Embedded Coder® Release Notes

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Embedded Coder® Release Notes

Contents

R2013b

R2011a

R2013b

Version: 6.5

New Features: Yes

Bug Fixes: Yes

Code Generation from MATLAB Code

Software-in-the-loop verification for MATLAB Coder

Use software-in-the-loop (SIL) execution to verify production-ready source code. SIL execution involves compiling and running static library code on your host computer. Through SIL execution, you can reuse test vectors developed for your MATLAB® functions to verify the numerical behavior of static library code.

Previously, verification was restricted to code generated for execution only within MATLAB. Now, in MATLAB, you can compile and run standalone code on the host computer through a MATLAB SIL interface.

You can run a SIL execution:

- Using the MATLAB Coder™ project interface. See "Software-in-the-Loop (SIL) Execution Through the Project Interface".
- From the command line. See "Software-in-the-Loop (SIL) Execution From the Command Line".

Custom generated identifiers for `emxArray` utility functions

You can customize generated identifiers for `emxArray` (embeddable `mxArray`) utility functions. When you generate code that uses variable-size data, the code generation software exports utility functions to interact with `emxArray` data structures. Customize utility function identifiers to avoid name collisions when a function that uses variable-size data calls a library function that uses variable-size data.

To customize generated identifiers for `emxArray` utility functions:

- In a project

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

- At the command line

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

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

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

Model Architecture and Design

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

R2013b enhances AUTOSAR modeling, component import, and programmatic control. See also “Support for AUTOSAR release 4.0.3 XML and generated code” on page 14.

Enhanced modeling and simulation of AUTOSAR multiple runnables

In previous releases, AUTOSAR multi-runnables were modeled as function-call subsystems within a wrapper subsystem in a Simulink® model. To generate code, you right-clicked the wrapper subsystem and exported functions.

Beginning in R2013b, you can model AUTOSAR multi-runnables as function-call subsystems at the top level of a model, without having to use a wrapper subsystem. When you generate code for the model, each function-call subsystem representing a runnable appears in the model C code as a callable model entry-point function.

You can simulate multiple runnables in an AUTOSAR software component in multiple simulation modes. For example:

- For a periodic runnable, you can edit the properties of the function-call subsystem inport to set the sample time for a periodic event simulation.
- For a non-periodic runnable, you can edit the **Data Import/Export** pane of the Configuration Parameters dialog box to set up data loading for an asynchronous event simulation.

For more information, see “Configure Multiple Runnables”.

Enhanced ARXML import of AUTOSAR software component internal behavior

The AUTOSAR software component importer tool can automatically import the internal behavior of a multi-runnable AUTOSAR software component into a Simulink model. You can use the `createComponentAsModel` method of the class `arxml.importer` to specify that internal behavior be imported. For example:

```
>> obj = arxml.importer(`mySWC.arxml');
>> obj.createComponentAsModel('/pkg/swc', 'CreateInternalBehavior', true)
```

The importer:

- Adds subsystem blocks in the model and maps them to corresponding runnables imported from the AUTOSAR software component.
- Adds signal lines in the model and maps them to corresponding interrunnable variables (IRVs) imported from the AUTOSAR software component.

Ability to model AUTOSAR mode receiver ports and events

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

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

For more information, see “Configure AUTOSAR Mode Receiver Ports and Events”.

AUTOSAR dual-scaled parameter

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

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

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

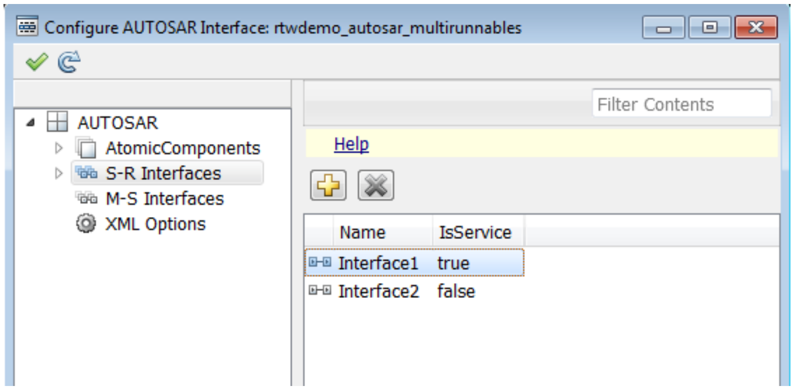
For more information, see `AUTOSAR.DualScaledParameter`.

Programmatic interface for configuring AUTOSAR properties and Simulink-AUTOSAR mapping

R2013b provides a programmatic interface for configuring AUTOSAR properties and Simulink mapping information using MATLAB functions. You can programmatically get, set, add, and remove the same component properties and mapping information displayed in the **AUTOSAR Properties** Explorer and **Simulink-AUTOSAR Mapping** Explorer views of the Configure AUTOSAR Interface dialog box.

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

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



Name	IsService
Interface1	true
Interface2	false

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

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

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

Reorganization of Model Advisor Embedded Coder checks

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

- "Simulink Coder Checks"
- "Embedded Coder Checks"

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

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

For more information, see:

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

Protected model Web view

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

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

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

RTW.AutosarInterface class to be removed in a future release

Compatibility Considerations: Yes

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

Compatibility Considerations

If you are using the `RTW.AutosarInterface` class and methods to programmatically control and validate the AUTOSAR configuration of a model, you should migrate to using the new AUTOSAR property and mapping functions listed in “AUTOSAR Component Development”. The new functions are designed to work with the component properties and mapping information displayed in the **AUTOSAR Properties** Explorer and **Simulink-AUTOSAR Mapping** Explorer views of the Configure AUTOSAR Interface dialog box.

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

Compatibility Considerations: Yes

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

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

Compatibility Considerations

If you are using `createComponentAsSubsystem` or `createOperationAsConfigurableSubsystems`, you should migrate to using the top model oriented approach described in “Configure Multiple Runnables”.

Data, Function, and File Definition

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

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

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

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

For more information on files created during code generation, see “Files Created During the Build Process”.

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

Name space control for scoping C++ encapsulated model classes

R2013b adds name space control for scoping model classes generated using C++ encapsulation. You can use the **Namespace** parameter in the Configure C++ Encapsulation Interface dialog box to specify a name space for a model class. If specified, the name space is emitted in the generated code for the model class. To scope the C++ encapsulated model classes in a model reference hierarchy, you can specify a different name space for each referenced model. For more information, see "Use Name Spaces to Scope C++ Encapsulated Model Classes".

For more information on configuring C++ encapsulated model classes, see "Configure C++ Encapsulation Interfaces Using Graphical Interfaces".

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

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

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

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

Shared utility naming control

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

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

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

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

For more information see

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

Expanded support for identifier names

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

- You can include `$A` in naming for local temporary variables where previously it was supported only for local block output variables and field names of global types. For more information, see “Identifier Format Control”, “Local temporary variables” and “Field name of global types”.

- You can customize identifier names by specifying `$A` which maps to the data type replacement setting. Previously the generated code changed the types, but not the identifier names. For more information, see “Data Type Replacement”.

Code Generation

Support for AUTOSAR release 4.0.3 XML and generated code

R2013b adds AUTOSAR release 4.0.3 support, as follows:

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

See also “Enhanced modeling of AUTOSAR runnables and modes, and improved ARXML import of internal behavior” on page 4.

Indent style and size control for code generation

R2013b adds options for customizing code appearance. The following new parameters are located in the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane.

- **Indent style**: Specify `K&R` or `Allman` style for the placement of braces.
- **Indent size**: Specify the number of characters per indent level. Choose from `2`–`8` characters.

For more information on configuring code style parameters, see “Control Code Style”.

Subsystem functions return value in generated code

In the Subsystem Block Parameters dialog box, on the **Code Generation** tab, if you specify

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

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

Model reference step function void input and output arguments

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

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

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

Deployment

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

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

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

Support package for ARM Cortex processors

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

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

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

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

Support package for STMicroelectronics STM32F4-Discovery Board

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

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

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

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

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

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

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

Wind River VxWorks 6.9 support

Compatibility Considerations: Yes

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

To use this feature, install the corresponding support package:

1. In a MATLAB Command Window, enter `supportPackageInstaller`.

2. Use Support Package Installer to install the *Embedded Coder Support Package for Wind River® VxWorks RTOS*.

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

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

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

Compatibility Considerations

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

Support package for Texas Instruments C2000 processors

Compatibility Considerations: Yes

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

To use this feature, install the corresponding support package:

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

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

- "C2802x (c2802xlib)" block library

- “C2803x (c2803xlib)” block library
- “C2806x (c2806xlib)” block library
- “C280x (c280xlib)” block library
- “C281x (c281xlib)” block library
- “C2834x (c2834xlib)” block library
- “C28x3x (c2833xlib)” block library
- “Memory Operations” block library
- “Optimization — C28x™ DMC (c28xdmclib)” block library
- “Optimization — C28x IQmath (tiiqmathlib)” block library
- “RTDX™ Instrumentation (rtdxBlocks)” block library
- “Scheduling” block library
- “Target Communication” block library

For more information about this feature, see the “ Support Package for Texas Instruments C2000 Processors ” topic.

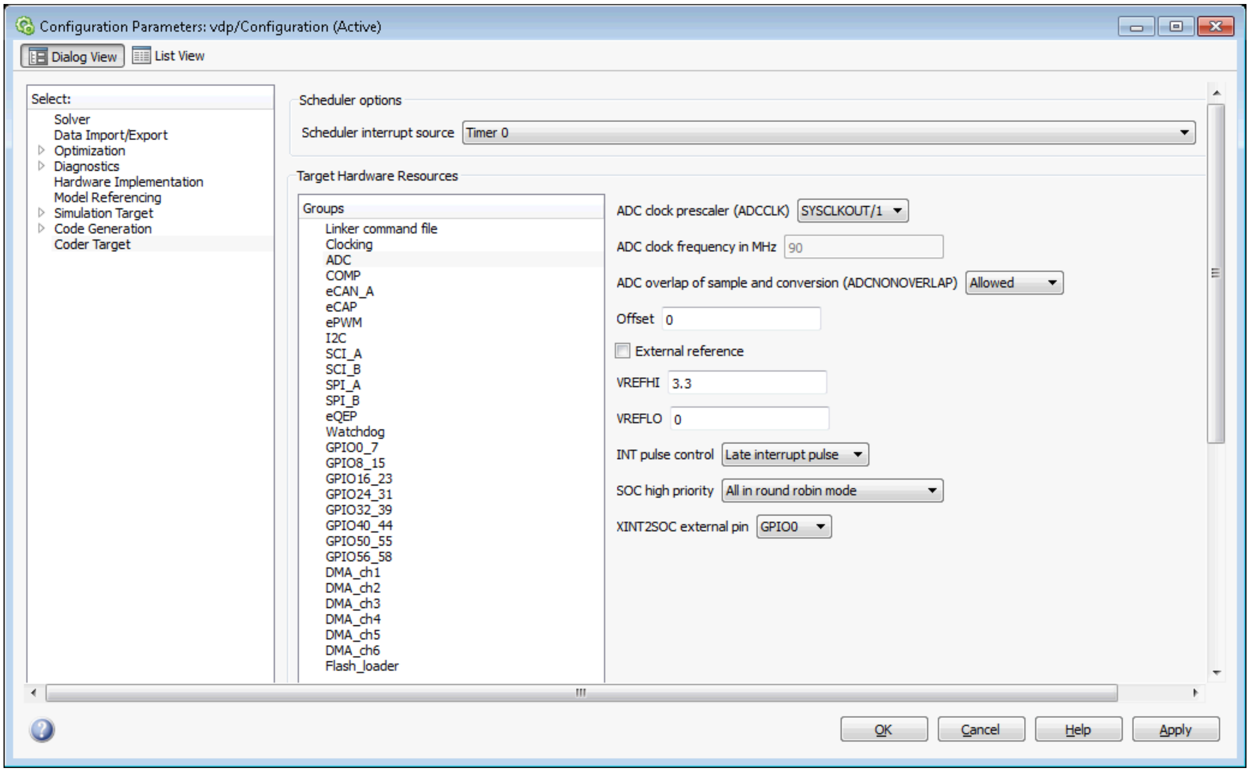
Compatibility Considerations

Previous versions of Embedded Coder software had built-in support for C2000 processors. The current version of Embedded Coder does not have built-in support for C2000 processors.

