
LED	Illuminate built-in LED
TCP/IP Receive	Receive data over TCP/IP network from a remote host
TCP/IP Send	Send data over TCP/IP network to another remote host
UDP Receive	Receive UDP packets from another UDP host on an Internet network
UDP Send	Send UDP packets to another UDP host on Internet network

Performance

Configuration Set: New location for optimization model configuration parameters

Previously, in the Configuration Parameters dialog box, there were three optimization panes. In R2018a, there is a single **Optimization** pane, and this pane is under **Code Generation**. This pane now contains only parameters that apply to code generation. Parameters that apply only to simulation have moved to either the new **Math and Data Types** pane or the **Simulation Target** pane. This table shows the moved parameters and their locations in R2018a.

Parameter	Pane
Default for underspecified data type	Math and Data Types
Use division for fixed-point net slope computation	Math and Data Types
Use floating-point multiplication to handle net slope corrections	Math and Data Types
Application lifespan	Math and Data Types
Implement logic signals as Boolean data (vs. double)	Math and Data Types
Evaluated application lifespan	Math and Data Types
Block reduction	Simulation Target
Compiler optimization level	Simulation Target
Conditional input branch execution	Simulation Target
Signal storage reuse	Simulation Target
Verbose accelerator builds	Simulation Target

For more information, see Performance and Optimization Pane.

Verification

Check bug reports for issues and fixes

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

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

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

R2017b

Model Architecture and Design

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

Generate C++ Code for Simulink functions

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

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

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

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

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

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

- Starts and stops an application component
- Calculates initial conditions

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

Data, Function, and File Definition

Tunable Parameters: Tune parameters in model workspace

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

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

Compatibility Considerations

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

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

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

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

Code Generation

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

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

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

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

Compatibility Considerations

- In R2017b, advanced parameters that were previously available only on the **All Parameters** tab can be found under the **Advanced Parameters** toggle of the relevant category pane. To access this toggle, hover over the ellipsis at the bottom of the pane. Alternatively, to find an advanced parameter, use the search tool at the top of the dialog box.
- If you use an `sl_customization.m` script to hide or disable parameters in the Configuration Parameters dialog box, the script requires updates to widget ID's and callback registrations. For example:
  - In R2017a:

    ```
    function sl_customization(cm)

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

    end

    function disableRTWBrowseButton(dialogH)

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

    end
    ```
  - In R2017b:

    ```
    function sl_customization(cm)

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

    end

    function disableRTWBrowseButton(dialogH)
    ```

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

    end
```

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

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

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

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

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

For more information, see:

- Manage Build Process Folders
- `Simulink.fileGenControl`

Warning Messages: Build process diagnostic warnings in Diagnostic Viewer

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

Code Generation Advisor: Updates to parameter recommendations for objectives

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

- For checks against safety precaution objectives, the Code Generation Advisor does not consider the parameter **Conditional input branch execution** (`ConditionallyExecuteInputs`).
- For checks against ROM efficiency objectives, the Code Generation Advisor considers the parameter **Remove code that protects against division arithmetic exceptions** (`NoFixptDivByZeroProtection`).
- For checks against ROM and execution efficiency objectives, the Code Generation Advisor considers the parameter **Support long long** (`ProdLongLongMode`).

Performance

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

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

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

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

Check bug reports for issues and fixes

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

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

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

R2017a

Model Architecture and Design

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

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

- Subsystems across model reference boundaries that contain Data Store blocks that read from or write to a global data store. In the Data Store Block Parameters dialog box, you specify a global data store for the **Data store name** parameter. You define the global data store in the base workspace using a signal object.
- Reusable functions that have user-specified names. In the Subsystem Block Parameters dialog box, on the **Code Generation** tab, you specify a name by selecting `User specified` for the **Function name options** parameter and providing a name for the **Function name** parameter. The code generator no longer appends the user-specified name with a checksum.

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

TMF and EXTMODE fields optional in TLC file

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

For more information, see Customize System Target Files.

Data, Function, and File Definition

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

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

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

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

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

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

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

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

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

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

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

Code Generation

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

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

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

For more information, see:

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

Source file includes shared utility header file

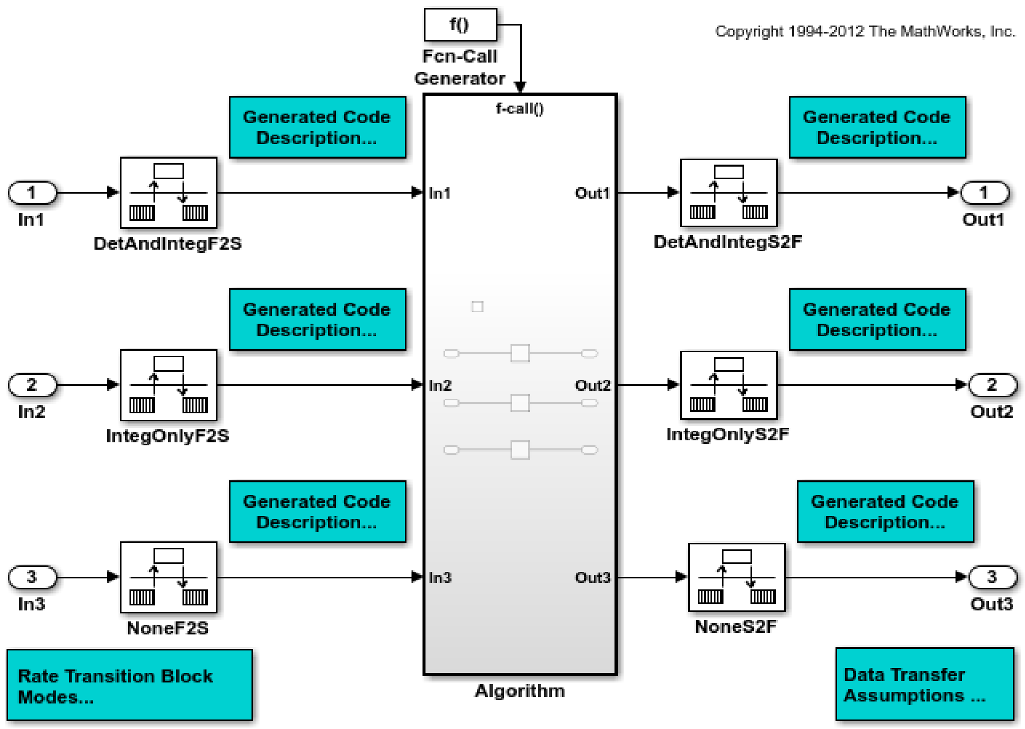
In R2016b, the *`model`*`.h` file or the *`subsystem`*`.h` file contained the `#include` command for the header file that contained declarations for shared utility functions. In R2017a, the source file contains the `#include` command for this header file. Including the header file in the source file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

Generated code for Rate Transition block variables with volatile qualifier

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

In R2017a, the generated code contains the `volatile` qualifier in variables in the `D_Work` structure. This qualifier indicates to most compilers not to perform optimizations that remove or reorder protection logic because the value of these variables can possibly change outside control or detection of the program. The presence of the `volatile` qualifier means that users no longer have to turn off compiler optimizations and sacrifice performance for safety.

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


f()
Fcn-Call Generator
Copyright 1994-2012 The MathWorks, Inc.
f-call()
Generated Code Description...
Generated Code Description...
1 In1
DetAndIntegF2S
In1
Out1
DetAndIntegS2F
1 Out1
Generated Code Description...
Generated Code Description...
2 In2
IntegOnlyF2S
In2
Out2
IntegOnlyS2F
2 Out2
Generated Code Description...
Generated Code Description...
3 In3
NoneF2S
In3
Out3
NoneS2F
3 Out3
Rate Transition Block Modes...
Algorithm
Data Transfer Assumptions ...

In R2016b, the `rtwdemo_ratetrans.h` file contained this code:

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

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

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

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

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

IncludeMdlTerminateFcn not checked against efficiency objectives

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

More information in code generation report summary

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

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

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

Deployment

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

This table lists the support for these new blocks.

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

Support for new board STMicroelectronics Nucleo F746ZG

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

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

Support for new board STMicroelectronics Nucleo F411RE

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

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

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

This table lists the support for these new blocks.

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

Performance

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

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

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

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

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

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

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

Dynamic memory allocation does not apply to:

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

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

Check bug reports for issues and fixes

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

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

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

R2016b

Model Architecture and Design

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

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

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

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

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

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

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

Supported state owner blocks include:

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

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

Updates to protected model message identifiers

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

Compatibility Considerations

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

Data, Function, and File Definition

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

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

In R2016b, use the Model Data Editor (see “Model Data Editor for applying storage classes to Inport blocks, Outport blocks, signals, and Data Store Memory blocks” on page 14-6) to apply a name and storage class directly to an Outport block. You can now:

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

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

ASAP2 file generation for bus signals and parameters

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

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

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

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

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

Compatibility Considerations

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

Perform additional steps after code generation for nested structures

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

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

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

   ```
   >> !readelf -wi model.elf > model.dwarf;
   ```
2. If you generated code for referenced models, each model reference build generates an `a2l` file. Merge the files using the `rtw.asap2MergeMdlRefs` function.

   ```
   >> rtw.asap2MergeMdlRefs('topmodel','merged.a2l');
   ```
3. To add addresses to the `a2l` file, use the `rtw.asap2SetAddress` function.

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

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

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

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

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

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

  end
end


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

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

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

```
end

end
```

Revise ASAP2 user template for nested structures

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

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

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

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

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

  The third parameter, `dataIdx`, is omitted, because ASAP2 file generation does not support arrays of busses in CHARACTERISTICs.
- Algorithm revision — In templates from previous releases, you could not recurse inside a structure to add CHARACTERISTICs for nested structures. The following line in the template guarded against recursion:

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

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

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

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

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

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

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

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

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

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

  The data appear in the generated code as fields of a single structure. To control the characteristics of the structure type, such as its name, use the properties of the object.
- For COM_AXIS, store each unique set of table data in a `Simulink.LookupTable` object and each unique breakpoint vector in a `Simulink.Breakpoint` object. Use each `Simulink.LookupTable` object in an Interpolation Using Prelookup block and each `Simulink.Breakpoint` object in a Prelookup block. You can reduce memory consumption by sharing breakpoint data between lookup tables.

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

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

Tunable Table Size

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

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

Calibration

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

More explicit purpose for SimulinkGlobal storage class

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

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

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

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

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

Compatibility Considerations

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

- Use the Signal Properties dialog box or block dialog boxes to apply the same name to multiple signals or states that use `SimulinkGlobal`.
- Resolve multiple signal lines or block states to a single `Simulink.Signal` object that uses the storage class `SimulinkGlobal`.

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

Additional tunability support for expressions

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

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

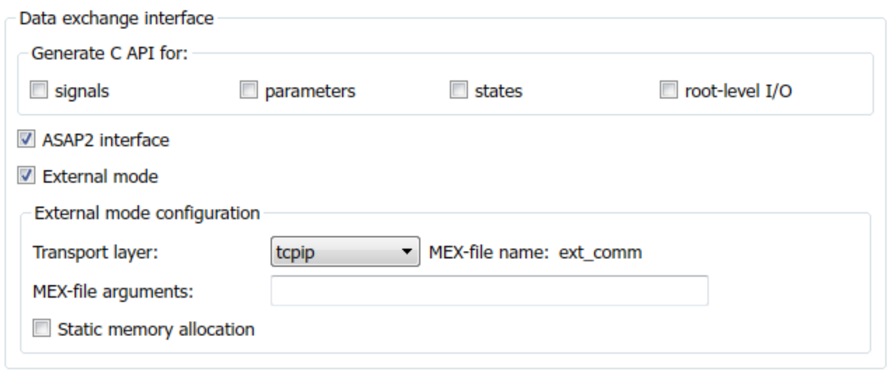
For more information, see Block Parameter Representation in the Generated Code, Parameter Data Types in the Generated Code, and Optimize Generated Code for Lookup Table Blocks.

Code Generation

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

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

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



Data exchange interface

Generate C API for:

☐ signals ☐ parameters ☐ states ☐ root-level I/O

☑ ASAP2 interface

☑ External mode

External mode configuration

Transport layer: tcpip MEX-file name: ext_comm

MEX-file arguments:

☐ Static memory allocation

Standard math library changes

These changes apply to standard math library configurations:

- When you create a model or configuration set, the default standard math library setting is ISO/IEC 9899:1999 C (`C99 (ISO)`). Previously, the default standard math library was ISO/IEC 9899:1990 C (`C89/C90 (ANSI)`). If you are using a compiler that does not support ISO/IEC 9899:1999 C (`C99 (ISO)`), set the **Standard math library** (`TargetLangStandard`) parameter to `C89/C90 (ANSI)`.
- The build process checks whether the specified standard math library and toolchain are compatible. If they are not compatible, a warning occurs during code generation and the build process continues.
- When you change the value of the **Language** parameter, the standard math library updates to ISO/IEC 9899:1999 C (`C99 (ISO)`) for C and ISO/IEC 14882:2003 C++ (`C++03 (ISO)`) for C++. Previously, you adjusted the standard math library to match your programming language selection.

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

Compatibility Considerations

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

For more information, see Standard math library and Toolchain.

SupportVariableSizeSignals not checked against efficiency objectives

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

Use default installation folder on Windows system with ReFS file system

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

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

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

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

Deployment

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

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

- Simulink Coder Support Package for NXP FRDM-KL25Z Board
- Simulink Coder Support Package for NXP FRDM-K64F Board.
- Simulink Coder Support Package for STMicroelectronics Nucleo Boards User Guide

Generate code for STMicroelectronics Nucleo boards

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

- STM32 Nucleo F031K6
- STM32 Nucleo F103RB
- STM32 Nucleo F302R8
- STM32 Nucleo F401RE
- STM32 Nucleo L053R8
- STM32 Nucleo L476RG

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

Support for I2C and PWM blocks for FRDM-KL25Z board

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

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

Support for new blocks for FRDM-K64F board

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

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

Check bug reports for issues and fixes

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

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

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

R2016a

Model Architecture and Design

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

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

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

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

Protected Model Callbacks: Define callbacks for customized protected models

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

Callback objects specify:

- The code to execute for the callback. The code can be a string of MATLAB commands or a script on the MATLAB path. The code can include protected model functions or any MATLAB command that does not require loading the model. You can use the `Simulink.ProtectedModel.getCallbackInfo` function in callback code to get information on the protected model. The function provides the protected model name and the names of submodels. If the callback is specified for `'CODEGEN'` functionality and a `'Build'` event, the function provides the target identifier and model code interface type (`'Top model'` or `'Model reference'`).
- The event that triggers the callback. The event can be `'PreAccess'` or `'Build'`.
- The protected model functionality that the event applies to. The functionality can be `'CODEGEN'`, `'SIM'`, `'VIEW'`, or `'AUTO'`. If you select `'AUTO'`, and the event is `'PreAccess'`, the callback is applied to each functionality. If you select `'AUTO'`, and the event is `'Build'`, the callback is applied only to `'CODEGEN'` functionality. If no functionality is selected, the default behavior is `'AUTO'`.
- The option to override the protected model build. This option applies only to `'CODEGEN'` functionality.

To create a protected model with callbacks:

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

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

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

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

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

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