To get support for C2000 processors, install *Embedded Coder Support Package for Texas Instruments C2000 Processors*, as described in the preceding section.

Coder Target pane in Configuration Parameters dialog box

You can use the Coder Target pane to configure target hardware settings for your model.



Scheduler options

Scheduler interrupt source: Timer 0

Target Hardware Resources

ADC clock prescaler (ADCCLK): SYSCLKOUT/1

ADC clock frequency in MHz: 90

ADC overlap of sample and conversion (ADCNONOVERLAP): Allowed

Offset: 0

☐ External reference

VREFHI: 3.3

VREFLO: 0

INT pulse control: Late interrupt pulse

SOC high priority: All in round robin mode

XINT2SOC external pin: GPIO0

This Coder Target pane has a the same name as the Code Generation > Coder Target sub-pane that appears when the **System target file** parameter is `idelink_ert.tlc` or `idelink_grt.tlc`.

Note For R2013b Prerelease, this Coder Target pane can only be used to configure models for Texas Instruments C2000 processors. Before using the Coder Target pane, install the support package for C2000 processors. For more information, see "Support package for Texas Instruments C2000 processors" on page 18 and "Coder Target Pane: Texas Instruments C2000 Processors".

To use the Coder Target pane:

1 Open Configuration Parameter dialog box by entering **Ctrl+E**.

2 Select the Code Generation pane.

3 Set the **System target file** parameter to `ert.tlc`. Click **Apply**.

4 Set the **Target hardware** parameter to match your target hardware.

   The Configuration Parameters dialog box displays the Coder Target pane with parameters for the specified target hardware.

ZedBoard hardware support

You can automatically generate code from Simulink models and execute it on ZedBoard™ hardware. Specifically, you can execute the code in the Linux® environment on the ZedBoard's ARM Cortex-A9 processor.

To use this feature, install the corresponding support package:

1 In a MATLAB Command Window, enter `supportPackageInstaller`.

2 Use Support Package Installer to install *Embedded Coder Support Package for Xilinx® Zynq®-7000 Platform*.

This feature includes the Embedded Coder Support Package for Xilinx Zynq-7000 Platform block library, which contains:

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

For more information, see the "Support Package for Xilinx Zynq-7000 Platform" topic.

Note For more information about using HDL Coder™ software with the FPGA on the Avnet® ZedBoard hardware, see " IP core integration into Xilinx EDK project for ZC702 and ZedBoard"

Simplified multi-instance code interface and dynamic memory allocation for ERT targets

Compatibility Considerations: Yes

Embedded Coder now provides a simplified multi-instance code interface, with a dynamic memory allocation option, for ERT-based models. The new capabilities support easier integration of multi-instance code into applications. The new interface to generated model code features:

- Use of a single model entry-point function argument for instance data such as signals, states, parameters, and optionally root-level input and output.
- Configurable argument list for model root-level input and output.
- Option to generate a function that dynamically allocates memory for model instance data.

For more information, see model option **Generate reusable code**, “Entry-Point Functions and Scheduling”, and “Generate Reentrant Code from a Top-Level Model”.

For an example of an ERT-based model configured to generate reusable, reentrant code, see the example model `rtwdemo_reusable`.

Compatibility Considerations

Beginning in R2013b, when you select **Generate reusable code** for an ERT-based model, model data structures, such as Block I/O, DWork, and Parameters, are packaged into the real-time model data structure. The real-time model data structure is passed in a single argument to the model entry-point functions `model_initialize`, `model_step`, and `model_terminate`.

In earlier releases, when you selected **Generate reusable code** for an ERT-based model, model data structures were passed in separately as arguments to `model_step`. The number of generated arguments varied, depending on the data requirements of the model.

If you have code that calls reusable code generated from ERT-based models, you should update the model entry-point function calls to use the new, simplified interface.

For example, consider model entry-point functions previously called as follows:

```
/* Step the model */
  rtwdemo_reusable_step(&rtP, &rtDWork, &rtU, &rtY);

/* Initialize model */
  rtwdemo_reusable_initialize();
```

In R2013b or later, the corresponding calls might be as follows:

```
/* Step the model */
rtwdemo_reusable_step(rtM, &rtU, &rtY);

/* Initialize model */
rtwdemo_reusable_initialize(rtM, &rtU, &rtY);
```

Beginning in R2013b, after selecting **Generate reusable code**, you also can select the model option **Generate function to allocate model data**, which generates a function to dynamically allocate memory (using `malloc`) for model data structures. If you do not select this option, the model instance data must be allocated either statically or dynamically by the calling code. For this case, an additional requirement beginning in R2013b is that pointers to the individual data structures (Block IO, DWork, and Parameters) must be set up in the top-level real-time model data structure.

Performance

Reusable custom storage class to reduce root I/O memory

In R2013b, if a pair of root-level model input and output signals uses the same storage class specification, code generation can reuse the root I/O signals in the generated code. The storage class specifications are the new custom storage class `Reusable(Custom)` or a custom storage class created from `Reusable(Custom)`. Reusing code for root input and output signals allows for further optimizations that reduce data copies, global variables, and ROM/RAM size. For more information, see “Signal Reuse for Root-Level Model Inputs and Outputs”.

Subsystem functions reused independently of output connection

Previously, code generation used different criteria to determine when to reuse code.

- Code generation used the connection status to help determine the number of subsystem functions to generate.
- Code generation reused subsystem functions with varied connection status only when the outputs were passed by structure reference.

Code generation can now reuse subsystem functions regardless of:

- The connection state of the outputs. You can specify multiple outputs as unconnected or terminated across subsystems.
- Whether the reusable system outputs are passed as `Structure reference` or `Individual arguments`.

Verification

SIL and PIL support fixed-point data types wider than 32 bits

Use software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations to verify generated code that contains fixed-point data types wider than 32 bits.

A number of host and target platforms support 64-bit native data types. On these platforms, implementing a fixed-point data type wider than 32 bits as a single word is more efficient than the multiword fixed-point approach. Previously, data types wider than 32 bits, including multiword fixed-point, were supported internally within a SIL or PIL component. However, the data types were not supported in the communication between the MATLAB and Simulink host and the SIL or PIL component on the target. Now, the software supports 33-bit to 64-bit single word, fixed-point data types in host-target communication.

Data types that SIL and PIL support include the following:

- 64-bit `long` and `long long`
- 64-bit execution profiling timer data type — Previously, the target returned only the 32 least significant bits to the MATLAB host, with the possibility of timer wrapping.
- `int64` and `uint64` — Used in MATLAB Coder SIL execution.

The following constraints apply:

- For 64-bit data type support, the data type must be representable as `long` or `long long` on the MATLAB host *and* the target. Otherwise, the software uses the multiword fixed-point approach, which SIL and PIL do not support.
- 32-bit Windows® does not support 64-bit `long` or `long long` data types. In this case, the software uses the multiword fixed-point approach which SIL and PIL do not support.
- The software does not support the 40-bit `long` data type of the TI's C6000™ target.

Through the **Configuration > Hardware Implementation** pane, you can enable support for the 64-bit `long long` data type. However, for data types with widths between 33 and 40 bits (inclusive), the software implements the data types using the 40-bit `long` data type which SIL and PIL do not support.

SIL and PIL protected model support

Software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation modes are now supported for protected models. You can run models that contain protected models in SIL and PIL simulation modes if the protected models support code generation. In previous releases, the only supported simulation modes were Normal and Accelerator.

Code execution profiling improvements

Standalone code generation with function profiling

You can generate executable code (**Ctrl+B**) for your model even if function profiling is enabled. The software produces the following warning message:

```
Warning: Code profiling instrumentation is not supported for standalone
builds (Ctrl+B). You can run the executable, but no profiling data will be
collected.
```

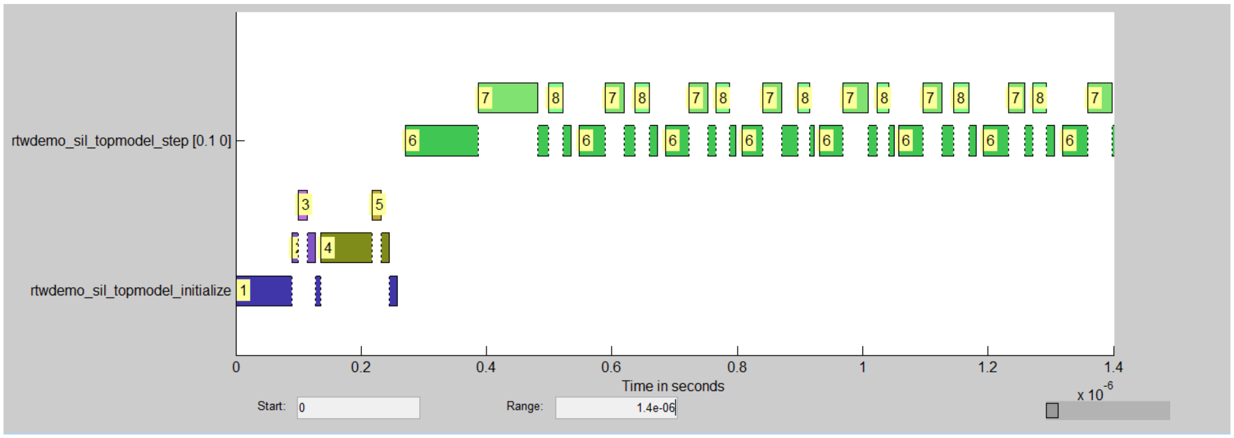
Previously, if function profiling was enabled for a SIL or PIL simulation, building your model produced an error message. For example:

```
Code profiling instrumentation within the generated code is not supported
for top model standalone builds (Ctrl+B). You cannot build the top model
rtwdemo_sil_modelblock in standalone mode because rtwdemo_sil_modelblock
has code profiling instrumentation enabled. You must either simulate
rtwdemo_sil_modelblock in SIL or PIL mode or disable code profiling
instrumentation for rtwdemo_sil_modelblock. To disable code profiling
instrumentation, clear the check box Simulation > Configuration Parameters
> Code Generation > Verification > Measure function execution times.
```

For information about obtaining execution time profiles for generated code, see “Code Execution Profiling”.

Display of code section invocations

You can display code section invocations over the execution timeline.



Start: 0
Range: 1.4e-06

For more information, see `timeline`.

SampleOffset and SamplePeriod removed

The `coder.profile.ExecutionTimeSection` `SampleOffset` and `SamplePeriod` methods have been removed.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2013b Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2013b&keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2013b&product=EC`

R2013a

Version: 6.4

New Features: Yes

Bug Fixes: Yes

Code Generation from MATLAB Code

Improved code replacement traceability for MATLAB code generation

In the R2013a release, there is now improved code replacement traceability for standalone code generated using MATLAB Coder. This capability is not available for generated MEX functions. When you choose to include code replacements in the code generation report:

- The code generation report includes a link to the Code Replacements Report.
- Code replacement trace information is generated for viewing in the **Trace Information** tab of the Code Replacement Viewer.
- The code replacement report lists replacement functions and their associated MATLAB code.

You can use the code replacement report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the MATLAB code that triggered the replacement.

For more information, see Enable the Code Replacements Report and Viewing Code Replacements in the Generated Code.

Static code metrics report for MATLAB Coder

When you generate standalone C code with MATLAB Coder, the HTML code generation report now includes a static code metrics report. The static code metrics report is not available for generated MEX functions.

The static code metrics include the:

- Number of source code files.
- Number of lines of code.

- List of global variables.
- Functions in a call tree format.
- Estimated stack size required for a function.

You can use the information in the static code metrics report to:

- Find the number of files and lines of code in each file.
- Estimate the number of lines of code and stack usage per function.
- Compare how many files, functions, variables, and lines of code are generated every time you change the MATLAB algorithm.
- Determine a target platform and allocation of RAM to the stack, based on the size of global variables plus the estimated stack size.
- Determine possible performance slow points, such as the largest global variables or the most costly call path in terms of stack usage.
- View the cyclomatic complexity of a function, which counts the number of linearly independent paths through a function.
- View the function call tree.
- Determine the longest call path to estimate the worst-case execution timing.
- View how target functions, provided by the selected code replacement library, are used in the generated code.

For more information, see Generate a Static Code Metrics Report for MATLAB Code and Static Code Metrics.

Model Architecture and Design

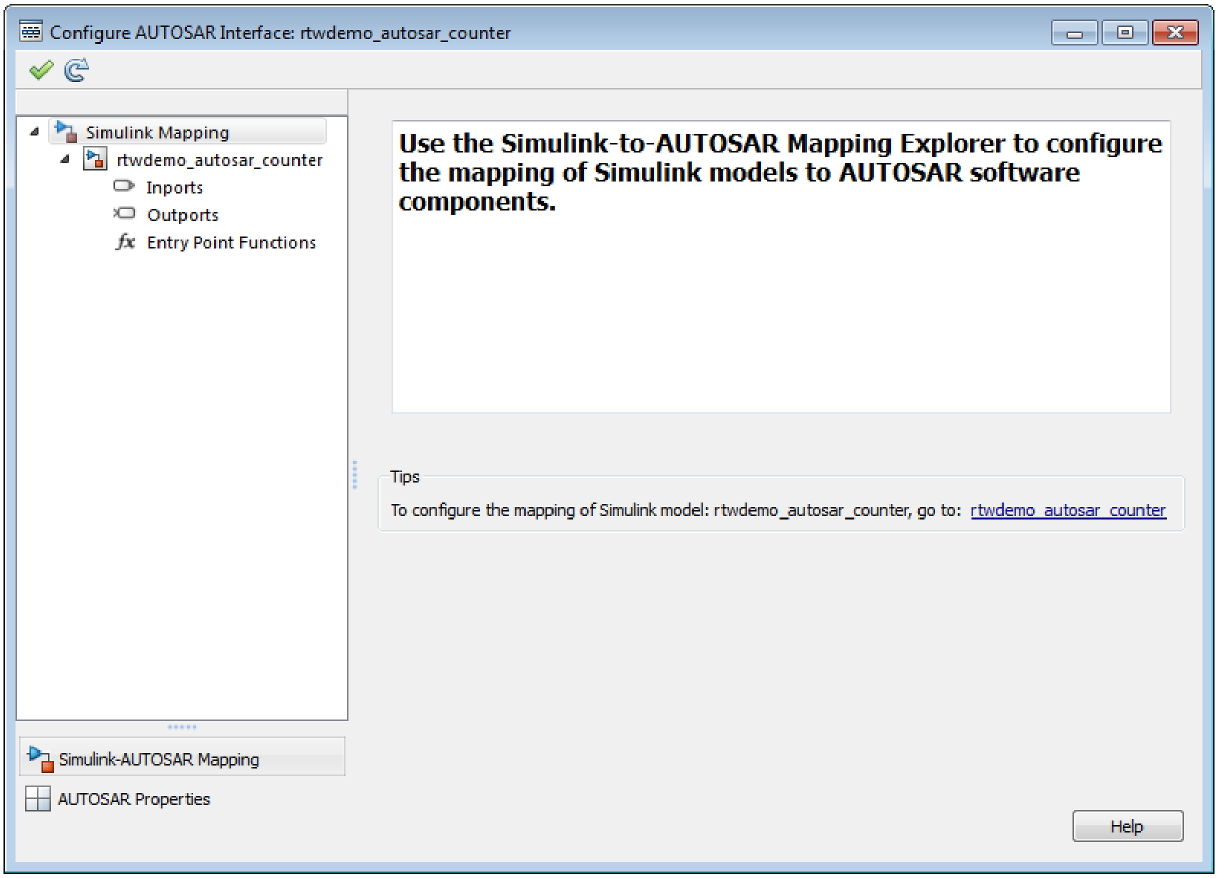
AUTOSAR user interface and round trip ARXML file import and export improvements

Improved graphical user interfaces for AUTOSAR configuration

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

- The Configure AUTOSAR Interface dialog box now provides separate **Simulink-AUTOSAR Mapping** and **AUTOSAR Properties** Explorers, which clearly distinguish mapping and editing activities.
- In both the Mapping and Properties Explorers:
  - Parameters that previously required text entry now offer selectable values or attributes.
  - Displays are more scalable (accommodating more elements) with fewer refresh issues.
  - Graphical layout reflects logical relationships between entities.
  - Filtering enhances element selection and modification.
- The Properties Explorer provides intuitive double-click and add/remove operations for configuring AUTOSAR ports, interfaces, data elements, runnables, and events.
- New check and synchronization icons provide single-click access to AUTOSAR configuration validation and Simulink model synchronization.
- A new AUTOSAR Component Builder dialog box allows you to interactively create a customized AUTOSAR component.

To explore the Configure AUTOSAR Interface dialog box, open a model that is already configured for AUTOSAR (such as the example model `rtwdemo_autosar_counter`). Select **Code > C/C++ Code > Configure Model as AUTOSAR Component** to open the dialog box. From there, you can select either the **Simulink-AUTOSAR Mapping** Explorer or the **AUTOSAR Properties** Explorer.

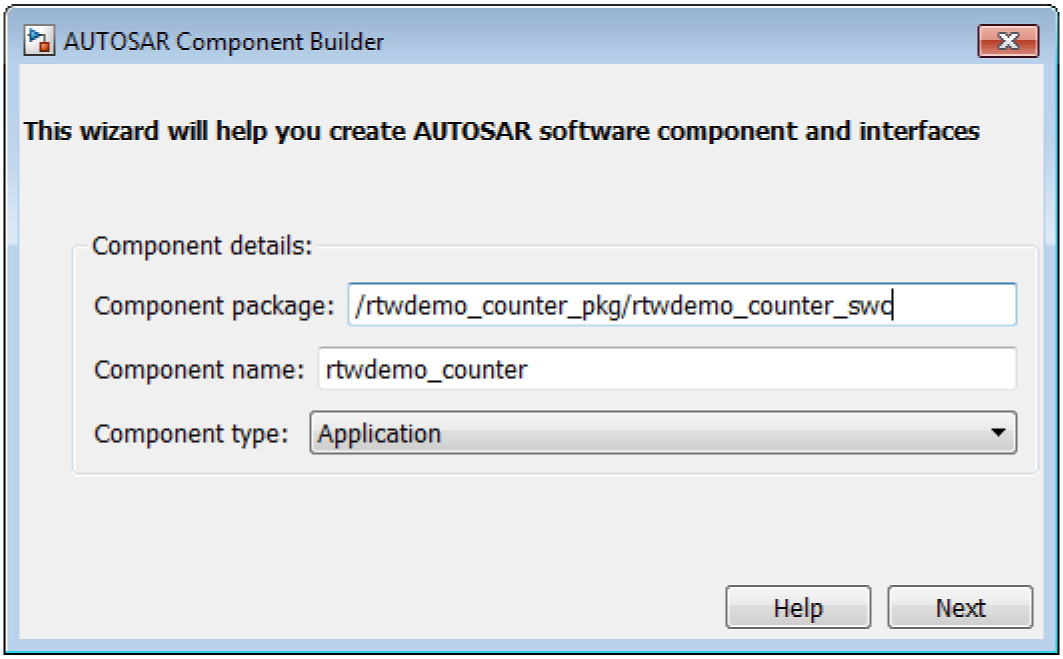


Use the Simulink-to-AUTOSAR Mapping Explorer to configure the mapping of Simulink models to AUTOSAR software components.

Tips

To configure the mapping of Simulink model: rtwdemo_autosar_counter, go to: rtwdemo_autosar_counter

To explore the AUTOSAR Component Builder dialog box, open a model that is not already configured for AUTOSAR (such as the example model `rtwdemo_counter`). Select the AUTOSAR target (`autosar.tlc`) for the model, and then select **Code > C/C++ Code > Configure Model as AUTOSAR Component**. This action opens a dialog box that allows you to select between creating a default AUTOSAR component or interactively creating an AUTOSAR component. To open the AUTOSAR Component Builder dialog box, click **Create Component Interactively**.



AUTOSAR Component Builder

This wizard will help you create AUTOSAR software component and interfaces

Component details:

Component package: /rtwdemo_counter_pkg/rtwdemo_counter_swc

Component name: rtwdemo_counter

Component type: Application

Round-trip preservation of AUTOSAR elements and UUIDs

To help support the round trip of AUTOSAR elements between an AUTOSAR authoring tool (AAT) and the Simulink model-based design environment, Embedded Coder now preserves AUTOSAR elements and their UUIDs across `arxml` import and export, as follows:

- When `arxml` files created by an AAT are imported into a Simulink model, AUTOSAR element information is preserved, including UUIDs (for Identifiables), properties, and reference packages.
- When `arxml` files are exported from a Simulink model, the elements are generated back into `arxml` with their UUIDs and other information preserved.

As a result, the `arxml` files exported from Simulink can more easily be merged back into the AAT environment. Existing elements retain their UUIDs, while new elements created in Simulink get new UUIDs.

Code generation for variable-size scalar signals

Previously, a model that used a variable-size scalar signal (width equals 1) would cause an error during a model update. This limitation has been removed and the model now simulates and generates code for a variable-size scalar signal.

Data, Function, and File Definition

Shortened system-generated identifier names

In R2013a, you have the option to shorten the system-generated identifier names to allow more space for user names. This option also provides a more predictable and consistent naming system that uses camel case, no underscores or plurals, and consistent abbreviations for both a type and a variable.

To use the new names, open the Configuration Parameters dialog box, and on the **Code Generation > Symbols** pane, set the **System-generated identifiers** parameter to `Shortened`. When you open a new model in R2013a, the default setting for **System-generated identifiers** is set to `Shortened`. When you open an existing model in R2013a, **System-generated identifiers** is set as `Classic`. With this setting, the system-generated identifiers use the names from previous releases.

For more information, see System-generated identifiers and Customize Generated Identifier Naming Rules.

Improved data initialization with custom storage classes

Previously, Embedded Coder generated initialization code for these two cases, even though the **DataInitialization** parameter was set to `None` or `Static`.

1. Initial output of an Enabled Subsystem configured to reset when it is enabled.

2. Fixed-point data with bias, which cannot have zero ground value

Now, Embedded Coder will not generate dynamic initialization code for data that uses a custom storage class whose **DataInitialization** parameter is set to `None` or `Static`.

Default specification for global types

Previously, on the Configuration Parameters **Symbol** pane, the default for **Global types** was `$N$R$M`. In R2013a, for new models, the default for **Global types** is `$N$R$M_T`. For existing models opened in R2013a, **Global types** does not change.

Subsystem block parameter Function packaging option renamed

In the Subsystem block parameter dialog box, on the **Code Generation** tab, the Function packaging option `Function` is renamed to `Nonreusable function`.

Code Generation

Model Advisor checks for code generation

The Model Advisor **By Product** folder contains the following checks to replace **Identify questionable blocks within the specified system**:

- Check for blocks not supported by code generation
- Check for blocks not recommended for C/C++ production code deployment

To display the **By Product** folder, in the Model Advisor window select **Settings > Preferences**. In the Model Advisor Preferences dialog box, select **Show By Product Folder**.