```
myTestSim
```

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

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

Configure the callback to specify the override option:

```
callbackForCodeGen.OverrideBuild = true
```

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

```
rtwbuild('myModel')
```

```
myTestCG
```

For more information, see Define Callbacks for Protected Model.

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

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

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

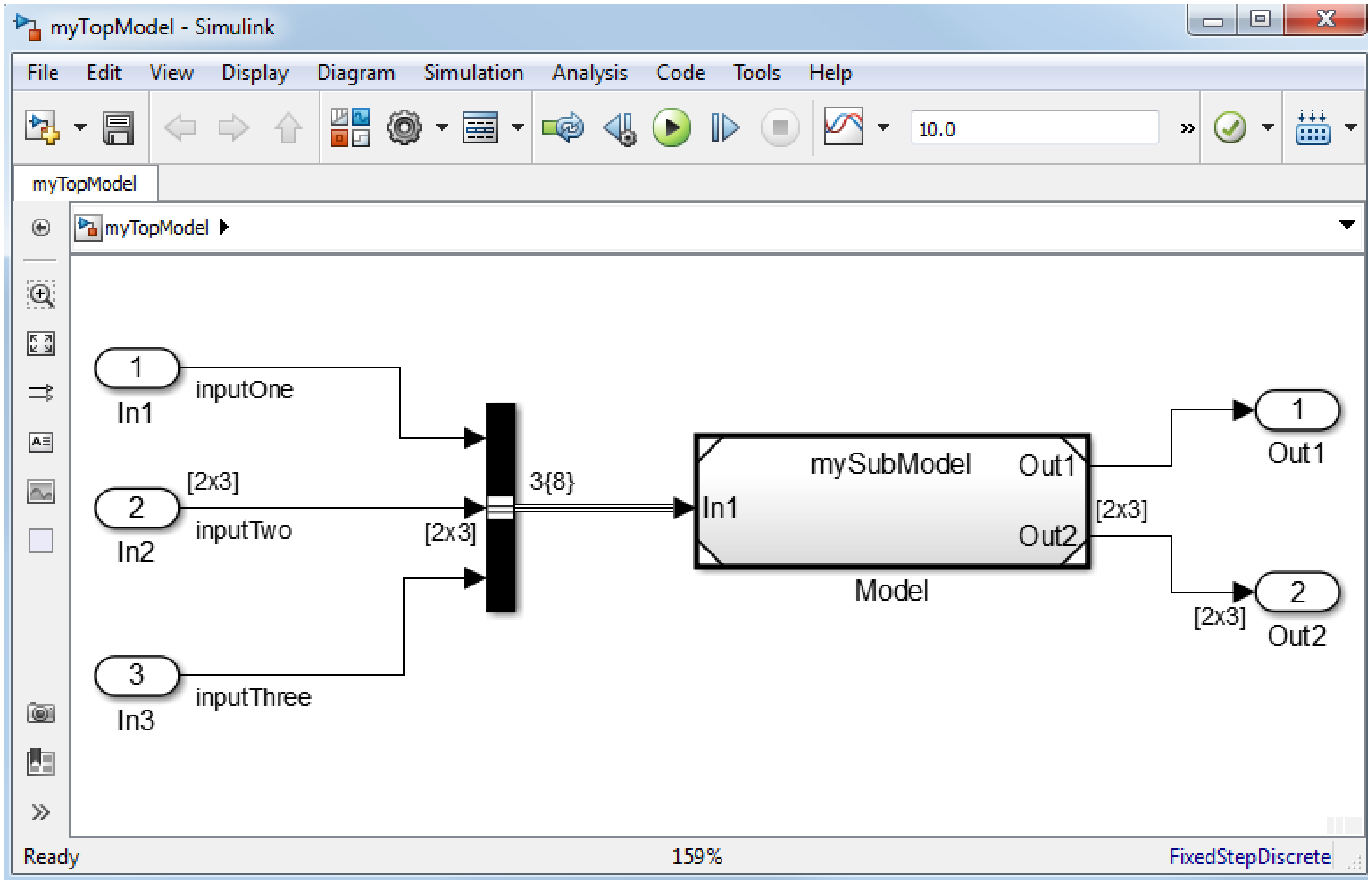
Model Block Virtual Buses: Interface to Model blocks by using virtual buses, reducing data copies in the generated code

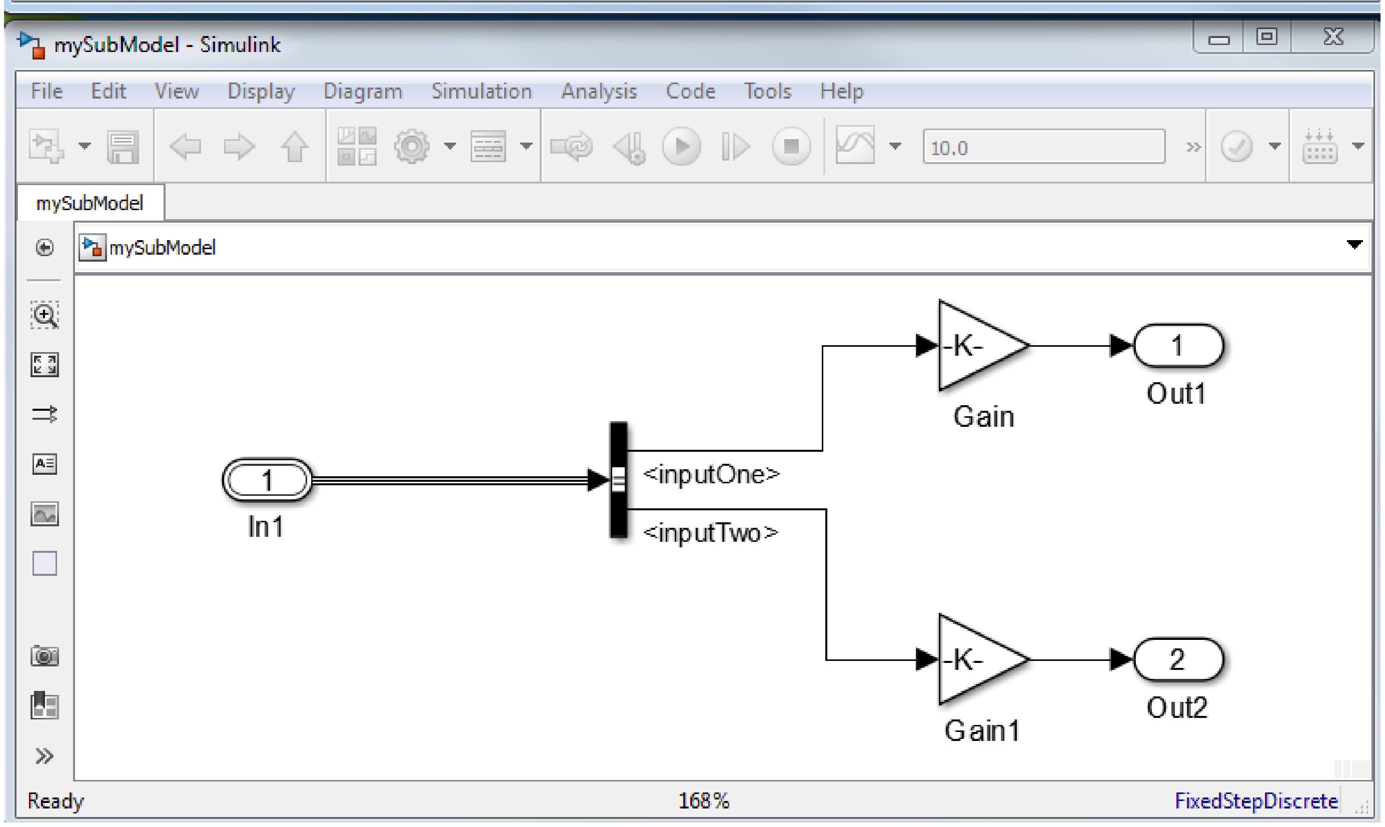
In Simulink, you can create virtual bus signals to exchange signal data between a referenced model and the parent model. You can use a virtual bus as an input to the Model block or as a root-level output of the referenced model.

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

In R2016a, the models exchange the signal data through multiple variables or pointers, each corresponding to a signal element of the bus, instead of a structure. This interface improves the efficiency of the generated code by eliminating the memory consumption of the structure. The code appears and functions as it would if you used multiple signal lines instead of a virtual bus.

For example, suppose that you created a parent model `myTopModel` and a referenced model `mySubModel` in R2015b.


myTopModel - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
myTopModel
1
In1
inputOne
[2x3]
2
In2
inputTwo
[2x3]
3
In3
inputThree
3{8}
mySubModel
In1
Out1
Out2
Model
[2x3]
1
Out1
[2x3]
2
Out2
Ready
159%
FixedStepDiscrete


mySubModel - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
mySubModel
1
In1
<inputOne>
<inputTwo>
-K-
Gain
-K-
Gain1
1
Out1
2
Out2
Ready
168%
FixedStepDiscrete

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

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

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

  rtb_BusConversion_InsertedFor_M.inputOne = myTopModel_U.inputOne;

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

  rtb_BusConversion_InsertedFor_M.inputThree = myTopModel_U.inputThree;

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

The code is inefficient because:

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

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

```
void myTopModel_step(void)
{

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

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

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

Compatibility Considerations

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

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

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

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

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

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

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

Data, Function, and File Definition

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

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

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

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

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

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

In R2016a, you can generate code if the numeric data types of bus signal elements do not match those of the corresponding fields of an initial condition structure. Previously, the code generator displayed an error if the initial condition appeared in the code as a tunable global structure. For more information, see “Tolerance of data type mismatch between bus elements and tunable structure fields” on page 15-8.

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

Simplified method to apply storage classes to signals and states

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

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

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

Storage Classes

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

Storage Type Qualifiers

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

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

Embedded Signal Objects

When you upgrade a model from a previous release to R2016a, signals and states for which you previously set **Package** to `None` and **Storage class** to a storage class other than `Auto` acquire an embedded `Simulink.Signal` object. Use the functions `get_param` and `set_param` to interact with the embedded signal object through the programmatic parameters `SignalObject` (for block output ports) and `StateSignalObject` (for block states).

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

Compatibility Considerations

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

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

  ```
  set_param(0,'DefaultDataPackage','mpt')
  ```
- Unless you already set the option value in a previous release, the option **Storage type qualifier** is hidden in the Signal Properties dialog box and on the **State Attributes** tab in block dialog boxes.

Conflict between different storage classes applied to same signal

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

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

In R2016a, the model generates an error.

Compatibility Considerations

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

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

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

Visibility and functionality changes for programmatic properties of data objects

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

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

Compatibility Considerations

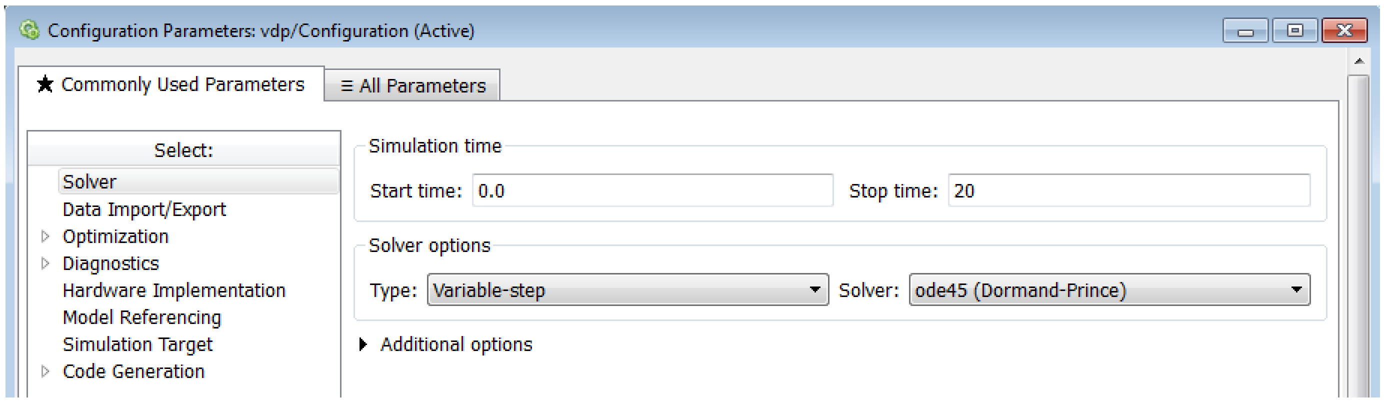
If you have scripts that set the properties of `Simulink.CoderInfo` objects, make sure that the scripts do not generate unnecessary warnings or errors in R2016a. For example, before you set the value of the property `CustomStorageClass`, set the value of the property `StorageClass` to `'Custom'`.