Deployment

Concurrent execution API to target embedded multicore platforms

Semaphore and mutex code replacement for multicore target environments

Embedded Coder software now provides Simulink code replacement support for the following semaphore and mutex operations.

Mutex Destroy
Mutex Init
Mutex Lock
Mutex Unlock
Semaphore Destroy
Semaphore Init
Semaphore Post
Semaphore Wait

Semaphore and mutex code replacement is supported for:

- Simulink code generation for data transfer between tasks
- Code generation targets

Semaphore and mutex code replacement is not supported for:

- Stateflow charts, MATLAB Function blocks, and MATLAB functions
- Simulation targets

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

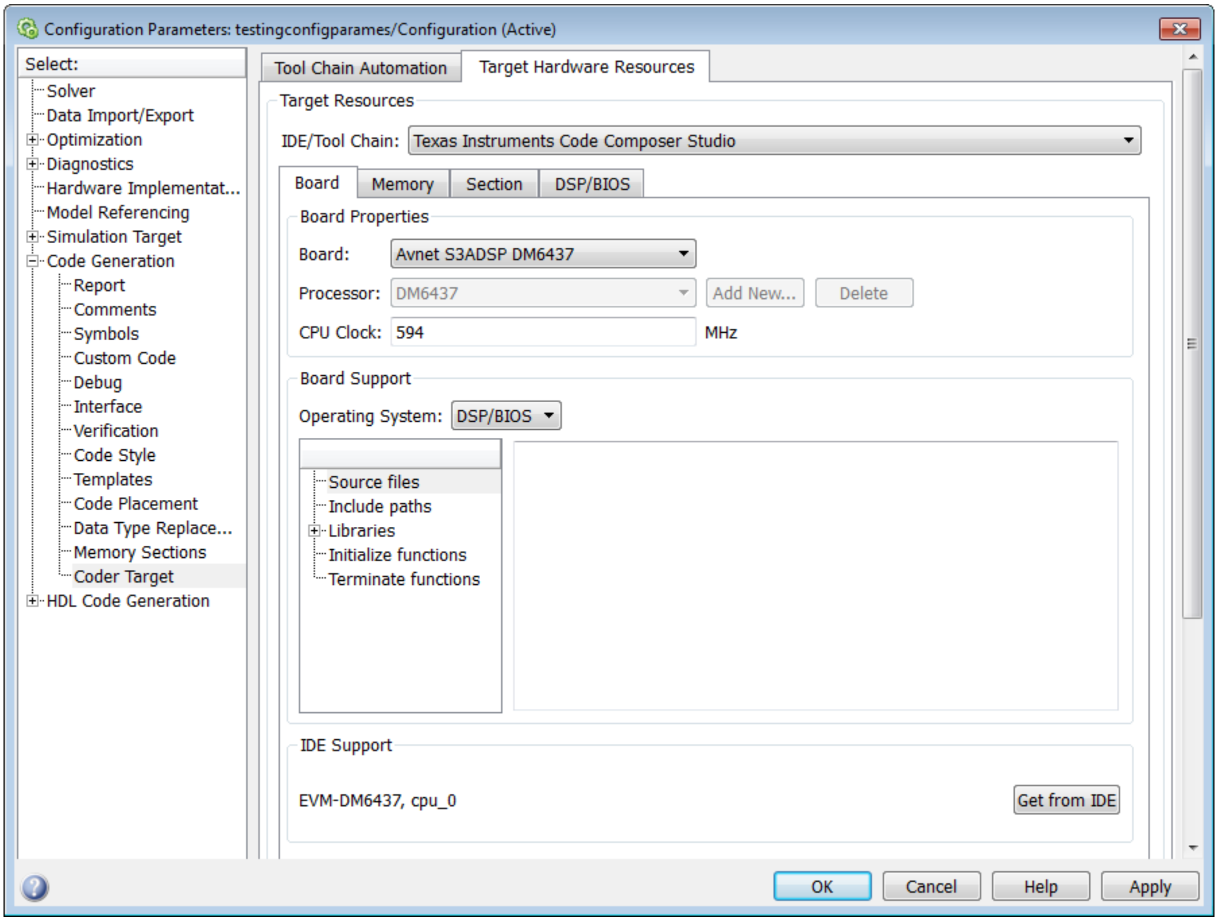
Hardware timer function replacement

You can create a hardware-specific timer object for SIL and PIL simulations with your hardware target. See *Specification of hardware timer through the*

Code Replacement Tool in “Code execution profiling improvements” on page 50.

Hardware configuration relocation from Target Preferences block to Configuration Parameters dialog box

The contents of the Target Preferences block have been relocated to the new **Target Hardware Resources** tab on the Coder Target pane in the Configuration Parameters dialog box.


Configuration Parameters: testingconfigparames/Configuration (Active)

IDE/Tool Chain: Texas Instruments Code Composer Studio

Board: Avnet S3ADSP DM6437

Processor: DM6437

CPU Clock: 594 MHz

Operating System: DSP/BIOS

EVM-DM6437, cpu_0

The Target Preferences block has been removed from the Embedded Targets block library.

If you open a model that contains a Target Preferences block, a warning instructs you that the block is optional and can be removed from your model.

Opening the Target Preferences block automatically displays the **Target Hardware Resources** tab.

For instructions on how to use **Target Hardware Resources** to build and run a model on target hardware, see Model Setup.

For information about specific parameters and settings, see Code Generation: Coder Target Pane.

Downloadable support and blocks for Analog Devices DSPs

Compatibility Considerations: Yes

If you have an Embedded Coder license, you can install support for Analog Devices™ VisualDSP++® IDE and DSPs as described in Install Support for Analog Devices DSPs. Support for Analog Devices VisualDSP++ IDE and DSPs includes the same capabilities that were previously available.

Use the "Embedded Coder Support Package for Analog Devices DSPs" block library to manage peripherals, scheduling, and memory on Blackfin®, SHARC®, and TigerSHARC® DSPs.

To get these capabilities, in a MATLAB Command Window, enter `supportPackageInstaller`. Then, use Support Package Installer to install the support package for Analog Devices DSPs. For more information, see the Working with Analog Devices VisualDSP++ IDE topic.

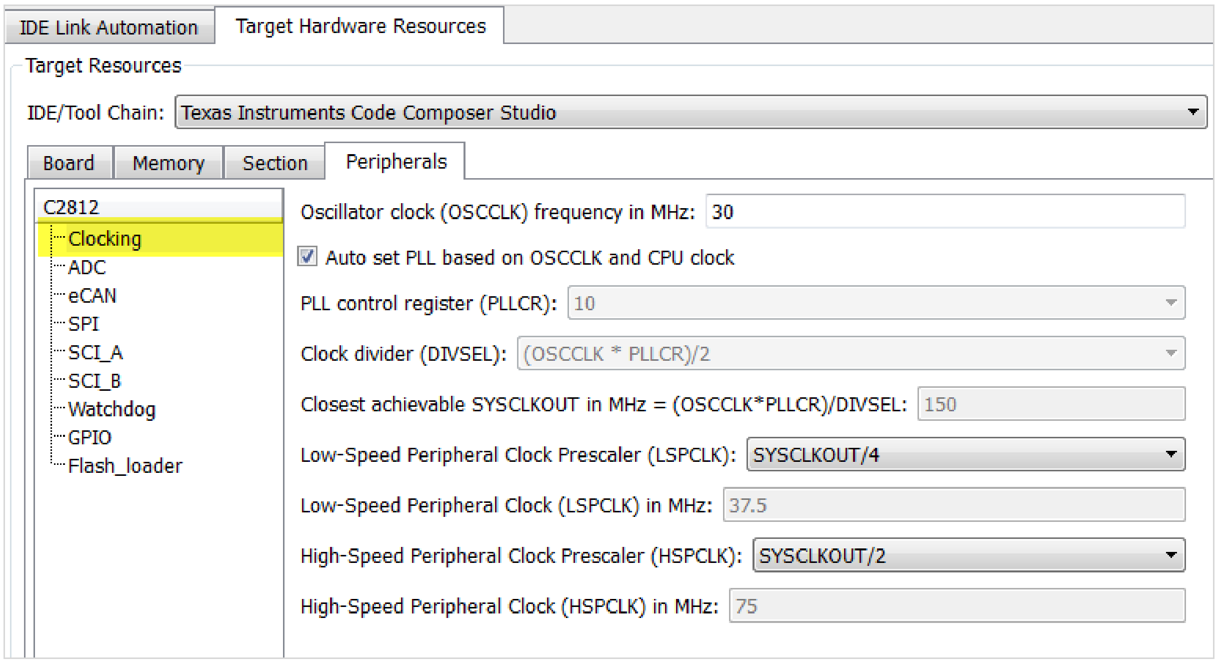
After installing the support package, you can open the block library. In the MATLAB Command Window, enter `adivdsplib`. The "Embedded Coder Support Package for Analog Devices DSPs" block library is also available in the Simulink Library Browser.

Compatibility Considerations

Previously, installing Embedded Coder software automatically installed support and blocks for Analog Devices DSPs. Effective this release, you must use Support Package Installer to install support before using Embedded Coder with Analog Devices DSPs.

Texas Instruments C2000 Clocking Options

In the Configuration Parameters dialog box, on the **Peripherals** tab, the new **Clocking** options help you to configure different timers that you use in the processor peripherals.



IDE/Tool Chain: Texas Instruments Code Composer Studio

Oscillator clock (OSCCLK) frequency in MHz: 30

[✓] Auto set PLL based on OSCCLK and CPU clock

PLL control register (PLLCR): 10

Clock divider (DIVSEL): (OSCCLK * PLLCR)/2

Closest achievable SYSCLKOUT in MHz = (OSCCLK*PLLCR)/DIVSEL: 150

Low-Speed Peripheral Clock Prescaler (LSPCLK): SYSCLKOUT/4

Low-Speed Peripheral Clock (LSPCLK) in MHz: 37.5

High-Speed Peripheral Clock Prescaler (HSPCLK): SYSCLKOUT/2

High-Speed Peripheral Clock (HSPCLK) in MHz: 75

- The high-speed and low-speed clock settings help you to configure the baud rates for peripherals, such as SCI and SPI.
- You can specify the oscillator clock frequency used in the processor to set the system clock out parameter for the device. Based on the system clock out value, you can get feedback on the baud rate and the time settings.
- Automatic setting of the prescalers is done based on user-defined baud rate for peripherals, such as SCI and SPI.
- Based on the settings that you make in the Clocking peripheral, you can see the timing-related feedback for the peripherals, such as eCAN, I2C, ADC, and Watchdog.

- The parameter relationship is shown at the prompts in some of the peripherals. For example, in eCAN, at the baud rate parameter, you can see, CAN Module Clock/BRP/(TSEG1+TSEG2+1)) in bits/sec.

Support for Texas Instruments C2802x and Texas Instruments C2803x variants

You can now run models on the following variants of TI C2802x and TI C2803x processors:

- F28030
- F28031
- F28032
- F28033_cpu
- F28034
- F280200
- F28020
- F28021
- F28022
- F28026

You can use the following block libraries with these variants:

- C2802x (c2802xlib)
- C2803x (c2803xlib)

Downloadable support and blocks for Xilinx Zynq-7000 platform

Use the Embedded Coder Support Package for Xilinx Zynq-7000 Platform to automatically build (makefile-based), download, and run an executable on the *Xilinx Zynq-7000 SoC ZC702 Evaluation Kit*. The executable runs in the Linux environment on the ARM Cortex-A9 processor on the *ZC702 Evaluation Kit*.

Use blocks from the Embedded Coder Support Package for Xilinx Zynq-7000 Platform block library:

- The UDP Receive and UDP Send blocks enable communication with networked devices using an Ethernet port.
- The Linux Task block spawns a task function as separate Linux thread.

To download and install this feature, click **Add-Ons > Get Hardware Support Packages** on the MATLAB toolstrip. Then, use Support Package Installer to install the Embedded Coder Support Package for Xilinx Zynq-7000 Platform. For more information, see the Working with the Xilinx Zynq-7000 Platform topic.

Support ending for Eclipse IDE in a future release

Support for the Eclipse™ IDE will end in a future release of the Embedded Coder and Simulink Coder products.

Support ending for remoteBuild method in a future release

Compatibility Considerations: Yes

Support for the `remoteBuild` method will end in a future release of the Embedded Coder products.

Compatibility Considerations

Use Support Package Installer to install the support package for BeagleBoard hardware, as described in Install Support for BeagleBoard Hardware. Then, use the Simulink capability called “Run on Target Hardware” instead of `remoteBuild` to build and run models on BeagleBoard hardware.

For more information about using Run on Target Hardware with BeagleBoard, see the BeagleBoard topic.

Performance

Optimized function arguments for nonreusable subsystems

For nonreusable subsystems, you can specify the function interface in the generated code to use arguments. This specification reduces global RAM. It might reduce code size and improve execution speed, and allow the code generator to apply additional optimizations.

To optimize the function interface for a subsystem, in the Subsystem Block Parameter dialog box, on the **Code Generation** tab, set the **Function packaging** parameter to `Nonreusable function` (previously, named `Function`). The **Function packaging** parameter enables the **Function interface** parameter. Set the **Function interface** parameter to `Allow arguments`.

For more information, see Function interface and Reduce Global Variables in Nonreusable Subsystem Functions.

Reduced data copies for tunable parameter expressions

Previously, in the generated code, tunable parameter expressions were copied to a temporary variable. In R2013a, the generated code removes this temporary variable. The removal of this unnecessary data copy improves execution speed, reduces code size and global RAM, and allows for additional code optimizations.

For example, for a tunable parameter, `b`, used in a Constant block, the code was:

```
/*Constant: '<Root>/Constant'*/
for (i=0; i<9; i++){
   tunable_expr_copy_B.Constant[i] = Param.b[i];
}
/*End of Constant: '<Root>/Constant'*/
```

```
/*S-Function(MySFun2D): '<Root>/S-Function Builder'*/
MySFun2D_Outputs_wrapper(tunable_expr_copy_B.Constant);
```

Now, the generated code is:

```
/*S-Function(MySFun2D): '<Root>/S-Function Builder'*/
MySFun2D_Outputs_wrapper(Param.b);
```

Removal of unused global variables

In R2013a, unused global variables generated from a For Each subsystem and bitfields are removed. This code generation enhancement reduces global RAM.

Verification

Debugging during SIL simulations

If you notice differences between the results of a Normal mode simulation and a SIL mode simulation, you can select the **Configuration Parameters > Verification > Enable source-level debugging for SIL** check box and rerun the SIL simulation. Then, from the Microsoft® Visual Studio® IDE, you can insert break points in the generated source code and step through the code during the SIL simulation. Observing code behavior in this way can help you to understand the differences in results. For example, when you are trying to integrate legacy code with generated code and the integration does not run as expected.

For more information, see Debugging During SIL Simulations.

Simulation of multiple SIL Model blocks in a top model

If you have a top model containing Model blocks, you can simulate the model with multiple Model blocks in SIL mode. Previously, you could not simulate the top model with more than one Model block in SIL mode. To verify the different Model blocks, you had to run multiple simulations. Before each simulation, you had to specify the SIL mode for one Model block. The removal of this limitation reduces verification time.

If you specify code coverage or code execution profiling, the software does not support this feature.

API for testing `rtiostream` communications

To run PIL or External mode simulations with custom hardware, you write your own `rtiostream` implementations.

R2013a provides a test suite to debug and prove the behavior of custom `rtiostream` interface implementations.

This new API has the following advantages:

- Reduces time for integrating custom hardware that does not have built-in `rtiostream` support.
- Reduces time for testing custom `rtiostream` drivers.
- Helps analyze the performance of custom `rtiostream` drivers.

This test suite has two parts. One part of the test suite runs on the target.

To launch this part, compile and link the following files, which are in *`matlabroot`*`/toolbox/coder/rtiostream/src/rtiostreamtest`.

- `rtiostreamtest.c`
- `rtiostreamtest.h`
- `rtiostream.h`
- `rtiostream` implementation under investigation (e.g., `rtiostream_tcpip.c`)
- `main.c`

To run the second part of the test suite, invoke `rtiostreamtest`. The syntax is as follows:

```
function rtiostreamtest(connection,param1,param2)
```

- `connection` is a string indicating the communication method. It can have values `'tcp'` or `'serial'`.
- `param1` and `param2` have different values depending on the value of `connection`.
  - If `connection` is `'tcp'`: `param1,param2` are hostname and port, respectively.
  - If `connection` is `'serial'`: `param1,param2` are COM port and baud rate, respectively.

For example, you can run the second part of the test suite as follows:

```
function rtiostreamtest('tcp','localhost','2345')
```

SIL and PIL support for targets with multicore processors

R2013a allows you to run SIL and PIL simulations of models that are configured for targets with multicore processors:

- You can run SIL and PIL simulations of **single-rate** component models in a concurrent execution model hierarchy, without modifying models or regenerating code.
- Previously, the configuration parameters, `TargetOS` and `ConcurrentTasks`, had to be the same across a model hierarchy. This restriction has been removed.

Additional code annotation for justifying Polyspace checks

New Polyspace® code annotations have been added to justify occurrences of << and + inside fixed-point multiplication helper functions.

For more information, see Code Annotation for Justifying Polyspace Checks.

Code execution profiling improvements

Comprehensive measurement and reporting of function execution times

R2013a provides comprehensive measurement and reporting of function execution times:

- The software measures execution times for initialization, shared utility and math library functions.
- The software inserts instrumentation probes around a function call site so that the measured time includes the time taken to call the function. Previously, the software inserted instrumentation probes inside the function. As a result, the measured time represented the execution time for only the function body.

- You can specify the time unit and numeric format for the time measurements in the code execution profiling report. Previously, the software reported execution times only in clock ticks. For information about the new default specifications for time unit and numeric format, see report.
- The code execution profiling report contains hyperlinks to function call sites in the SIL/PIL test harness. Previously, the report provided hyperlinks to only source code files generated from the model.

For more information, see Code Execution Profiling.

Viewing and comparing execution time plots with the Simulation Data Inspector

You can use the Simulation Data Inspector to view and compare plots of function execution times. If you select `All measurement and analysis data` from the **Configuration Parameters > Code Generation > Verification > Save options** drop-down list, the software automatically imports SIL simulation results into the Simulation Data Inspector. This feature allows you to plot execution times and manage and compare plots from various simulations.

For more information, see Configure Code Execution Profiling and View and Compare Code Execution Times.

Specification of hardware timer through the Code Replacement Tool

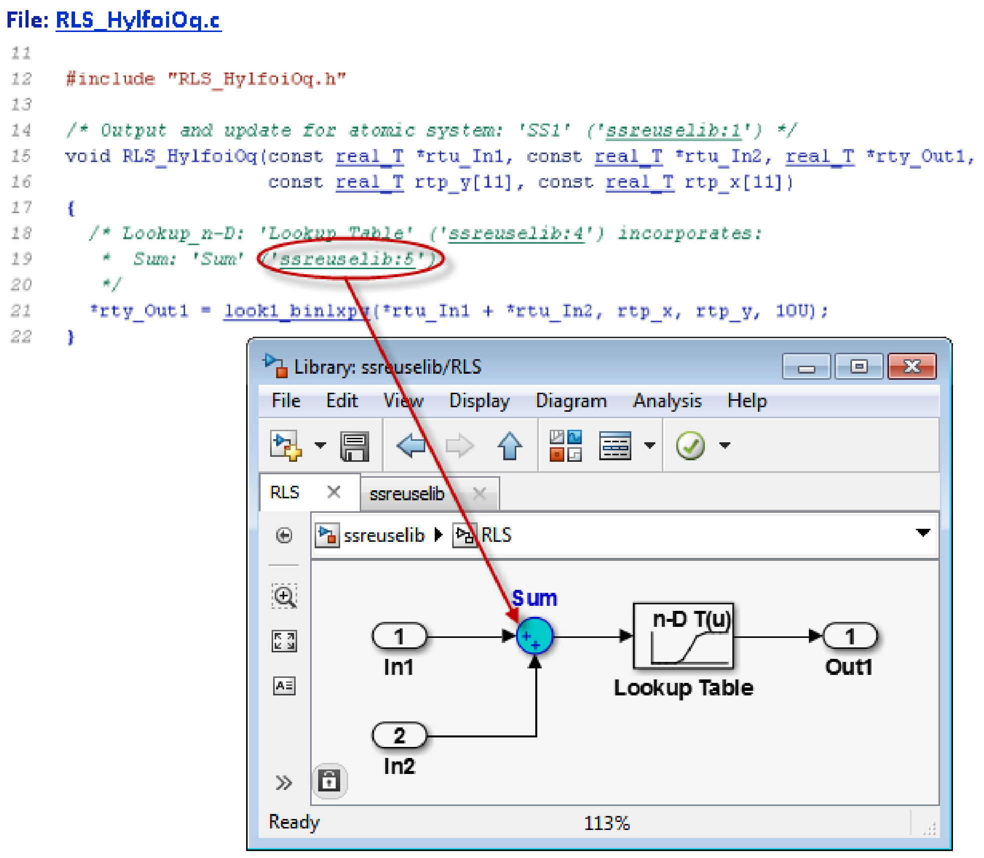
In SIL and PIL simulations, if your hardware target does not have built-in timer support, you must create a timer object that provides details of the hardware-specific timer and associated source files. In R2013a, you can specify this hardware-specific timer using either the graphical user interface of the Code Replacement Tool or the corresponding command line API. The software stores the timer information as a Code Replacement Library (CRL) table.

Previously, you could specify the timer using the MATLAB function `coder.profile.Timer`. However, support for this function will cease in a future release.

For more information, see Specify Hardware Timer.

Code-to-model traceability links for reusable subsystems in libraries

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



File: RLS_HylfoiOq.c

```

#include "RLS_HylfoiOq.h"

/* Output and update for atomic system: 'SS1' ('ssreuselib:1') */
void RLS_HylfoiOq(const real_T *rtu_In1, const real_T *rtu_In2, real_T *rty_Out1,
                  const real_T rtp_y[11], const real_T rtp_x[11])
{
  /* Lookup_n-D: 'Lookup Table' ('ssreuselib:4') incorporates:
   *  Sum: 'Sum' ('ssreuselib:5')
   */
  *rty_Out1 = look1_binlxpw(*rtu_In1 + *rtu_In2, rtp_x, rtp_y, 10U);
}
```

To include traceability links in the generated code comments, see Traceability in Code Generation Report.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2013a Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2013a`
`&keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2013a&product=EC`

R2012b

Version: 6.3

New Features: Yes

Bug Fixes: Yes

Cyclomatic complexity measurement in static code metrics report

In R2012b, the static code metrics report includes a cyclomatic complexity measurement for each function. You can view the measurement in the **Complexity** column of the Function Information table. For more information, see Analyze Static Code Metrics.

Custom code substitution for MATLAB functions using code replacement libraries

The `coder.replace` function provides the ability to replace a specified MATLAB function with a code replacement library (CRL) function in the generated code. You can use `coder.replace` both in MATLAB code from which you want to generate C code using MATLAB Coder and in MATLAB code in a MATLAB Function block. For more information, see `coder.replace`, Replace MATLAB Function with Custom Code, and Replace MATLAB Function Block Code with Custom Code.

In addition, you can use the code replacement tool to create and register code replacement tables. These tables provide the basis for replacing default math functions and operators in your generated code with target-specific code. The ability to control function and operator replacements potentially allows you to optimize target speed and memory and better integrate generated code with external and legacy code.