Code Generation

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

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

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



Start time: 0.0
Stop time: 20
Type: Variable-step
Solver: ode45 (Dormand-Prince)

Compatibility Considerations

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

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

Code Generation Pane

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

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

Code Generation > Interface Pane

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

- **Standard math library**
- **Support: non-inlined S-functions**
- **Multiword type definitions**

- **Maximum word length**
- **Use dynamic memory allocation for model initialization**
- **Classic call interface**
- **Single output/update function**
- **Terminate function required**
- **Combine signal/state structures**
- **Internal data visibility**
- **Internal data access**
- **Generate destructor**
- **Use dynamic memory allocation for model block instantiation**
- **MAT-file logging**
- **MAT-file variable name modifier**

Code Generation > Debug Pane

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

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

Data Import/Export Pane

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

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

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

Diagnostics Pane

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

- **Solver data inconsistency**

Diagnostics > Data Validity Pane

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

- **Array bounds exceeded**

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

Diagnostics > Saving Pane

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

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

Diagnostics > Solver Pane

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

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

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

- **SimState object from earlier release**

Optimization Pane

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

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

Optimization > Signals and Parameters Pane

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

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

- **Simplify array indexing**

Simulation Target Pane

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

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

Simulation Target > Custom Code Pane

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

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

Simulation Target > Symbols Pane

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

- **Reserved names**

Add macro definitions to custom code

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

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

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

Faster generated code for linear algebra in the MATLAB Function block

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

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

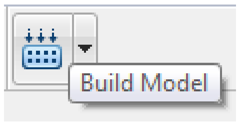
To specify that you want to generate LAPACK function calls and link to a specific LAPACK library, see Speed Up Linear Algebra in Code Generated from a MATLAB Function Block.

Build button removed from Configuration Parameters dialog box

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

Compatibility Considerations

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



Deployment

Hardware implementation parameters enabled by default

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

Simulink Coder Support Package for ARM Cortex-Based VEX Microcontroller

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

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

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

Performance

Removal of Minimize data copies between local and global variables parameter

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

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

Check bug reports for issues and fixes

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

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

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

R2015aSP1

Version: 8.8.1

Bug Fixes

Check bug reports for issues and fixes

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

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

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

R2015b

Model Architecture and Design

Support for C++ code generation in protected models

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

Reusable code for subsystems containing Stateflow charts

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

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

Header file change for model containing messages in Stateflow charts

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

Type definitions in rapid accelerator mode

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

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

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

Data, Function, and File Definition

Configuration parameter Inline parameters name and functionality change

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

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

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

In the Configuration Parameters Dialog Box

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

At the Command Prompt

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

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

In R2015b, constant sample time does not directly indicate block code placement. Block parameters are tunable during simulation regardless of the setting of **Default parameter behavior**. You can still control block parameter tunability in the generated code by adjusting the setting for **Default parameter behavior** and by applying storage classes to parameter data objects. The placement of code for blocks that have constant output values still depends on the tunability of the block parameters in the generated code. However, these blocks use constant sample time in the model regardless of parameter tunability.

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

Compatibility Considerations

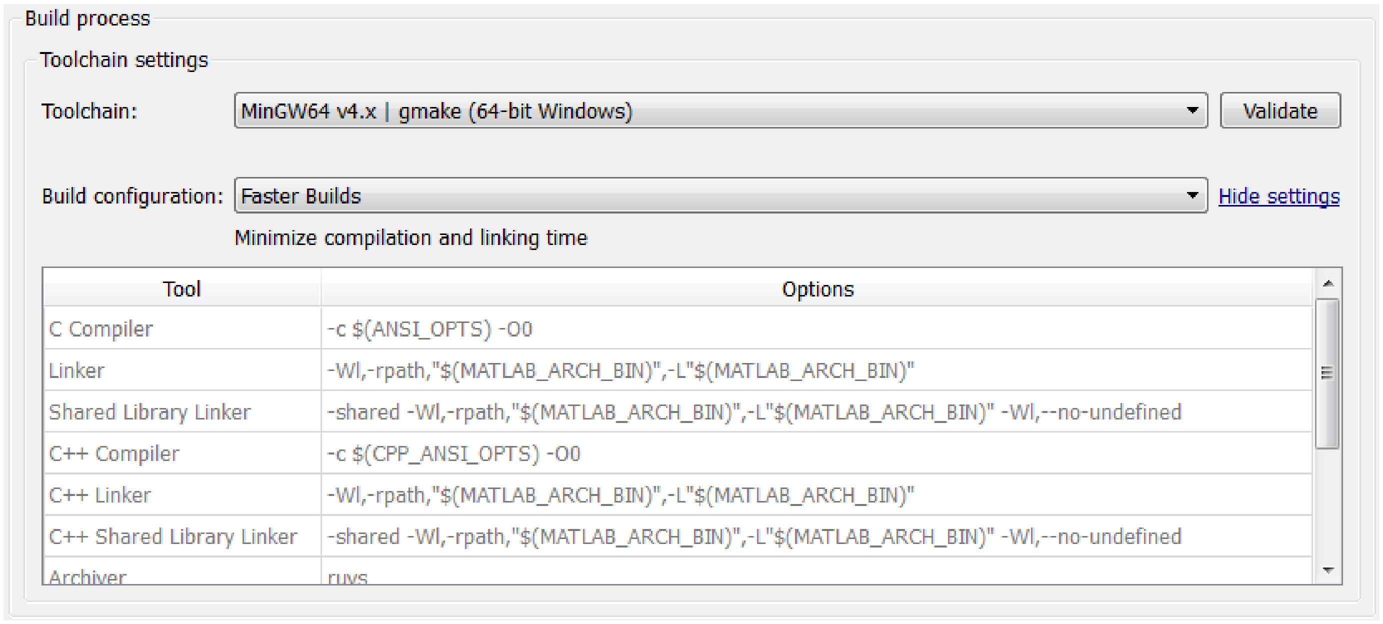
If you use scripts that change code generation targets, confirm that the scripts do not alter the setting for **Default parameter behavior**.

Code Generation

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

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

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



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

Build configuration: Faster Builds

Minimize compilation and linking time

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

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

In R2015b, the code generator introduces support for non-US-ASCII characters in compilable portions of generated source code. The code generator processes strings without loss of information or character corruption by replacing unrepresented characters of the user default encoding with an escape sequence of the form `&#x`*`code-unit`*`;`. *`code-unit`* is the hexadecimal value for the unrepresented character. For example, the code generator replaces the Japanese full-width Katakana letter ア with the escape sequence `&#x30A2;`. Cases where escape sequence replacements occur include:

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

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

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

For more information, see Internationalization and Code Generation.

Hardware Implementation Selection: Quickly generate code for popular embedded processors

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

This list summarizes the R2015b changes and new behavior:

- By default, the **Hardware Implementation** pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only.
- If you use Simulink without a Simulink Coder license, initially parameters on the **Hardware Implementation** pane are disabled. To enable them, click **Enable hardware specification**. The parameters remain enabled for the current MATLAB session.
- By default, the **Hardware board** list includes: `None` or `Determine by Code Generation system target file`, and `Get Hardware Support Packages`. After installing a hardware support package, the list also includes corresponding hardware board names.
- If you select a hardware board name, parameters for that board appear in the dialog box display.
- Lists for the **Device vendor** and **Device type** parameters have been updated to reflect hardware that is available on the market. The default **Device vendor** and **Device type** are `Intel` and `x86-64 (Windows64)`, respectively.
- If Simulink Coder is installed, the revised **Hardware Implementation** pane identifies the system target file that you selected on the **Code Generation** pane.
- A **Device details** option provides a way to display parameters for setting details such as number of bits and byte ordering.
- To specify target hardware for a Simulink support package, select a value from **Configuration Parameters** > **Hardware Implementation** > **Hardware board**. Before R2015b, you selected **Tools** > **Run on Target Hardware** > **Prepare to run**. Then, you selected a value from **Configuration Parameters** > **Run on Target Hardware** > **Target hardware**.
- To specify target hardware for an Embedded Coder support package, select a value from **Configuration Parameters** > **Hardware Implementation** > **Hardware board**. Before R2015b, you selected a value from **Configuration Parameters** > **Code Generation** > **Target hardware**.
- The **Test hardware** section was removed. Configure test hardware from the Configuration Parameters list view. Set `ProdEqTarget` to `off`, which enables parameters for configuring test hardware details.
- If you set **Configuration Parameters** > **Code Generation** > **System target file** to `ert.tlc`, `realtime.tlc`, or `autosar.tlc`, the default setting for **Configuration Parameters** >

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

For more information, see Hardware Implementation Pane.

Compatibility Considerations

Starting in R2015b:

- By default, the **Hardware Implementation** pane lists **Hardware board**, **Device vendor**, and **Device type** parameter fields only. To view parameters for setting details, such as number of bits and byte ordering, click **Device details**.
- The following devices appear on the **Hardware Implementation** pane only for models that you create with a version of the software earlier than R2015b. These devices are considered legacy devices.

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

  In R2015b, if you open a model configured for a legacy device and change the **Device type** setting, you cannot select the legacy device again.
- Device parameter **Signed integer division rounds to** is set to `Zero` instead of `Undefined`. For some cases, numerical differences can occur in results produced with `Zero` versus `Undefined` for simulation and code generation.

  This change does not apply to legacy devices.
- To associate a new model with an existing configuration set that has the following characteristics, configure the model to use the same hardware device as the existing model.

  - The model consists of a model reference hierarchy. Models in the hierarchy use different configuration sets.
  - The existing configuration set was saved as a script and associated with a configuration set variable.

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

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

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

- Toggling the **Generate code only** option on or off between builds no longer forces regeneration of source code. For example, suppose that you clear **Generate code only** after generating code, and make no other model change that affects code generation. The next build detects that up-to-date source code is already available and compiles the code without regenerating it.
- In a model reference hierarchy, the **Generate code only** setting of the top model overrides the **Generate code only** setting of referenced models. This change relaxes the constraint that the **Generate code only** setting must be consistent within a model reference hierarchy. The change helps prevent unnecessary regeneration of referenced model code.
- If you have an Embedded Coder license, running a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation for a top model or Model block no longer requires that you clear **Generate code only**. See Embedded Coder release note Removal of Generate code only parameter restriction.

Toolchain approach with custom targets added

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

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

Build configuration setting can affect setting for toolchain

When using the toolchain approach to build a model, you can configure the code generator to use a specific toolchain and build configuration. On the Configuration Parameters dialog box, you can set values for **Code Generation** > **Toolchain** and **Code Generation** > **Build configuration**.