Access the code replacement tool using one of these methods:

- At the MATLAB command line, enter:

  ```
  crtool
  ```

- On the MATLAB Coder **Project Settings** dialog box **Hardware** tab, click the **Custom** link.

For more information, see Create Code Replacement Table for a Sample MATLAB Coder Project.

SIL and PIL support for signal logging, encapsulated C++, and AUTOSAR calibration parameters

Beginning in R2012b, Embedded Coder software supports using Simulink signal logging, encapsulated C++ code, and AUTOSAR calibration parameters in SIL and PIL mode simulations.

Signal logging for SIL and PIL simulations

In R2012b, Simulink signal logging is extended to the SIL and PIL simulation modes. This allows you to:

- Collect signal logging outputs (e.g., `logsout`) during SIL and PIL simulations.
- Log the internal signals and the root-level outputs of a SIL or PIL component.
- Manage the SIL and PIL signal logging settings using the Simulink Signal Logging Selector.
- More easily compare logged signals between normal, SIL, and PIL simulations, for example, using Simulation Data Inspector.

Signal logging is supported with the following forms of SIL and PIL simulation:

- Top-model SIL or PIL
- Model block (referenced model) SIL or PIL

SIL or PIL signal logging requires the following model configuration settings:

- On the **Data Import/Export** pane of the Configuration Parameters dialog box, set **Signal logging format** to `Dataset`.
- On the **Code Generation > Interface** pane of the Configuration Parameters dialog box, set **Interface** to `C API`.

Use SIL and PIL simulations to verify encapsulated C++ code

Previously, you could use SIL and PIL simulations to verify code generated with the model configuration **Language** setting `C` or `C++`. Beginning with R2012b, you can also use the **Language** setting `C++ (Encapsulated)`.

Encapsulated C++ code is supported with the following forms of SIL and PIL simulation:

- SIL or PIL block
- Top-model SIL or PIL
- Model block (referenced model) SIL or PIL

Improved SIL and PIL verification for AUTOSAR-compliant code

The following forms of SIL and PIL simulation support AUTOSAR calibration parameters in generated code:

- SIL or PIL block
- Top-model SIL or PIL

You can use the calibration parameter custom storage classes `CalPrm` and `InternalCalPrm` to reference data.

AUTOSAR 4.0 nonscalar data support

R2012b extends Embedded Coder support for using nonscalar data in models from which AUTOSAR 4.0 compatible code is generated. Previously, you could use nonscalar data associated with port elements, calibration parameters, and per-instance memory. Beginning in R2012b, you also can use nonscalar interrunnable variables (IRVs) in models configured for AUTOSAR.

For information about other AUTOSAR-related enhancements and changes, see “AUTOSAR software component import and export enhancements” on page 64.

Code annotation for justifying Polyspace checks

You can apply Polyspace verification to generated code using the Polyspace Model Link™ SL product. The software detects run-time errors in the generated code. It also helps you to locate and fix model faults.

Because of the way Embedded Coder implements certain operations, Polyspace might indicate potential overflows for operators or operations that are actually legitimate.

Previously, you manually justified the associated orange checks in the Polyspace verification environment.

Now, if you select the new check box, **Configuration Parameters > Code Generation > Comments > Auto generate comments > Operator annotations**, the Embedded Coder software annotates the generated code with comments for Polyspace. When you run a Polyspace verification, the Polyspace software uses the comments to justify overflows associated with legitimate operations and assigns the `Not a Defect` classification to the corresponding checks.

For more information, see Code Annotation for Justifying Polyspace Checks.

Texas Instruments Code Composer Studio IDE 5.1 support

This release adds support for version 5.1 of the Texas Instruments Code Composer Studio™ IDE (CCS) to existing support for CCS versions 3.3 and 4.1.

Support for CCS version 5.1 includes the following capabilities:

- Automatic creation of makefile projects
- Support for DSP/BIOS™ version 5.41.xx
- Support for C6000™ Compiler version 7.3.x

For more information, see Working with Texas Instruments Code Composer Studio IDE.

External mode support for ERT targets with static main

Previously, Embedded Coder software supported External mode for ERT targets only if the associated main program was automatically generated by the model build process. Beginning in R2012b, the software also supports External mode for ERT targets with a static main program. Specifically, the static main file `matlabroot/rtw/c/src/common/rt_main.c` has been enhanced to support External mode.

If you have authored a custom ERT-based target, you can support External mode with your custom main program by updating your main program, using the code in `rt_main.c` as an example.

Downloadable support for Green Hills MULTI

Compatibility Considerations: Yes

If you have an Embedded Coder license, you can install support for Green Hills® MULTI® IDE (MULTI) as described in Install Support for Green Hills MULTI IDE. Support for MULTI includes the same capabilities that were previously available.

After installing support for MULTI, you can use the “Target for Use with Green Hills MULTI IDE” block library, located in the Simulink Library Browser. You can open this block library by entering `idelinklib_ghsmulti` in the MATLAB Command Window.

The block library contains blocks for:

- Analog Devices Blackfin processors
  - Memory Allocate
  - Memory Copy
  - Blackfin Hardware Interrupt
  - Idle Task
- Freescale™ MPC55xx and MPC74xx processors
  - Memory Allocate

- Memory Copy
- Idle Task
- MPC5500 Interrupt
- MPC7400 Hardware Interrupt

Compatibility Considerations

Previously, Embedded Coder software included support for MULTI. Now, use Target Installer to install support before using Embedded Coder with MULTI.

Support for Texas Instruments C2806x processors

This release adds support for Texas Instruments C2806x processors to Embedded Coder.

This support adds the C2806x (c2806xlib) block library to the Simulink Library Browser. The C2806x block library includes the following blocks:

- C2802x/C2803x/C2806x ADC
- C2802x/C2803x/C2806x AnalogIO Input
- C2802x/C2803x/C2806x AnalogIO Output
- C28x CAN Calibration Protocol
- C2802x/C2803x/C2806x COMP
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x GPIO Digital Input
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x GPIO Digital Output
- C28x I2C Receive
- C28x I2C Transmit
- C28x SCI Receive
- C28x SCI Transmit
- C28x SPI Receive
- C28x SPI Transmit

- C28x Software Interrupt Trigger
- C28x Watchdog
- C28x eCAN Receive
- C28x eCAN Transmit
- C28x eCAP
- C280x/C2802x/C2803x/C2806x/C28x3x/c2834x ePWM
- C28x eQEP

For more information, see C2806x (c2806xlib).

Performance enhancement of Simulink data objects

In R2012b, Simulink can create and load subclasses of Simulink data classes more efficiently. To take advantage of this enhancement, use the `setupCoderInfo` method to configure the `CoderInfo` object of your class. The `setupCoderInfo` method is called once during object construction.

Consider the example of the `ECoderDemos.Parameter` class. Previously, this class was defined as follows. Notice how the `CoderInfo` object is configured in the class constructor.

```
classdef Parameter < Simulink.Parameter
% ECoderDemos.Parameter Class definition.

        methods
                function h = Parameter(optionalValue)
                % Use custom storage classes from this package
                useLocalCustomStorageClasses(h, 'ECoderDemos');

                % Set up object to use custom storage classes by default
                h.CoderInfo.StorageClass = 'Custom';

                % Initialize Value property
                        switch nargin
                                case 0,
                                        % No action
                        case 1,
                                h.Value = optionalValue;
                end
            end
        end % methods
end % classdef
```

In this release, the `ECoderDemos.Parameter` class is defined as follows. Notice the use of the `setupCoderInfo` method to configure the `CoderInfo` object. The rest of the constructor method is unchanged.

Note You can access this class definition at `matlabroot/toolbox/rtw/targets/ecoder/ecoderdemos/dataclasses-/+ECoderDemos/@Parameter/Parameter.m`.

```
classdef Parameter < Simulink.Parameter
% ECoderDemos.Parameter Class definition

        methods
                function setupCoderInfo(h)
                % Use custom storage classes from this package
                useLocalCustomStorageClasses(h, 'ECoderDemos');

                % Set up object to use custom storage classes by default
                h.CoderInfo.StorageClass = 'Custom';
                end

                function h = Parameter(optionalValue)
                % Initialize Value property
                        switch nargin
                                case 0,
                                % No action
                                case 1,
                                        h.Value = optionalValue;
                        end
                end
        end % methods
end % classdef
```

AUTOSAR software component import and export enhancements

R2012b adds AUTOSAR workflow improvements, including import validation and faster import and export of `arxml` files. See also “AUTOSAR 4.0 nonscalar data support” on page 58.

Import validation

Beginning in R2012b, the AUTOSAR software component importer validates the XML in the imported `arxml` files. If XML validation fails for a file, the importer displays errors. For example:

```
Error
The IsService attribute is undefined for interface /mtest_pkg/mtest_if/In1
in file hArxmlFileErrorMissingIsService_SR_3p2.arxml:48.
Specify the IsService attribute to be either true or false
```

In this example message, the file name is a hyperlink, and you can click the hyperlink to see the location of the error in the `arxml` file.

Faster import and export of arxml files

Beginning in R2012b, Embedded Coder software provides up to 20 times faster import and export of AUTOSAR software component descriptions.

Explicit access mode for AUTOSAR Sender and Receiver ports

Previously, the AUTOSAR software component importer did not support explicit data access modes for AUTOSAR component Sender and Receiver ports. It issued a warning for an explicit data access mode and set the port data access mode to implicit. Beginning in R2012b, the importer analyzes the AUTOSAR software component to determine whether the data access mode for a port is implicit or explicit. The importer honors an explicit access mode setting. However, if conflicting data access modes are detected, the importer issues a warning and sets the data access mode to implicit.

Import port-based calibration parameters

The AUTOSAR software component importer has been enhanced to import any port-based calibration parameters referenced in the AUTOSAR software component. For each imported parameter, the importer creates a data object in the MATLAB base workspace.

Highlight virtual blocks in model Web view of code generation report

In the model Web view of the code generation report, when tracing between the model and the code, if you click a virtual block and no code is highlighted in the generated code pane, the virtual block is highlighted yellow.

Code Execution Profiling Improvements

Compatibility Considerations: Yes

Updated Code Execution Profiling API

The existing code execution profiling APIs, rtw.pil.ExecutionProfile and rtw.pil.ExecutionProfileSection, have been replaced with coder.profile.ExecutionTime and coder.profile.ExecutionTimeSection respectively.

Compatibility Considerations

The old class names and methods forward to the corresponding new class names and methods. A warning is not issued. The old method names are hidden and no longer documented.