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

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

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

Deployment

External mode MEX-file build requires sl_services library

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

Compatibility Considerations

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

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

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

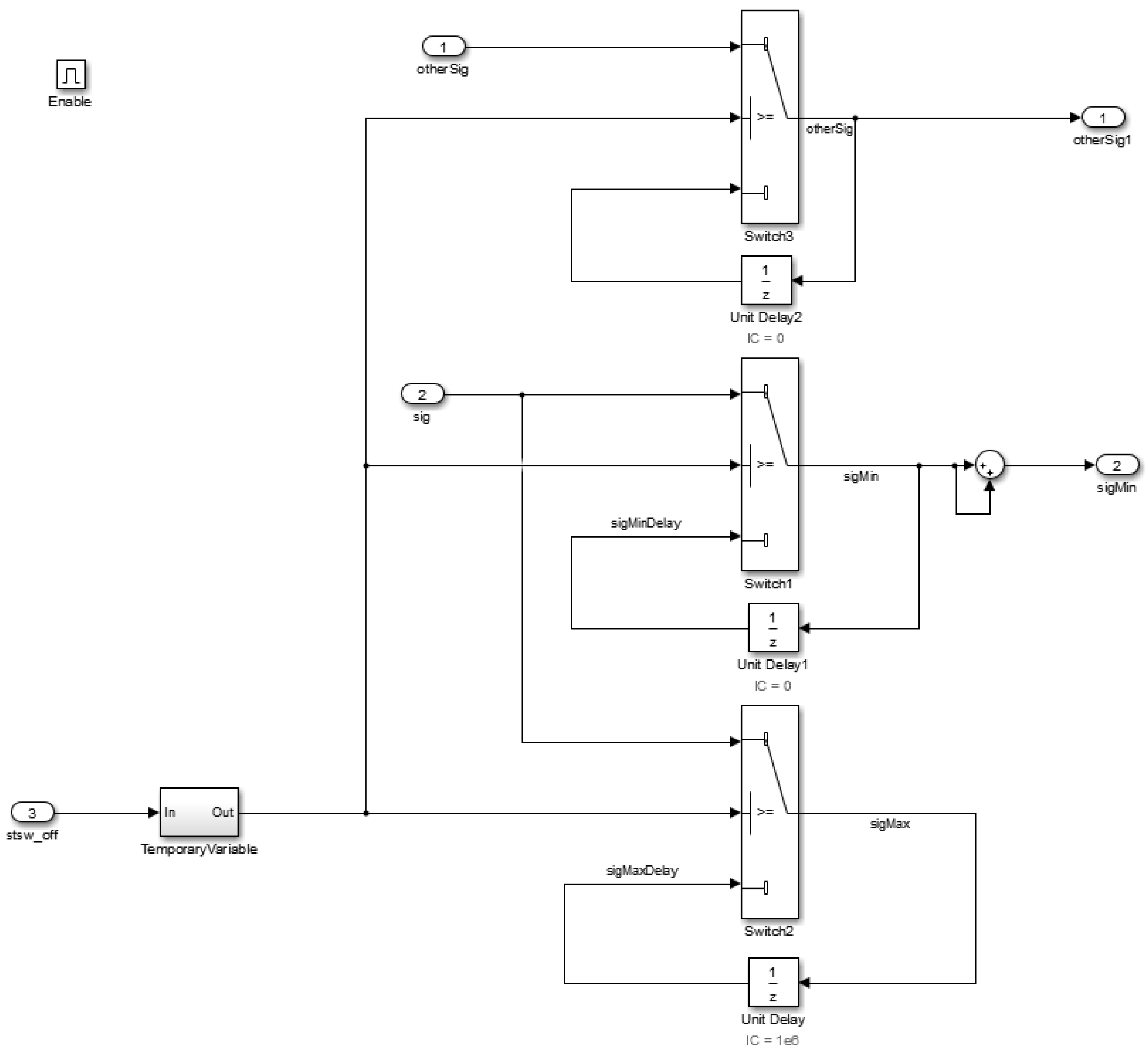
Performance

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

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

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

Consider the following Enabled Subsystem block named `S4`. This subsystem contains three Switch blocks named `Switch 1`, `Switch 2`, and `Switch 3`.



In R2015a, the code generator produced the following code:

```
void IfElseWithState_R2015a_step(void)
{
  real32_T rtb_sigMin;
  if (enable) {
    if (stsw_off) {
      rtb_sigMin = sig;
    } else {
      rtb_sigMin = IfElseWithState_R2015a_DW.UnitDelay1_DSTATE;
    }

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

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

In R2015b, the code generator produces the following code:

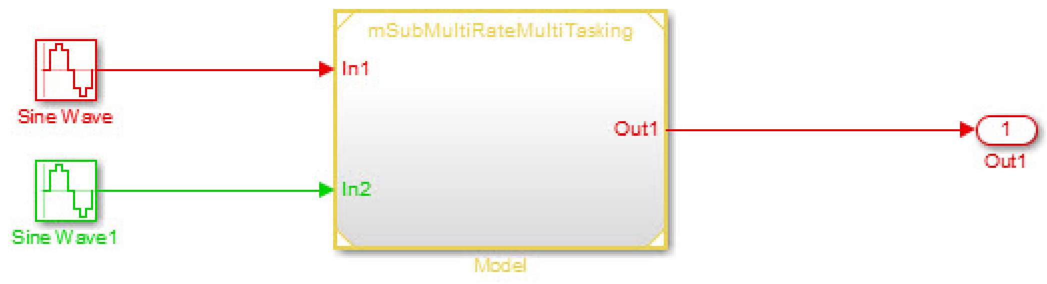
```
    void IfElseWithState_step(void)
{
  real32_T rtb_sigMin;
  if (enable) {
    if (stsw_off) {
      rtb_sigMin = sig;
      sigMax = sig;
      otherSigCapture = otherSig;
    } else {
      rtb_sigMin = IfElseWithState_DW.UnitDelay1_DSTATE;
      sigMax = IfElseWithState_DW.UnitDelay_DSTATE;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

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

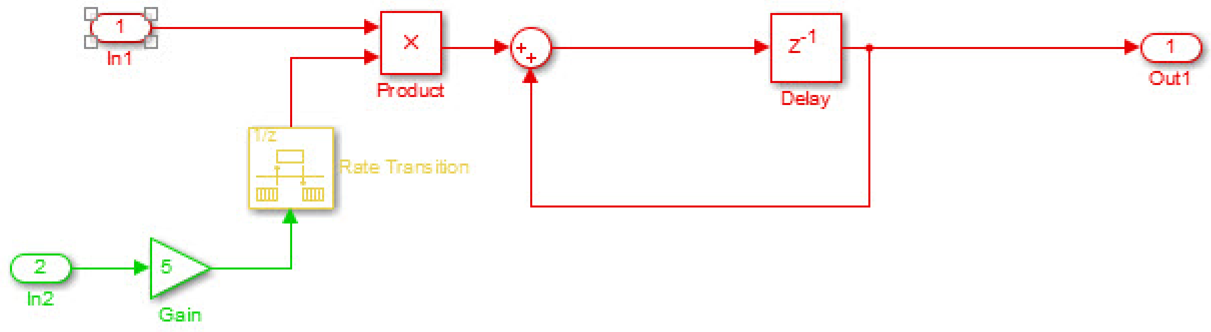
In R2015a, the generated code contained two `if-else` statements because of the sum operation that followed `Switch 1`. In R2015b, there is one `if-else` statement for all three Switch blocks. The code generator combines the `if-else` statements because they share the same condition. The outcome of the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

More efficient code for multirate models

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

Consider the following model named `mTopMultiRateMultiTasking`. This model contains a referenced model, `mSubMultiRateMultiTasking`, that executes at two different sample times.





In R2015a, for `mSubMultiRateMultiTasking`, the code generator produced this code:

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

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

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

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

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

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

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

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

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

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

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

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

Check bug reports for issues and fixes

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

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

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

R2015a

Model Architecture and Design

Command-line APIs for protected models

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

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

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

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

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

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

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

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

Improved use of workers for faster parallel builds

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

Usability enhancements for protected models

Open support for protected models

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

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

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

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

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

Protected model support for Rapid Accelerator mode

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

Platform independence for protected model Web view

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

No code reuse for function-call subsystems with mask parameters

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

1. You clear the model configuration parameter **Inline parameters**.
2. You use a mask parameter of any kind in any of the subsystem instances.

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

Compatibility Considerations

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

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

Check bug reports for issues and fixes

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

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

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

R2014b

Model Architecture and Design

Code generation for Simulink Function and Function Caller blocks

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

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

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

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

Option to suppress generation of shared constants

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

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

Usability enhancements for protected models

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

- Previously, you created a protected model only from a command-line API or from the context menu of a model reference block. In R2014b, from the Simulink Editor menu bar, you can select **File** > **Export Model To** > **Protected Model** to create a protected model from the current model. For more information on protecting a model, see Protect a Referenced Model.
- Previously, to update the configuration options for a protected model, you deleted the current protected model and recreated it with the new options. In R2014b, using the `Modifiable` option for `Simulink.ModelReference.protect`, you can create a protected model that is modifiable by an authorized user. An authorized user can then use the `Simulink.ModelReference.modifyProtectedModel` and the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` functions to update the options for the protected model and provide a password for authorization.
- The protected model report provides more information. For each possible functionality that the protected model can support, the **Supported functionality** section reports `On`, `Off`, or `On with password protection`. The new **Licenses required** section lists all licenses required to run the protected model. For more information on the protected model report, see Protected Model Report.
- The read-only view functionality for protected models now uses the model Web view features introduced in R2014a. For more information on the Web view, see the Simulink Report Generator documentation.

Data, Function, and File Definition

Enumerated data type size control

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

For example, the code generator uses this class definition:

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

to generate this code:

```
typedef int8_T Colors;

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

For more information, see Enumerations.

Vector and matrix expressions as model argument values

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

Code Generation

Option to separate output and update functions for GRT targets

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

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

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

Highlighted configuration parameters from Code Generation Advisor reports

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

License requirement for viewing code generation report

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

Compatibility Considerations

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

Improved report generation performance

When you use `codegen.rpt` to create code generation reports with Simulink Report Generator™, in the Report Options dialog box on the **Properties** pane, the **Compile model to report on compiled information** check box is selected by default. With this option, the software updates a model only once when creating the report. You get much faster report generation, especially for models with many atomic subsystems. For more information, see Document Generated Code with Simulink Report Generator.

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

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

- Intel IPP for x86-64 (Windows)

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

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

- Intel IPP
- Intel IPP/SSE with GNU99 extensions

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

See Choose a Code Replacement Library.

Deployment

Support for Eclipse IDE and Desktop Targets has been removed

Simulink Coder support for Eclipse™ IDE has been removed.

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

Compatibility Considerations

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

Performance

Block reduction optimization improvement

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

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

Check bug reports for issues and fixes

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

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

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

R2014a

Model Architecture and Design

Custom post-processing function for protected models

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

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

Context-sensitive help for the Create Protected Model dialog box

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

Data, Function, and File Definition

C++ class generation

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

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

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

1. Select the `C++` language for your model.
2. Select `C++ class` code interface packaging for your model.
3. Generate model code.
4. Examine the C++ class interfaces in the generated files and the HTML code generation report.

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

Simpler behavior for tuning all parameters and support for referenced models

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

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

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

Compatibility Considerations

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

Block parameter expression	Code generated previously	Code generated in R2014a

Expressions referencing global variables (e.g., `K+1`)	Variable name is not preserved. Block parameter name is preserved.
	`struct Parameters_model_ {`
	`  real_T Gain_Gain; // Expression: K+1`
	`}`
	`y = model_P.Gain_Gain*u;`
		Expression is considered tunable. Variable name is preserved in code and is tunable.
		`real_T K = 2.0;`
		`y = (K+1)*u;`

Expressions referencing mask parameters for nonreusable subsystems (e.g., `MP*3`), the value of `MP` being a nontunable expression.	Variable name is not preserved. Block parameter name is preserved.
	`struct Parameters_model_ {`
	`  real_T Gain_Gain; // Expression: MP*3`
	`}`
	`y = model_P.Gain_Gain*u;`
		Expression is considered tunable. Variable name is substituted by parameter value.
		`struct Parameters_model_ {`
		`  real_T Subsystem_MP;`
		`}`
		`y = (model_P.Subsystem_MP * 3) * u;`

Expressions referencing model arguments (resp. mask parameters) for referenced models (resp. reusable subsystems) (e.g., `Arg+1`)	Variable name is not preserved. Block parameter name is preserved.
	`struct Subsystem {`
	`  Gain_Gain; // Expression: Arg+1`
	`}`
	`y = model_P.Subsystem1.Gain_Gain*u;`
		Variable name is preserved as an argument name.
		`subsystem(y, u, rtp_Arg) {`
		`  y = (rtp_Arg+1)*u;`
		`}`

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

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

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

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

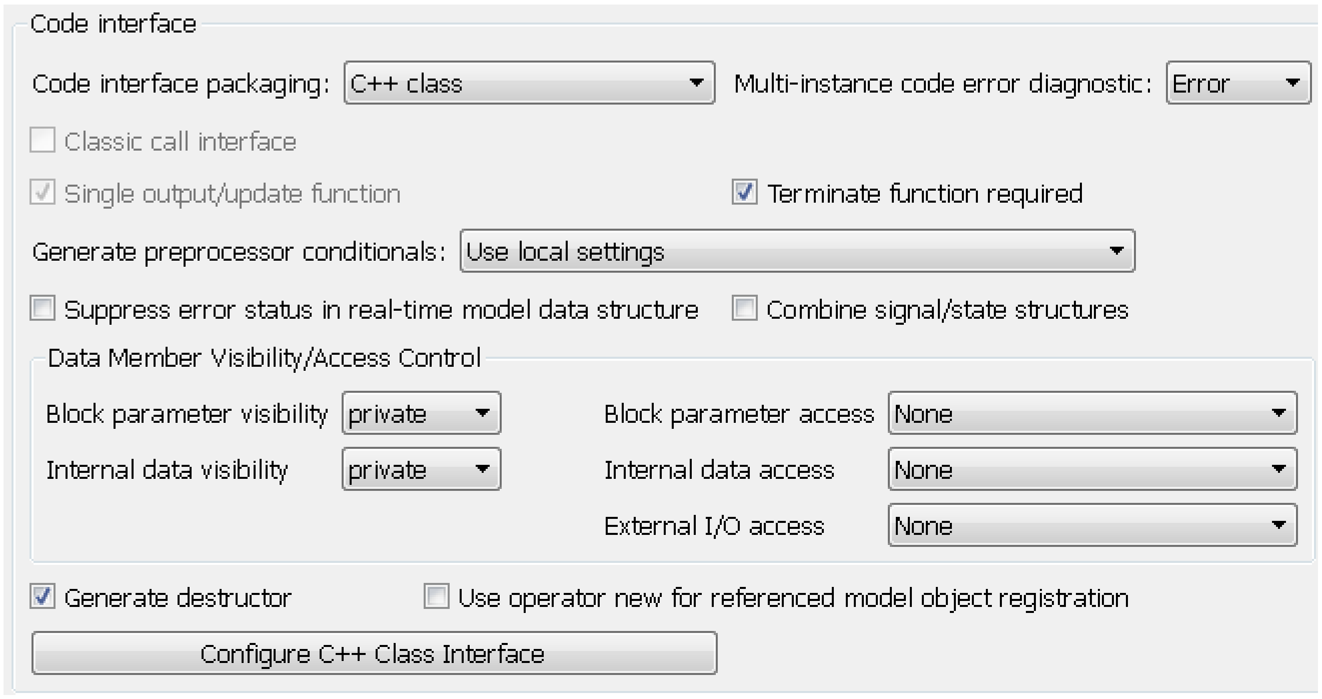
Improved control of C and C++ code interface packaging

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

- The new model parameter **Code interface packaging** (`CodeInterfacePackaging`) selects the packaging for the C or C++ code interface generated for your model. The possible values are:
  - `Nonreusable function`
  - `Reusable function`
  - `C++ class` (available if **Language** is set to `C++`)

Note As described in “C++ class generation” on page 20-3, C++ class code generation is now available to GRT-based models as well as ERT-based models.

- The model option **Generate reusable code** (`MultiInstanceERTCode`) has been removed. Setting the value of **Code interface packaging** to `Reusable function` or `Nonreusable function` is equivalent to selecting or clearing **Generate reusable code** in releases before R2014a.
- The model parameter **Reusable code error diagnostic** has been renamed to **Multi-instance code error diagnostic**. The parameter now supports GRT models and C++ class code generation. For more information, see “Multi-instance code error diagnostic for reusable function code and C++ class code” on page 20-6.
- Parameters within the **Code interface** subpane have been regrouped and relocated to improve pane navigation and code interface configuration workflows. The following figure shows the **Code interface** subpane when **Code interface packaging** is set to `C++ class` for an ERT model.


Code interface

Code interface packaging: C++ class　　Multi-instance code error diagnostic: Error

☐ Classic call interface

☑ Single output/update function　　☑ Terminate function required

Generate preprocessor conditionals: Use local settings

☐ Suppress error status in real-time model data structure　　☐ Combine signal/state structures

Data Member Visibility/Access Control

Block parameter visibility	private	Block parameter access	None
Internal data visibility	private	Internal data access	None
		External I/O access	None

☑ Generate destructor　　☐ Use operator new for referenced model object registration

Compatibility Considerations

- For GRT and ERT-based models, selecting the **Language** (`TargetLang`) value `C++` now provides two possible forms of C++ code interface packaging:
  - If you set **Code interface packaging** (`CodeInterfacePackaging`) to `C++ class`, the build generates a C++ class interface to model code. The generated interface encapsulates required model data into C++ class attributes and model entry point functions into C++ class methods.
  - If you set **Code interface packaging** to a value other than `C++ class`, the build generates C ++ compatible `.cpp` files containing model interfaces enclosed within an `extern "C"` link directive. This behavior is equivalent to code generation with **Language** set to `C++` before R2014a.
- A script created before R2014a might select C++ `extern "C"` code packaging for a model by using `set_param` to set `TargetLang` to `C++`. Beginning in R2014a, setting `TargetLang` to `C++`

selects only the C++ language, and does not alter the C++ code interface packaging. To produce the same result as before R2014a, update the script to set both `TargetLang` and `CodeInterfacePackaging`.

- For ERT-based models, the **Language** value `C++ (Encapsulated)` is no longer available. To configure encapsulated C++ class code generation, set **Language** to `C++` and set **Code interface packaging** to `C++ class`. This pair of settings is equivalent to setting **Language** to `C++ (Encapsulated)` in releases before R2014a.
- For GRT and ERT-based models, the model option **Generate reusable code** (`MultiInstanceERTCode`) has been removed. Setting the value of **Code interface packaging** to `Reusable function` or `Nonreusable function` is equivalent to selecting or clearing **Generate reusable code** in releases before R2014a.

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

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

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

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

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

Compatibility Considerations

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

  If you load a GRT model created before R2014a, for which reusable code generation is selected, by default, code generation now applies strict multi-instance code requirements to the model during code generation, and the build might fail. If the build fails, examine the condition that caused the error message. Decide whether to reset **Multi-instance code error diagnostic** to `Warning` or `None`, or leave **Multi-instance code error diagnostic** set to `Error` and modify the model to remove the condition.
- Before R2014a, the setting of the ERT model parameter **Reusable code error diagnostic** was ignored if a model was configured to do all of the following:

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

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

Removal of TRUE and FALSE from rtwtypes.h

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

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

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

Code Generation

Independent configuration selections for standard math and code replacement libraries

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

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

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

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

For more information, see:

- Language
- Standard math library
- Code replacement library

Compatibility Considerations

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

- When you load a model created with an earlier version:
  - The `CodeReplacementLibrary` parameter setting remains the same unless previously set to `C89/C90 (ANSI)`, `C99 (ISO)`, `C++ (ISO)`, `Intel IPP (ANSI), or Intel IPP (ISO)`. In these cases, Simulink Coder software sets `CodeReplacementLibrary` to `None` or `Intel IPP`.
  - Simulink Coder software sets the new `TargetLangStandard` parameter to `C89/C90 (ANSI)`, `C99 (ISO)`, or `C++03 (ISO)`, depending on the previous `CodeReplacementLibrary` setting.

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

- When you select a code replacement library provided by MathWorks after you load a model, the code generator can produce different code than in previous versions depending on the `TargetLangStandard` setting. Verify generated code.
- When you export a model created in R2014a, the Simulink Coder software:
  - Uses the `TargetLangStandard` setting to map to the closest available code replacement library or the default library in the previous version, if `CodeReplacementLibrary` is set to `None` or `Intel IPP`.
  - Otherwise, ignores the `TargetLangStandard` parameter.

Generated code compilation using LCC-64 bit on Windows hosts

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

Improved code integration of shared utility files

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

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

Optimized inline constant expansion

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

rtwtypes.h included before tmwtypes.h

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

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

Constant block output value used when in nonreusable subsystem

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

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

Deployment

Support for Eclipse IDE and Desktop Targets will be removed

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

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

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

Compatibility Considerations

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

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

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

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

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

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

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

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

MATLAB command line, use the `set_param` command to set the `SystemTargetFile` parameter to `'tornado.tlc'`.

Compatibility Considerations

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

Performance

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

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

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

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

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

Optimized reusable subsystem inputs

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

Check bug reports for issues and fixes

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

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

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

R2013b

Model Architecture and Design

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

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

- Model viewing
- Simulation
- Code generation

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

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

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

Simulink Coder checks in Model Advisor

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

Data, Function, and File Definition

Imported data can be shared

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

Some shared items using imported data types include:

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

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

Compatibility Considerations

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

Readability improved for constant names

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

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

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

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

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

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

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

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

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

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

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

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

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

Compatibility Considerations

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

Code Generation

Optimized code for long long data type

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

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

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

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

For more information, see Model Parameters.

<LEGAL> tokens removed from comments in generated code

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

Deployment

Compiler toolchain interface for automating code generation builds

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

The toolchain approach enables you to:

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

To use the toolchain approach:

1. Open the Configuration Parameters dialog box for Simulink model by pressing **Ctrl+E**.
2. In Configuration Parameters, select the **Code Generation** pane.
3. Click the **Browse** button for the **System target file** parameter, and select one of the following:
   - `grt.tlc — Generic Real-Time Target` (default)
   - `ert.tlc — Embedded Coder` (Requires the Embedded Coder product)
   - `ert_shrlib.tlc — Embedded Coder (host-based shared library target)` (Requires the Embedded Coder product)

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

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

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

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

For more information, see:

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

Compatibility Considerations

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

Log data on Linux-based target hardware

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

This capability:

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

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

Note This feature requires a Simulink Coder license.

Modified file locations and commands for rebuilding external mode MEX files

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

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

Compatibility Considerations

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

Performance

Reduced data copies for bus signals with global storage

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

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

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

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

Customization

Support for user-authored MATLAB system objects

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

TLC Options removed from Configuration Parameters dialog box

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

Check bug reports for issues and fixes

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

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

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

R2013a

Data, Function, and File Definition

Optimized interfaces for Simulink functions called in Stateflow

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

Shortened system-generated identifier names

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

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

Before R2013a	In R2013a	Type
`BlockIO, B`	`B`	Type
`ExternalInputs`	`ExtU`	Type
`ExternalInputSizes`	`ExtUSize`	Type
`ExternalOutputs`	`ExtY`	Type
`ExternaOutputSizes`	`ExtYSize`	Type
`Parameters`	`P`	Type
`ConstBlockIO`	`ConstB`	`Const` Type
`MachineLocalData`	`MachLocal`	`Const` Type
`ConstParam, ConstP`	`ConstP`	`Const` Type, Global Variable
`ConstParamWithInit`, `ConstWithInitP`	`ConstInitP`	`Const` Type, Global Variable
`D_Work`, `DWork`	`DW`	Type, Global Variable
`MassMatrixGlobal`	`MassMatrix`	Type, Global Variable
`PrevZCSigStates`, `PrevZCSigState`	`PrevZCX`	Type, Global Variable
`ContinuousStates`, `X`	`X`	Type, Global Variable
`StateDisabled`, `Xdis`	`XDis`	Type, Global Variable
`StateDerivatives`, `Xdot`	`XDot`	Type, Global Variable
`ZCSignalValues`, `ZCSignalValues`	`ZCV`	Type, Global Variable
`DefaultParameters`	`DefaultP`	Global Variable
`GlobalTID`	`GlobalTID`	Global Variable
`InvariantSignals`	`Invariant`	Global Variable

Before R2013a	In R2013a	Type
`Machine`	`MachLocal`	Global Variable
`NSTAGES`	`NSTAGES`	Global Variable
`Object`	`Obj`	Global Variable
`TimingBridge`	`TimingBrdg`	Global Variable
`U`	`U`	Global Variable
`USize`	`USize`	Global Variable
`Y`	`Y`	Global Variable
`YSize`	`YSize`	Global Variable

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

Before R2013a	In R2013a	Type
`rtB`, `B`	`B`	Type, Global Variable
`rtC`, `C`	`ConstB`	Type, Global Variable
`rtDW`, `DW`	`DW`	Type, Global Variable
`rtMdlrefDWork` , `MdlrefDWork`	`MdlRefDW`	Type, Global Variable