New Properties and Methods

The following new methods and properties have been added:

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

Functionality Being Removed or Changed

The following functionality is being removed or changed:

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.connectivity.Timer	Call is forwarded to coder.profile.Timer without warning message.	`coder.profile.Timer`	All methods are the same as rtw.connectivity.Timer.
rtw.pil.ExecutionProfile-.display	Call is forwarded to coder.profile.Execution-Time.display without warning message.	`display`	None
rtw.pil.ExecutionProfile-.report	Call is forwarded to coder.profile.Execution-Time.report without warning message.	`report`	None
rtw.pil.ExecutionProfile-.getSectionProfile rtw.pil.ExecutionProfile-.getNumSectionProfiles	Call is forwarded to coder.profile.Execution-Time.Sections without warning message.	`Sections`	Uses property syntax
rtw.pil.ExecutionProfile-.getTimerTicksPerSecond rtw.pil.ExecutionProfile-.setTimerTicksPerSecond	Calls are forwarded to property coder.profile.Execution-Time.TimerTicksPerSecond without warning message.	`TimerTicksPerSecond`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getMaxTicks	Call is forwarded to coder.profile.Execution-TimeSection.Maximum-ExecutionTimeInTicks without warning message.	`MaximumExecution-TimeInTicks`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getName	Call is forwarded to coder.profile.Execution-TimeSection.Name without warning message.	`Name`	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.pil.ExecutionProfile-Section.getNumCalls	Call is forwarded to coder.profile.Execution-TimeSection.NumCalls without warning message.	`NumCalls`	Uses property syntax
rtw.pil.ExecutionProfile-.getSectionNumber	Call is forwarded to coder.profile.Execution-Time.Number without warning message.	`Number`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getTicks	Call is forwarded to coder.profile.Execution-TimeSection.Execution-TimeInTicks without warning message.	`ExecutionTimeInTicks`	Uses property syntax
rtw.pil.ExecutionProfile-.getTimes	Call is forwarded to the legacy getTimes function without warning message.	Calculate execution time in seconds by the formula ExecutionTimeInSecs = ExecutionTimeInTicks / TimerTicksPerSecond.	No equivalent to getTimes in new interface.
rtw.pil.ExecutionProfile-Section.getTotalTicks	Call is forwarded to coder.profile.Execution-TimeSection.TotalExecution-TimeInTicks without warning message.	`TotalExecution-TimeInTicks`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getSampleOffset	Call is forwarded to coder.profile.Execution-TimeSection.SampleOffset without warning message.	`SampleOffset`	Uses property syntax

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
rtw.pil.ExecutionProfile-Section.getSamplePeriod	Call is forwarded to coder.profile.Execution-TimeSection.SamplePeriod without warning message.	`SamplePeriod`	Uses property syntax
rtw.pil.ExecutionProfile-Section.getTotalSelfTicks	Call is forwarded to coder.profile.Execution-TimeSection.TotalSelf-TimeInTicks without warning message.	`TotalSelfTimeInTicks`	Uses property syntax

Code Execution Profiling Supports Single Object Output

Code execution profiling during a SIL or PIL simulation honors the **Save simulation output as a single object** setting.

If the **Measure task execution time** check box is selected in the **Verification** pane and the **Save simulation output as a single object** check box is selected in the **Data Import/Export** pane, then the **Workspace variable** defined in the **Verification** pane is saved in the single output object instead of in the base workspace.

Incremental Compilation with Changes in Code Coverage Settings

If only code coverage settings have changed and the generated code is otherwise up to date, code is not regenerated. Instead, the existing up-to-date code is recompiled using the new code coverage settings.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2012b Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2012b`
`&keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2012b&product=EC`

R2012a

Version: 6.2

New Features: Yes

Bug Fixes: Yes

AUTOSAR Enhancements

AUTOSAR Release 4.0

R2012a supports AUTOSAR Release 4.0 (version 4.0.2), which includes:

- Import and export of AUTOSAR R4.0 XML files
- Generation of AUTOSAR R4.0 code
- Support for *application* and *implementation* data types and *base* types. For more information, see Data Type Support for Release 4.0.
- **Code replacement library** (CRL) support for over 300 routines from the following AUTOSAR libraries:
  - Floating-Point Math (`AUTOSAR_SWS_MFLLibrary`)
  - Fixed-Point Math (`AUTOSAR_SWS_MFXLibrary`)

Support for Schema 2.0 Removed

Support for AUTOSAR schema version 2.0 has been removed from R2012a. The software now supports the following schema versions:

- 4.0 (4.0.2)
- 3.2 (3.2.1)
- 3.1 (3.1.4) — Default
- 3.0 (3.0.2)
- 2.1 (XSD rev 0017)

Code Efficiency Enhancements

For Each Subsystem Loop Bound Passed by Value

The generated code of the For Each subsystem includes a loop bound that was previously passed by a pointer. In R2012a, the loop bound is passed by value which improves memory usage and execution speed.

For example, if you have a For Each subsystem with a **Function name**, `myFcnVectorized`, the generated code for the function prototype is:

```
void myFcnVectorized(int32_T NumIters,  ) {
   for (ForEach_itr = 0;
        ForEach_itr < NumIters;
        ForEach_itr++) { ...
```

The argument `NumIters` is passed by value, instead of by pointer. The function is called as follows:

```
myFcnVectorized(3, ...
```

For more information, see For Each Subsystem in the Simulink documentation.

Fully Inlined S-functions from Legacy Code Tool

The Legacy Code Tool now automatically generates fully inlined S-functions for legacy code. Previously, the generated code included an unnecessary data copy for the function-call input. In R2012a, these temporary variables are no longer generated. This enhancement reduces memory usage and improves execution speed, as well as enabling other optimizations and a consistent coding style.

For example, temporary variables, `tmp` and `tmp_0`, were used for the generated function-call input:

```
int32_T i;
real_T tmp[6];
real_T tmp_0[6];
for (i = 0; i < 6; i++) {
/* S-Function (rtwdemo_sfun_ndarray_add):'<S1>/rtwdemo_sfun_ndarray_add' */

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

Now, the generated code is:

```
int32_T i;

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

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

For more information, see Integrate External Code Using Legacy Code Tool.

Element-Wise Operations as Inputs to Intrinsic Functions

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

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

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

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

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

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

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

Enhancements to Custom Storage Classes in Simulink and mpt Packages

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

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

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

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

Code Generation Report Includes Simulink Web View

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

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

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

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

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

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

LDRA Testbed Code Coverage Annotations in Code Generation Report

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

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

Generated Identifiers Enhancements

Simplified Identifiers for Model Reference Code

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

Consistent Identifiers for Comparing Generated Code

To generate unique identifiers in the generated code, the code generation process inserts a mangling string in an identifier name. Previously, the mangling string was generated using the full block path name, which included the model name. In R2012a, the mangling string uses the Simulink Identifier (SID), which is unique within the model. This mangling string allows for consistent identifiers for similar or derived models, because the SID is persistent even if you change the name of the model. If you create another model using `Save As`, the SID is preserved for each block. For blocks in a subsystem, the SID is preserved whether you build the subsystem or build the model containing the subsystem.

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

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

Code Replacement Enhancements

Compatibility Considerations: Yes

R2012a provides the following enhancements to code replacement library support.

Target Function Libraries Renamed to Code Replacement Libraries

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

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

the demo are renamed to `rtwdemo_crl*`. For example, the model `rtwdemo_tfladdsub` is renamed to `rtwdemo_crladdsub`.

- The code replacement demo `coderdemo_tfl` is renamed to `coderdemo_crl`.
- The Target Function Library (TFL) Viewer is renamed to Code Replacement Viewer.

Code replacement related items that have *not* been renamed include code replacement classes, functions, and commands. Examples include the `RTW.TflCOperationEntry` class, the `setTflCFunctionEntryParameters` function, and the `RTW.viewTfl` command.

Enhanced Code Replacement Traceability

R2012a provides enhanced code replacement traceability, using the model option **Summarize which blocks triggered code replacements**, which is located on the **Code Generation > Report** pane of the Configuration Parameters dialog box. When you select **Summarize which blocks triggered code replacements**:

- Code generation includes a *code replacement report* in the HTML code generation report for your model.
- Code replacement trace information is generated for viewing in the **Trace Information** tab of the Code Replacement Viewer.

The code replacement report lists replacement functions and their associated blocks. You can use the report to:

- Determine which replacement functions were used in the generated code.
- Trace each replacement instance back to the Simulink block that triggered the replacement.

For more information, see Analyze Code Replacements in the Generated Code

The **Trace Information** tab of the Code Replacement Viewer lists **Hit Source Locations** and **Miss Source Locations**. The Viewer provides links to each source location (the source block for which code replacement was considered) and, for misses, lists a **Miss Reason**. For example, if a rounding mode setting did not match between a CRL entry and a block, the Viewer displays a reason similar to the following: “Mismatched rounding mode: actual ’RTW_ROUND_SIMPLEST’, expected ’RTW_ROUND_CEILING’.” After generating code for your model, you can open the Code Replacement Viewer for viewing hits and misses using the following commands:

```
>> crl=get_param('model','TargetFcnLibHandle')
>> RTW.viewTfl(crl)
```

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

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

Code Replacement Support for Simulink Matrix Division and Inversion Operators

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

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

For more information, see Map Nonscalar Operators to Target-Specific Implementations.

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

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

Integer Functions Now Return Real-World Values

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

Compatibility Considerations

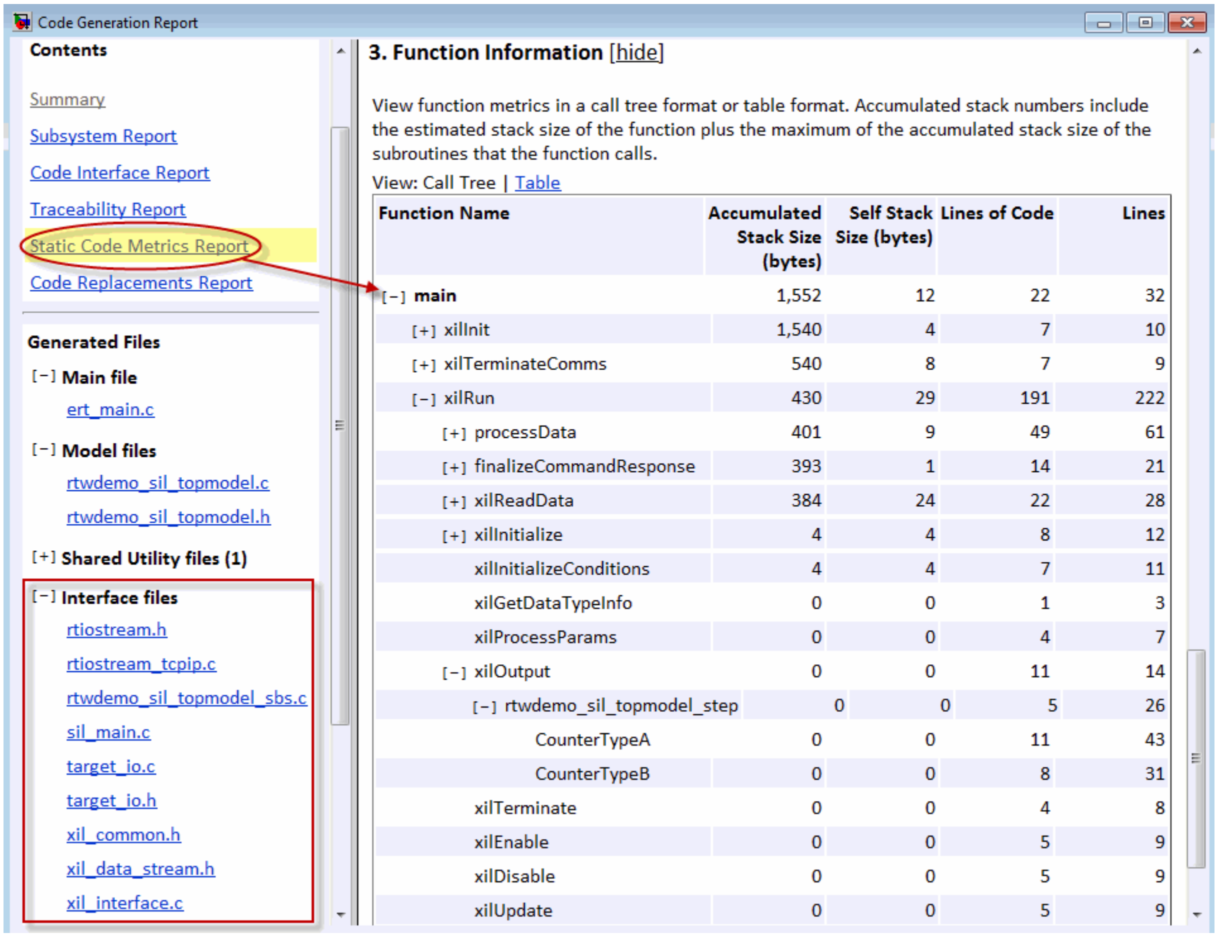
In code generation with MATLAB Coder or Simulink Coder, if you used a CRL to replace a cast in your replacement function, silent incorrect numerical results may occur. The numerical results will not change if the input fi object has binary-point scaling and zero fractional length. To optimize code generation,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

SIL and PIL Enhancements

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

SIL and PIL Test Harness Files in Code Generation Report

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



3. Function Information [hide]

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

View: Call Tree | Table

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

This feature helps you to:

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

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

PIL Support for Code Coverage with LDRA Testbed

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

- Top-Model PIL
- Model block PIL

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

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

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

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

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

This feature:

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

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

Enhanced Hardware Implementation Support

Host and Target Floating Point Data Type Sizes

The host and target floating point data type sizes must be the same. Previously, a mismatch would produce undefined behaviour resulting in a simulation failure. Now, the software generates an error with a clear message when the host and target data types are *not*:

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

For more information, seeHardware Implementation Support.