`rtP`, `P`	`P`	Type, Global Variable
`rtRTM`, `RTM`	`RTM`	Type, Global Variable
`rtX`, `X`	`X`	Type, Global Variable
`rtXdis`, `Xdis`	`XDis`	Type, Global Variable
`rtXdot`, `Xdot`	`XDot`	Type, Global Variable
`rtZCE`, `ZCE`	`ZCE`	Type, Global Variable
`rtZCSV`, `ZCSV`	`ZCV`	Type, Global Variable

For more information, see Construction of Generated Identifiers.

Code Generation

Shared utility name consistency across builds with maximum identifier length control

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

Code Generation Advisor available on menu bar

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

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

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

Code generation build when reusable library subsystem link status changes

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

Protected models usable in model reference hierarchies

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

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

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

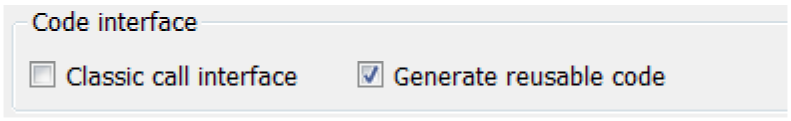
Deployment

Simplified multi-instance code with support for referenced models

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

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

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


Code interface

☐ Classic call interface ☑ Generate reusable code

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

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

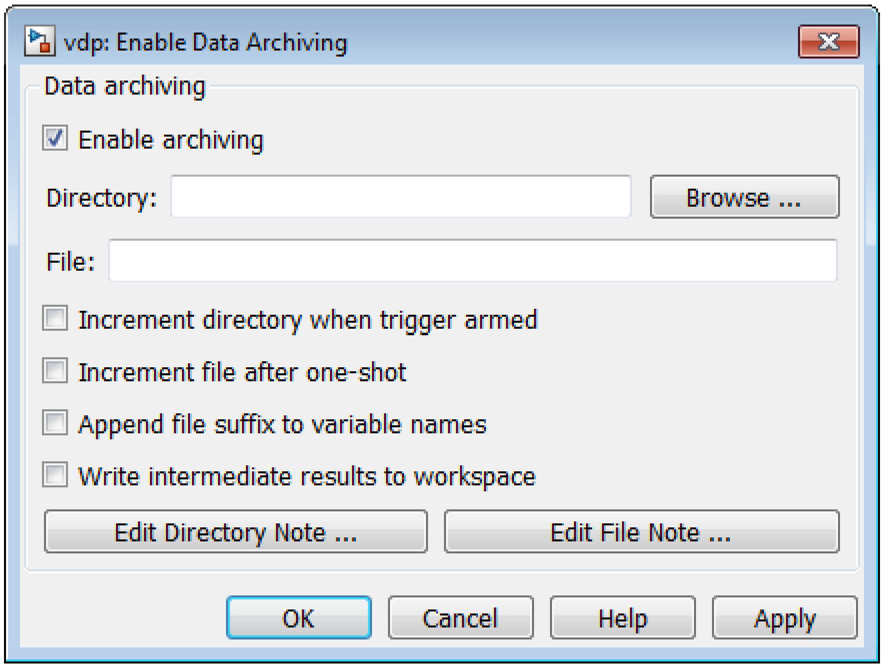
With the new GRT model option **Generate reusable code**, you can generate and deploy multi-instance code for your model without selecting a specialized target, and you can include referenced models in your model design.

Note Use of the `grt_malloc.tlc` target is no longer recommended. For more information, see "GRT malloc target to be removed in future release" on page 22-8.

External mode control panel improvements and C API access

Improved External mode graphical controls

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


vdp: Enable Data Archiving

Data archiving

[✓] Enable archiving

Directory:

File:

[ ] Increment directory when trigger armed

[ ] Increment file after one-shot

[ ] Append file suffix to variable names

[ ] Write intermediate results to workspace

To view the improved External mode dialog boxes, open a model and select **Code** > **External Mode Control Panel**.

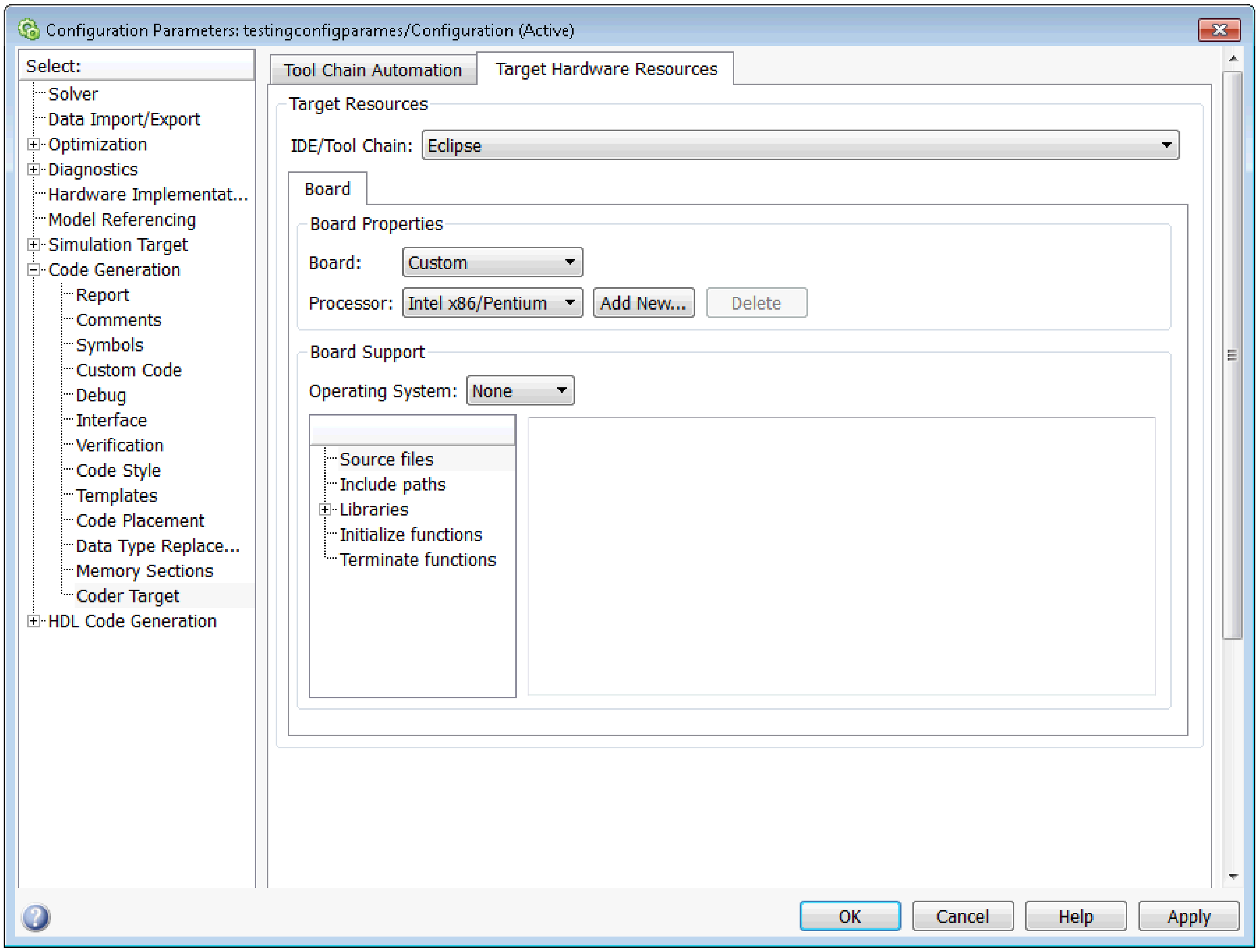
C API access from External mode simulations

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

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

Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog

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



IDE/Tool Chain: Eclipse

Board: Custom

Processor: Intel x86/Pentium

Operating System: None

The Target Preferences block has been removed from the Desktop Targets block library.

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

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

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

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

Support ending for Eclipse IDE in a future release

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

GRT malloc target to be removed in future release

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

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

Compatibility Considerations

If you are using the system target file `grt_malloc.tlc` to generate reentrant code with dynamic memory allocation, switch to using `grt.tlc` with the model configuration option **Generate reusable code**. As described in “Simplified multi-instance code with support for referenced models” on page 22-5, the **Generate reusable code** option offers several advantages over the GRT malloc target, including a simple multi-instance call interface and support for model reference hierarchies. For more information, see the help for **Generate reusable code**.

Customization

MakeRTWSettingsObject model parameter removed

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

Compatibility Considerations

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

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

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

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

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

Check bug reports for issues and fixes

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

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

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

R2012b

Unified and simplified code interface for ERT and GRT targets

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

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

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

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

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

Note

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

Compatibility Considerations

If you use a GRT-based target with a static main program, and if you configure your models with the simplified call interface that was made available to GRT targets in R2012a (that is, you do not use the model option **Classic call interface**), you must update your static main program to be compatible with the R2012b static main changes. Use the code in `matlabroot/rtw/c/src/common/rt_main.c` as an example. The following sections outline some of the key changes to look for.

Error status handling

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

- For ERT targets, a stop simulation request caused the error status to be set to `Simulation finished`. The main program (`ert_main.c`) treated this error status as a non-error, while treating all other non-NULL status values as errors.

- For GRT targets, a stop simulation request did not cause the error status to be set (it remained NULL). The main program treated all non-NULL status values as errors.

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

Unused macros

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

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

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

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

Multitasking functions

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

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

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

Convenient packNGo dialog for packaging generated code and artifacts

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

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

If you select **Package code and artifacts**, the build process runs the `packNGo` function after code generation to package generated code and artifacts for relocation. Selecting **Package code and**

artifacts also enables the **Zip file name** parameter for specifying a `.zip` file name. The default file name is *`model`*`.zip`. (*`model`* represents the name of the top model for which code is being generated.)

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

Reusable code for subsystems shared by referenced models

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

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

Code generation for protected models for accelerated simulations and host targets

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

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

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

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

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

Reduction of cyclomatic complexity with virtual bus expansion

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

Simplifying for loop control statements

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

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

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

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

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

Constant parameters generated as individual constants to shared location

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

Code efficiency enhancements

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

- Removing a root-level outport data copy in the generated code when data is from a Stateflow chart. This enhancement reduces RAM and ROM consumption and improves execution speed.
- Removing a data copy for masked subsystems when a parameter is a matrix data type. This enhancement reduces RAM and ROM consumption and improves execution speed.
- Removing a limitation where the joint presence of initial value and function prototype control prevent removal of the root-level outport data copy in the generated code. The outport data copy is removed when the initial value is zero. This enhancement reduces RAM and ROM consumption and improves execution speed.
- Removing an unnecessary global variable generated by a For Each Subsystem or as a result from the selected configuration parameter **Pack Boolean data into bitfields**. In R2012b, the variable is removed from the global block structure which reduces global RAM.

Optimized code generation of Delay block

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

Search improvements in code generation report

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

GRT template makefile change for MAT-file logging support

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

Compatibility Considerations

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

To update your TMF, do the following:

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

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

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

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

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

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

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

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

Compatibility Considerations

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

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

Model rtwdemo_f14 removed from software

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

Compatibility Considerations

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

Check bug reports for issues and fixes

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

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

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

R2012a

Simplified Call Interface for Generated Code

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

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

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

Note

- The **Classic call interface** (`GRTInterface`) option is available for both GRT-based and ERT-based models. For Embedded Coder users, it replaces the ERT model option **GRT-compatible call interface** (`GRTInterface`).
- For new GRT and ERT models, the **Classic call interface** option is cleared by default. New models use the simplified call interface.
- For GRT models created before R2012a, **Classic call interface** is selected by default. Existing GRT models can continue to use the pre-R2012a GRT-specific call interface.

Incremental Code Generation for Top-Level Models

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

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

1. Skips model code regeneration.
2. Still calls build process hooks, including `*STF*_make_rtw_hook` functions and the post code generation command.
3. Reruns the makefile to make sure external dependencies are recompiled and relinked.

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

Minimal Header File Dependencies with packNGo Function

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

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

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

ASAP2 Enhancements for Model Referencing and Structured Data

Ability to Merge ASAP2 Files Generated for Top and Referenced Models

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

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

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

ASAP2 File Generation for Test Pointed Signals and States

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

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

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

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

For more information, see Customize an ASAP2 File.

ASAP2 File Generation for Tunable Structure Parameters

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

For more information, see Customize an ASAP2 File.

Serial External Mode Communication Using rtiostream API

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

Improved Data Transfer in External Mode Communication

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

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

Changes for Desktop IDEs and Desktop Targets

- “Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE” on page 24-4
- “Limitation: Parallel Builds Not Supported for Desktop Targets” on page 24-4

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

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

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

Limitation: Parallel Builds Not Supported for Desktop Targets

The Simulink Coder product provides an API for MATLAB Distributed Computing Server™ and Parallel Computing Toolbox products. The API allows these products to perform parallel builds that reduce build time for referenced models. However, the API does not support parallel builds for models whose **System target file** parameter is set to `idelink_ert.tlc` or `idelink_grt.tlc`. Thus, you cannot perform parallel builds for Desktop Targets.

Code Generation Report Enhancements

Post-build Report Generation

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

Generate Code Generation Report Programmatically

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

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

Searching in the Code Generation Report

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

New Reserved Keywords for Code Generation

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

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

Improved MAT-File Logging

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

rtwdemo_f14 Being Removed in a Future Release

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

Compatibility Considerations

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

New and Enhanced Demos

The following demos have been enhanced in R2012a:

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

Check bug reports for issues and fixes

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

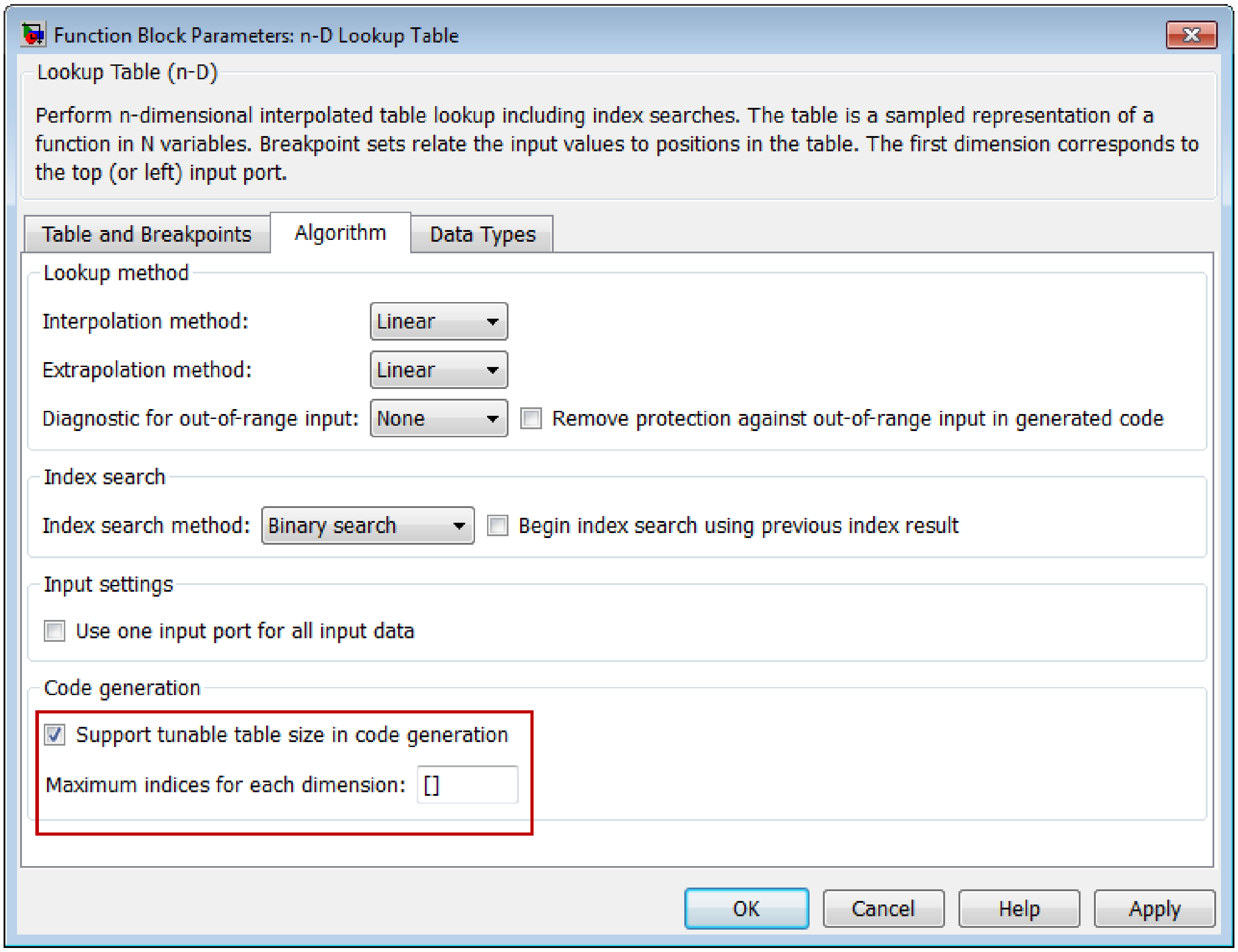
The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

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

R2011b

n-D Lookup Table Block Supports Tunable Table Size

The n-D Lookup Table block provides new parameters for specifying a tunable table size in the generated code.



Function Block Parameters: n-D Lookup Table

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Lookup method

Interpolation method: Linear

Extrapolation method: Linear

Diagnostic for out-of-range input: None

☐ Remove protection against out-of-range input in generated code

Index search

Index search method: Binary search

☐ Begin index search using previous index result

Input settings

☐ Use one input port for all input data

Code generation

☑ Support tunable table size in code generation

Maximum indices for each dimension: []

This enhancement enables you to change the size and values of your lookup table and breakpoint data without regenerating or recompiling the code.

Complex Output Support in Generated Code for the Trigonometric Function Block

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

Code Optimizations for the Combinatorial Logic Block

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

Benefits include:

- Reuse of variables
- Dead code elimination
- Constant folding
- Expression folding

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

Code Optimizations for the Product Block

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

Benefit	Small matrices (2-by-2 to 5-by-5)	Medium matrices (6-by-6 to 20-by-20)	Large matrices (larger than 20-by-20)
Faster code execution time	Yes, much faster	No, slightly slower	Yes, faster
Reduced ROM and RAM usage	Yes, for real values	Yes, for real values	Yes, for real values
Reuse of variables	Yes	Yes	Yes
Dead code elimination	Yes	Yes	Yes
Constant folding	Yes	Yes	Yes
Expression folding	Yes	Yes	Yes
Consistency with MATLAB Coder	Yes	Yes	Yes

Compatibility Considerations

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

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

Enhanced MISRA C Code Generation Support for Stateflow Charts

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

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

In R2011b, that line has been modified to:

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

This enhancement supports MISRA C™ 2004, rule 10.1.

Change for Constant Sample Time Signals in Generated Code

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

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

Compatibility Considerations

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

New Code Generation Advisor Objective for GRT Targets

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

Improved Integer and Fixed-Point Saturating Cast

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

Generate Multitasking Code for Concurrent Execution on Multicore Processors

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

See Configuring Models for Targets with Multicore Processors.

Changes for Desktop IDEs and Desktop Targets

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

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

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

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

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

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

- Processor architecture of the target
- Optimized library support for the target
- The type and number of TFL replacements in the generated algorithmic code

Evaluate the performance of your application before and after using the TFL.

To use the SIMD capabilities with GCC and Intel processors, enable the `Intel IPP/SSE (GNU)` TFL. See Code Replacement Library (CRL).

Reserved Keyword UNUSED_PARAMETER

The Simulink Coder software adds the `UNUSED_PARAMETER` macro to the reserved keywords list for code generation. To view the complete list, see Reserved Keywords. In R2011b, code generation now defines `UNUSED_PARAMETER` in `rt_defines.h`. Previously, it was defined in *`model`*`_private.h`.

Target API for Verifying MATLAB® Distributed Computing Server™ Worker Configuration for Parallel Builds

R2010b added the ability to use remote workers in MATLAB® Distributed Computing Server™ configurations for parallel builds of model reference hierarchies. This introduced the possibility that parallel workers might have different configurations, some of which might not be compatible with a specific Simulink Coder target build. For example, the required compiler might not be installed on a worker system.

R2011b provides a programming interface that target authors can use to automatically check the configuration of parallel workers and, if the parallel workers are not set up as required, take action, such as throwing an error or reverting to sequential builds. For more information, see Support Model Referencing.

For more information about parallel builds, see Reduce Build Time for Referenced Models.

License Names Not Yet Updated for Coder Product Restructuring

The Simulink Coder and Embedded Coder license name strings stored in `license.dat` and returned by the `license ('inuse')` function have not yet been updated for the R2011a coder product restructuring. Specifically, the `license ('inuse')` function continues to return `'real-time_workshop'` for Simulink Coder and `'rtw_embedded_coder'` for Embedded Coder, as shown below:

```
>> license('inuse')
matlab
matlab_coder
real-time_workshop
rtw_embedded_coder
simulink
>>
```

The license name strings intentionally were not changed, in order to avoid license management complications in situations where Release 2011a or higher is used alongside a preR2011a release in a common operating environment. MathWorks plans to address this issue in a future release.

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

New and Enhanced Demos

The following demos have been enhanced in R2011b:

Demo...	**Now...**
`rtwdemo_pmsmfoc_script`	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine

Check bug reports for issues and fixes

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

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

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

R2011a

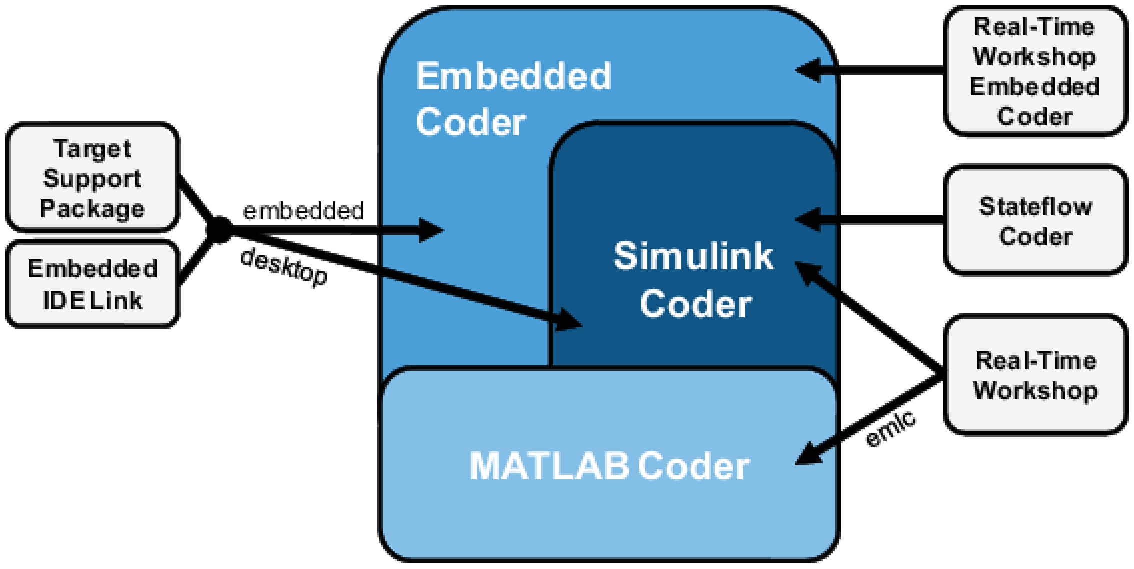
Coder Product Restructuring

Product Restructuring Overview

In R2011a, the Simulink Coder product combines and replaces the Real-Time Workshop® and Stateflow Coder products. Additionally,

- The Real-Time Workshop facility for converting MATLAB code to C/C++ code, formerly referred to as Embedded MATLAB® Coder, has migrated to the new MATLAB Coder product.
- The previously existing products Embedded IDE Link™ and Target Support Package™ have been integrated into the new Simulink Coder and Embedded Coder products.

The following figure shows the R2011a transitions for C/C++ code generation related products, from the R2010b products to the new MATLAB Coder, Simulink Coder, and Embedded Coder products.



The following sections address topics related to the product restructuring.

Resources for Upgrading from Real-Time Workshop or Stateflow Coder

If you are upgrading to Simulink Coder from Real-Time Workshop or Stateflow Coder, review information about compatibility and upgrade issues at the following locations:

- *Release Notes for Simulink Coder* (latest release), "Compatibility Summary" section
- In the Archived documentation on the MathWorks web site, select R2010b, and view the following tables, which are provided in the release notes for Real-Time Workshop and Stateflow Coder:

  - *Compatibility Summary for Real-Time Workshop Software*
  - *Compatibility Summary for Stateflow and Stateflow Coder Software*

  These tables provide compatibility information for releases up through R2010b.
- If you use the Embedded IDE Link or Target Support Package capabilities that now are integrated into Simulink Coder and Embedded Coder, go to the Archived documentation, select R2010b, and view the corresponding tables for each product:
  - *Compatibility Summary for Embedded IDE Link*
  - *Compatibility Summary for Target Support Package*

You can also refer to the rest of the archived documentation, including release notes, for the Real-Time Workshop, Stateflow Coder, Embedded IDE Link, and Target Support Package products.

Migration of Embedded MATLAB Coder Features to MATLAB Coder

In R2011a, the function `codegen` replaces the Real-Time Workshop function `emlc`. The `emlc` function still works in R2011a but generates a warning, and will be removed in a future release. For more information, see Migrating from Real-Time Workshop emlc Function.

Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder

In R2011a, the capabilities formerly provided by the Embedded IDE Link and Target Support Package products have been integrated into Simulink Coder and Embedded Coder. The follow table summarizes the transition of the Embedded IDE Link and Target Support Package hardware and software support into coder products.

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
Embedded IDE Link	Altium® TASKING		x
	Analog Devices® VisualDSP++®		x
	Eclipse IDE	x	x
	Green Hills® MULTI®		x
	Texas Instruments® Code Composer Studio™		x
Target Support Package	Analog Devices Blackfin®		x
	ARM		x
	Freescale® MPC5xx		x
	Infineon® C166®		x
	Texas InstrumentsC2000™		x
	Texas InstrumentsC5000®		x
	Texas InstrumentsC6000®		x
	Linux OS	x	x
	Windows OS	x	
	VxWorks RTOS		x

User Interface Changes Related to Product Restructuring

Some user interface changes were made as part of merging the Real-Time Workshop and Stateflow Coder products into Simulink Coder. They include:

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

Simulink Graphical User Interface Changes

Where...	Previously...	Now...
Configuration Parameters dialog box	**Real-Time Workshop** pane	**Code Generation** pane
Model diagram window	**Tools > Real-Time Workshop**	**Tools > Code Generation**
Subsystem context menu	**Real-Time Workshop**	**Code Generation**
Subsystem Parameters dialog box	Following parameters on main pane: • **Real-Time Workshop system code** • **Real-Time Workshop function name options** • **Real-Time Workshop function name** • **Real-Time Workshop file name options** • **Real-Time Workshop file name (no extension)**	On new **Code Generation** pane and renamed: • **Function packaging** • **Function name options** • **Function name** • **File name options** • **File name (no extension)**

Changes for Desktop IDEs and Desktop Targets

- "Feature Support for Desktop IDEs and Desktop Targets" on page 26-4
- "Location of Blocks for Desktop Targets" on page 26-5
- "Location of Demos for Desktop IDEs and Desktop Targets" on page 26-5
- "Multicore Deployment with Rate Based Multithreading" on page 26-6

Feature Support for Desktop IDEs and Desktop Targets

The Simulink Coder software provides the following features as implemented in the former Target Support Package and Embedded IDE Link products:

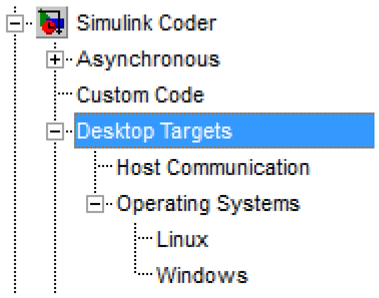
- Automation Interface
- External Mode

- Multicore Deployment with Rate Based Multithreading
- Makefile Generation (XMakefile)

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

Location of Blocks for Desktop Targets

Blocks from the former Target Support Package product and Embedded IDE Link product are now located in Simulink Coder under Desktop Targets.

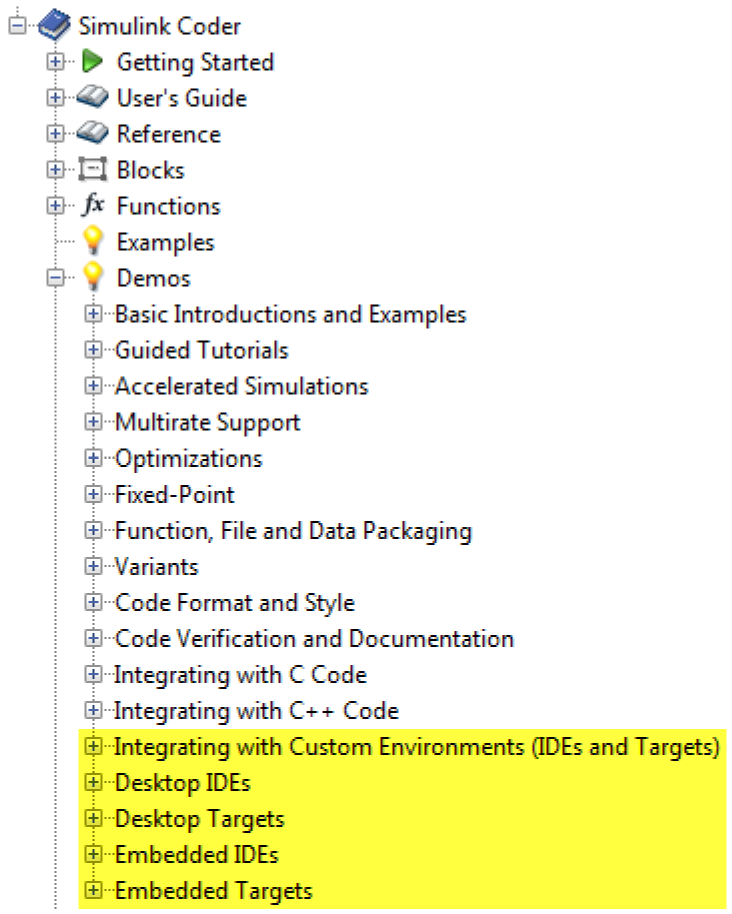


Desktop Targets includes the following types of blocks:

- Host Communication
- Operating Systems
  - Linux
  - Windows

Location of Demos for Desktop IDEs and Desktop Targets

Demos from the former Target Support Package product and Embedded IDE Link product now reside under Simulink Coder product help. Click the expandable links, as shown.



Simulink Coder
- Getting Started
- User's Guide
- Reference
- Blocks
- Functions
- Examples
- Demos
  - Basic Introductions and Examples
  - Guided Tutorials
  - Accelerated Simulations
  - Multirate Support
  - Optimizations
  - Fixed-Point
  - Function, File and Data Packaging
  - Variants
  - Code Format and Style
  - Code Verification and Documentation
  - Integrating with C Code
  - Integrating with C++ Code
  - Integrating with Custom Environments (IDEs and Targets)
  - Desktop IDEs
  - Desktop Targets
  - Embedded IDEs
  - Embedded Targets

Multicore Deployment with Rate Based Multithreading

You can deploy rate-based multithreading applications to multicore processors running Windows and Linux. This feature potentially improves performance by taking advantage of multicore hardware resources.

Also see the Running Target Applications on Multicore Processors user's guide topic.

Code Optimizations for Discrete State-Space Block, Product Block, and MinMax Block

The Simulink Coder build process uses a new technique to provide more efficient code for the following blocks:

- Discrete State-Space
- Product (element-wise matrix operations)

Benefits include:

- Reuse of variables
- Dead code elimination
- Constant folding

- Expression folding

For example, in previous releases, temporary buffers were created to carry concatenated signals for these blocks. In R2011a, the build process eliminates unnecessary temporary buffers and writes the concatenated signal to the downstream global buffer directly. This enhancement reduces the stack size and improves code execution speed.

The build process also provides more efficient code for the MinMax block. In R2011a, expression folding is enhanced with several local optimizations that enable more aggressive folding. This enhancement improves code efficiency for foldable matrix operations.

Ability to Share User-Defined Data Types Across Models

In previous releases, user-defined data types that were shared among multiple models generated duplicate type definitions in the `*model*_types.h` file for each model. R2011a provides the ability to generate user-defined data type definitions into a header file that can be shared across multiple models, removing the need for duplicate copies of the data type definitions. User-defined data types that you can set up in a shared header file include:

- Simulink data type objects that you instantiate from the classes `Simulink.AliasType`, `Simulink.Bus`, `Simulink.NumericType`, and `Simulink.StructType`
- Enumeration types that you define in MATLAB code

For more information, see Share User-Defined Data Types Across Models.

C API Provides Access to Root-Level Inputs and Outputs

The C API now provides programmatic access to root-level inputs and outputs. This allows you to log and monitor the root-level inputs and outputs of a model while you run the code generated for the model. To generate C API code for accessing root-level inputs and outputs at run time, select the model option **Generate C API for: root-level I/O**.

Macros for accessing C API generated structures are located in *`matlabroot`*`/rtw/c/src/rtw_capi.h` and *`matlabroot`*`/rtw/c/src/rtw_modelmap.h`, where *`matlabroot`* represents your MATLAB installation root.

For more information, see Generate C API for: root-level I/O and Data Interchange Using the C API.

ASAP2 File Generation Supports Standard Axis Format for Lookup Tables

In previous releases, ASAP2 file generation for lookup table blocks supported the Fix Axis and Common Axis formats, but not the Standard Axis format, a format in which axis points are global in code but not shared among tables. R2011a adds support for Standard Axis format.

For more information, see Define ASAP2 Information for Lookup Tables.

ASAP2 File Generation Enhancements for Computation Methods

Custom Names for Computation Methods

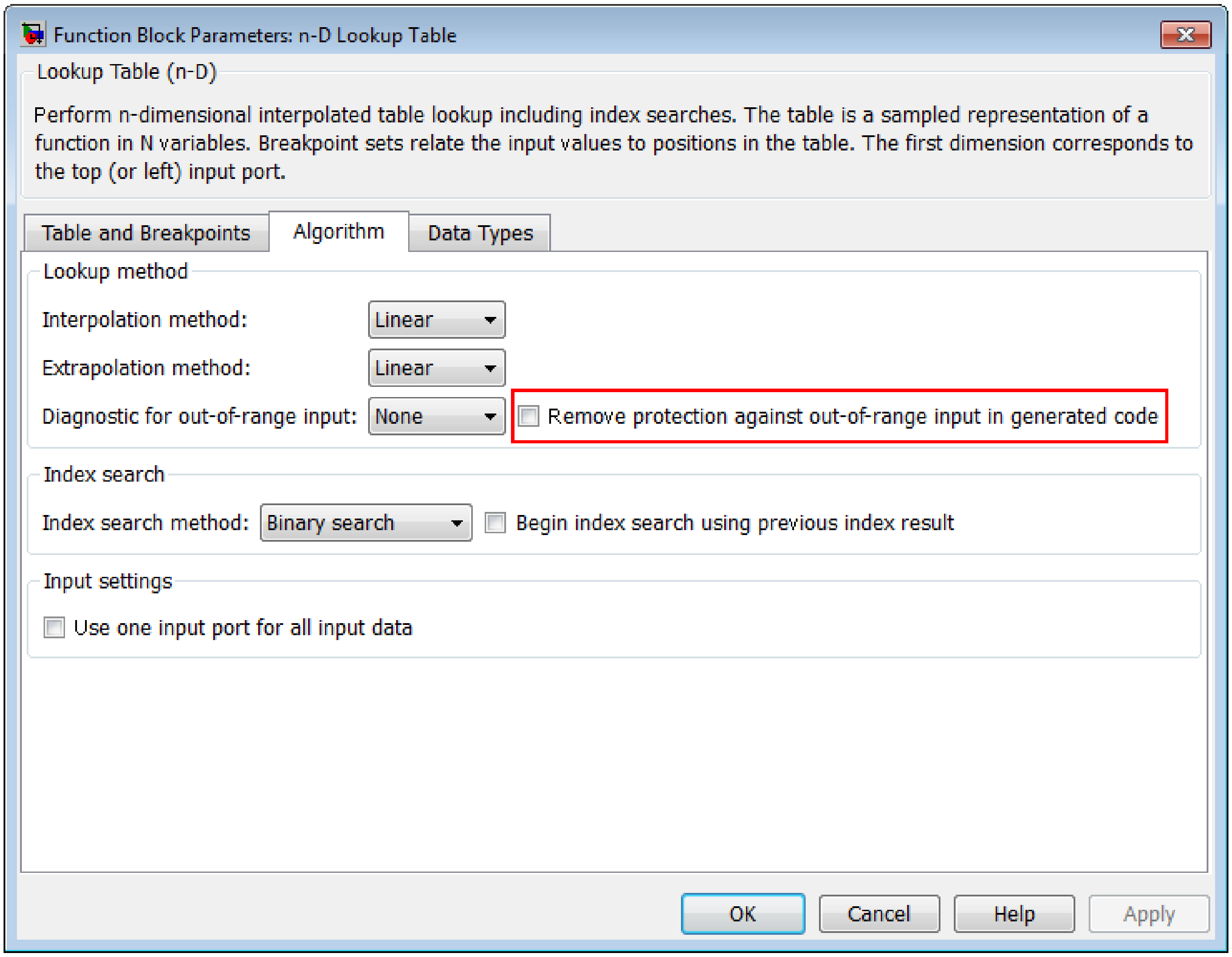
In generated ASAP2 files, computation methods translate the electronic control unit (ECU) internal representation of measurement and calibration quantities into a physical model oriented representation. R2011a adds the MATLAB function `getCompuMethodName`, which you can use to customize the names of computation methods. You can provide names that are more intuitive, enhancing ASAP2 file readability, or names that meet organizational requirements. For more information, see Customize Computation Method Names.

Ability to Suppress Computation Methods for FIX_AXIS When Not Required

Versions 1.51 and later of the ASAP2 specification state that for certain cases of lookup table axis descriptions (integer data type and no doc units), a computation method is not required and the Conversion Method parameter must be set to the value `NO_COMPU_METHOD`. Beginning in R2011a, you can control whether or not computation methods are suppressed when not required, using the Target Language Compiler (TLC) option `ASAP2GenNoCompuMethod`. For more information, see Suppress Computation Methods for FIX_AXIS.

Lookup Table Block Option to Remove Input Range Checks in Generated Code

When the breakpoint input to a Prelookup, 1-D Lookup Table, 2-D Lookup Table, or n-D Lookup Table block always falls within the range of valid breakpoint values, you can disable range checking in the generated code. By selecting **Remove protection against out-of-range input in generated code** on the block dialog box, your code can be more efficient.


Function Block Parameters: n-D Lookup Table

Lookup Table (n-D)

Perform n-dimensional interpolated table lookup including index searches. The table is a sampled representation of a function in N variables. Breakpoint sets relate the input values to positions in the table. The first dimension corresponds to the top (or left) input port.

Table and Breakpoints | Algorithm | Data Types

Lookup method

Interpolation method: Linear

Extrapolation method: Linear

Diagnostic for out-of-range input: None

☐ Remove protection against out-of-range input in generated code

Index search

Index search method: Binary search

☐ Begin index search using previous index result

Input settings

☐ Use one input port for all input data

Reentrant Code Generation for Stateflow Charts That Use Events

When you generate code for Stateflow charts that use events, the code does not use a global variable to keep track of the currently active event. Elimination of this global variable enables the code to be reentrant, which allows you to:

- Deploy your code in multithreading environments
- Share the same algorithm with different persistent data
- Compile code that uses function variables that are too large to fit on the stack

In previous releases, reentrant code generation was not possible for charts that used events.

Redundant Check Code Removed for Stateflow Charts That Use Temporal Operators

When you generate code for Stateflow charts that use temporal operators, the code excludes redundant checks for `tick` events and input events that are always true. This enhancement enables the code to be more efficient and applies to temporal operators `after`, `before`, `at`, `every`, and `temporalCount`.

In previous releases, the code generated for a temporal logic expression such as `after(x,tick)` would check for two conditions:

```
(event == tick) && (counter > x)
```

In R2011a, the code generated for `after(x,tick)` checks only for when the temporal counter exceeds x:

```
(counter > x)
```

This enhancement does not apply when a chart with multiple input events has super-step semantics enabled.

Support for Multiple Asynchronous Function Calls Into a Model Block

Simulink and Simulink Coder software now support multiple asynchronous function calls into a Model block. This capability relies in part on the new Asynchronous Task Specification block.

The Asynchronous Task Specification block, in combination with a root-level Inport block, allows you to specify an asynchronous function-call input to a Model block. After placing this block at the output port of each root-level Inport block that outputs a function-call signal, select **Output function call** on the **Signal Attributes** pane of the Inport block. The Inport block then accepts function-call signals. You can use Asynchronous Task Specification blocks to specify parameters for the asynchronous task associated with the respective Inport blocks.

Note Use the new function call API, `LibBlockExecuteFcnCall`, to make function calls from an asynchronous source block to reference model destination blocks.

Note The demo model `rtwdemo_async_mdlreftop` shows how you can simulate and generate code for asynchronous events on a real-time multitasking system, using asynchronous function calls as Model block inputs.

Changes to ver Function Product Arguments

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

- The new argument `'simulinkcoder'` returns information about the installed version of the Simulink Coder product.
- The argument `'rtw'` works but now returns information about Simulink Coder instead of Real-Time Workshop. The software also displays the following message:

  ```
  Warning: Support for ver('rtw') will be removed in a future release.
  Use ver('simulinkcoder') instead.
  ```
- The argument `'coder'`, which previously returned information about the installed version of the Stateflow Coder product, no longer works. The software displays a “not found” warning.

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

Compatibility Considerations

If a script calls the `ver` function with the`'rtw'` argument or the`'coder'` argument, update the script appropriately. For example, you can update the `ver` call to use the `'simulinkcoder'` argument, or remove the `ver` call.

Updates to Target Language Compiler (TLC) Semantics and Diagnostic Information

Updates to TLC simplifies semantics and produces diagnostic information when using the scope resolution operator (`::`) and built-in function `EXISTS(::)`.

- If *`var`* cannot be resolved in global scope, `::`*`var`* errors out
- If *`var`* can only be resolved in local scope, `EXISTS(::`*`var`*`)` returns false
- Diagnostic information highlights problematic TLC coding

For more information, see Introducing the Target Language Compiler.

Change to Terminate Function for a Target Language Compiler (TLC) Block Implementation

Previously, the code generator attempted to execute the `Terminate` function from the TLC implementation of a block, even if the function did not exist. Now, the code generator only attempts to execute a `Terminate` function if it is defined in the TLC implementation of a block. In the case where the TLC implementation of a block includes a secondary TLC file, which includes a `Terminate` function, that `Terminate` function no longer executes.

New and Enhanced Demos

The following demos have been added in R2011a:

Demo...	Shows How You Can...
`rtwdemo_async_mdlreftop`	Simulate and generate code for asynchronous events on a real-time multitasking system, using asynchronous function calls as Model block inputs.

The following demos have been enhanced in R2011a:

Demo...	Now...
`vipstabilize_fixpt_beagleboard` `videostabilization_host_templ`	Use the new Video Capture block to simulate or capture a video input signal in the Video Stabilization demo.

Check bug reports for issues and fixes

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