Word-Addressable Targets

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

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

The software uses the MATLAB host byte order when sending words through the `rtIOStream` API. For information about host byte ordering, see `computer` in the MATLAB Reference documentation.

Top-Model Output Limitations Removed

Previously, in a top-model SIL/PIL simulation, not all signal and output logging fields matched the fields produced by a Normal simulation. For example:

- With signal logging, the software would add the suffix `_wrapper` to the block path for signals in `logsout`.
- With output logging, if the save format was `Structure` or `Structure with time`, the software would add the suffix `_wrapper` to the block name for signals in `yout`.

These limitations are not present in R2012a, except if you do one of the following:

- Specify the signal logging format to be `ModelDataLogs`. In this case, `yout` will still contain references to the wrapper model. You should use the `Dataset` signal logging format. See `Simulink.SimulationData.Dataset` in the Simulink reference documentation.
- Run command line simulations using the `sim` command but without specifying the single-output format. See Using the sim Command in the Simulink documentation.

Model Block SIL/PIL Support for Absolute Time

Previously, you could not run a Model block in the SIL or PIL mode if the Model block contained Simulink blocks that depended on absolute time. Now, Model block SIL/PIL supports absolute time except for the following case: the Model block contains Simulink blocks that require absolute time **and** the Model block is conditionally executed. See Configuration Parameters Support.

Changes for ERT and ERT-Based Targets

Compatibility Considerations: Yes

In R2012a, the simplified model call interface used by ERT targets has been further streamlined. (The simplified call interface also is now available to GRT target users — see Simplified Call Interface for Generated Code in the R2012a Simulink Coder Release Notes.) With the call interface enhancements come some compatibility considerations for static ERT main program (`ert_main.c`) files created before R2012a.

Compatibility Considerations

ERT Main Programs Now Include rtmodel.h Instead of autobuild.h

- In previous releases, GRT-based main programs such as `grt_main.c` and `grt_malloc_main.c` included `rtmodel.h` (which includes *`model`*`.h`) to access model-specific data structures and entry points. However, the static ERT main program `ert_main.c` included a different file, `autobuild.h`.

- Beginning in R2012a, GRT and static ERT main programs include `rtmodel.h`. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to include `rtmodel.h` instead of `autobuild.h`.

tid Argument to Model Step or Model Output/Update Function No Longer Generated As part of streamlining the model call interface, code generation no longer generates the *`tid`* argument to *`model`*`_step` or *`model`*`_output`/*`model`*`_update` functions in multirate, single-tasking models. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to remove the *`tid`* argument in model function calls.

firstTime Argument to Model Initialize Function No Longer Generated As part of streamlining the model call interface, code generation no longer generates the *`firstTime`* argument to the *`model`*`_initialize` function. If you have a static ERT main program created before R2012a that you want to use with R2012a generated code, update the main program to remove the *`firstTime`* argument in `model_initialize` function calls.

Note The target configuration parameter `ERTFirstTimeCompliant` and the model configuration parameter `IncludeERTFirstTime` will be removed from the Embedded Coder software in a future release.

MAT-file Logging and External Mode Calls Moved from Model Code to Main Program As part of streamlining the model call interface, some MAT-file logging and External mode calls have been moved from the generated model code in `model.c` or `.cpp` to the main program code in `ert_main.c`. MAT-file logging and External mode calls are not heavily used in production code environments. However, if you have a static ERT main program created before R2012a that you want to use with R2012a generated code, and if you do want to support MAT-file logging or External mode, update the main program to add the MAT-file logging and External mode calls.

Changes for Embedded IDEs and Embedded Targets

- "Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE" on page 86
- "Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors" on page 86
- "Support Removed for Freescale MPC5xx" on page 86
- "Limitation: Parallel Builds Not Supported for Embedded Targets" on page 87

Support Added for GCC 4.4 on Host Computers Running Linux with Eclipse IDE

Embedded Coder software now supports version 4.4 of GCC on host computers running Linux with Eclipse IDE. This support is on both 32-bit and 64-bit host Linux platforms.

If you were using an earlier version of GCC on Linux with Eclipse, upgrade to GCC 4.4.

Support Added for Using Processor-in-the-Loop (PIL) with Serial Communication Interface (SCI) for TI C2000 Processors

You can now perform PIL simulation over a SCI interface with Texas Instruments C280x, C2802x, C2803x, C28x3x, c2834x processors. Previously, this capability was supported only for TI C28035 and C28335 processors.

Support Removed for Freescale MPC5xx

This release removes support for the Freescale MPC5xx processor family from the Embedded Coder product.

Attempting to generate code from models that contain blocks for Freescale MPC5xx hardware produces an error message.

Limitation: Parallel Builds Not Supported for Embedded Targets

The Simulink Coder product provides an API for MATLAB Distributed Computing Server™ and Parallel Computing Toolbox™ products. The API allows these products to perform parallel builds that reduce build time for referenced models. However, the API does not support parallel builds for models whose **System target file** parameter is set to `idelink_ert.tlc` or `idelink_grt.tlc`. Thus, you cannot perform parallel builds for Embedded Targets.

New and Enhanced Demos

The following demos have been added in R2012a:

Demo...	**Shows How You Can...**
`rtwdemo_roll_axis`	Generate code for a roll axis autopilot control system. The `rtwdemo_roll` model represents a basic roll axis autopilot with two operating modes: roll attitude hold and heading hold. `rtwdemo_roll` replaces `rtwdemo_f14`.
`c28335_pmsmfoc_script`	Schedule a multi-rate controller for a permanent magnet synchronous machine (PMSM) motor control application that runs on a Texas Instruments F28335 processor.

The following demos have been enhanced in R2012a:

Demo...	**Now...**
`coderdemo_crl`	Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs).
`rtwdemo_crl_script`	• Reflects the renaming of target function libraries (TFLs) to code replacement libraries (CRLs). • Illustrates code replacement for Simulink matrix division and inversion operators.

Demo...	Now...
`rtwdemo_pmsmfoc_script`	Added torque and position control modes to controller, parameterized motor and sensor data, and added support for specifying baud rate in example PIL implementation.
`rtwdemo_radar`	Shows how to simulate and generate code for the model `rtwdemo_eml_aero_radar`, which contains a MATLAB script.
`rtwdemo_configuration_set`	Shows how to use the Code Generation Advisor and to automate the process of configuring a model for simulation and code generation.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2012a Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2012a &keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2012a&product=EC`

R2011b

Version: 6.1

New Features: Yes

Bug Fixes: Yes

Static Code Metrics in Code Generation Report

The HTML code generation report now includes a static code metrics report. The static code metrics include: number of source code files, number of lines of code, list of global variables, functions in a call tree format, and the estimated stack size required for a function.

To generate the static code metrics report, on the **Code Generation > Report** pane of the Configuration Parameters dialog box, select the **Static code metrics** parameter and build your model. For more information, see Analyze Static Code Metrics of the Generated Code.

AUTOSAR Enhancements

Import and Export of AUTOSAR Sensor/Actuator Components

Embedded Coder now supports Sensor/Actuator Software Components. The key difference between a sensor/actuator component and an application component is that a sensor/actuator component can access the I/O hardware abstraction part within the ECU abstraction layer.

This support allows you to import sensor/actuator components, implement and test designs within Simulink, and export sensor/actuator components. For more information, see Use the Configure AUTOSAR Interface Dialog Box.

Improved Simulink Library Support for Multiple Runnables

Previously, Embedded Coder did not support the creation of multiple runnables from subsystems with links to Simulink library blocks. For example, you had to disable and break links to library blocks in order to configure and validate the subsystems as AUTOSAR runnables.

Now, the software supports the creation of multiple runnables when:

- The wrapper subsystem (containing function-call subsystems) is a link to a library block
- The function-call subsystems (within the wrapper subsystem) are links to library blocks

For more information, see Configure Multiple Runnables in the Embedded Coder documentation.

AUTOSAR Schema Version 3.2

The software now supports AUTOSAR schema version 3.2 (3.2.1). See Select an AUTOSAR Schema.

Export AUTOSAR XML as Single File

When you export an AUTOSAR Software Component, you can generate XML as either a set of files (default) or a single file. The latter option is new. For more information, see Use the Configure AUTOSAR Interface Dialog Box.

SIL and PIL Enhancements

R2011b supports the following enhancements for software-in-the loop (SIL) and processor-in-the-loop (PIL) simulations.

Code Execution Profiling of Functions in Subsystems and Model Blocks

Previously, you could generate a profile of code execution times only for tasks within your generated code (for example, the step function for a sample rate). Now, you can also produce a profile of code execution times for functions generated from atomic subsystems and model reference hierarchies within the top model. The software places instrumentation probes inside these functions and calculates execution times during a SIL or PIL simulation. At the end of the simulation, you can view an HTML report and analyze execution times within the MATLAB environment:

- The HTML report provides a summary of maximum and average execution times, which allows you to identify code that requires optimization
- The supplied APIs allow you to carry out further analysis of time measurements.

For more information, see Code Execution Profiling in the Embedded Coder documentation.

Code Coverage with LDRA Testbed

You can measure code coverage using the LDRA Testbed from LDRA Software Technology. For more information, see Code Coverage.

BitField and GetSet Custom Storage Classes

The software previously did not support the `BitField` and `GetSet` custom storage classes. Now, the software supports these custom storage classes for all types of SIL and PIL simulations, with one limitation. `GetSet` behavior for the SIL block is different from top-model SIL/PIL, Model block SIL/PIL, and PIL block:

- SIL block — The C definitions of the `Get` and `Set` functions that you provide form part of the algorithm under test.
- Other types of SIL/PIL — The SIL/PIL test harness automatically provides C definitions of the `Get` and `Set` functions that are used during SIL/PIL simulations. In addition, the software supports only *scalar* signals, parameters and global data stores.

For more information, see I/O Support and GetSet Custom Storage Class.

Model Blocks with Variable-Size Signals

You can run Model block SIL and PIL simulations where the Model block contains variable-size signals. On the **Simulation > Configuration Parameters > Model Referencing** pane, in the **Propagate sizes of variable-size signals** field, you must specify `During execution`. See I/O Support.

Verification of Generated C++ Code

Previously, support for C++ was restricted to simulations with the SIL block. Now, you can verify generated C++ code using all types of SIL and PIL:

- Top-model
- Model block
- SIL or PIL block

As before, only the SIL block supports C++ encapsulation. See Configuration Parameters Support.

Generate Multitasking Code for Concurrent Execution on Multicore Processors

The Embedded Coder product extends the concurrent execution modeling capability of the Simulink product. With Embedded Coder, you can generate multitasking code that uses POSIX threads (Pthreads) for concurrent execution on multicore processors running Linux or VxWorks.

See Configuring Models for Targets with Multicore Processors.

Changes for Embedded IDEs and Embedded Targets

- "64-bit Version of Embedded Coder Supports Analog Devices™ VisualDSP++® and Texas Instruments™ Code Composer Studio™ 3.3 and 4.0" on page 96
- "Support Added for Wind River VxWorks 6.8" on page 96
- "Support Added for Serial Communications Interface with Processor-in-the-loop (PIL) for Texas Instruments™ C28035 and C28335" on page 96
- "New Target Function Library for Intel IPP/SSE (GNU)" on page 97
- "Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors" on page 97
- "Support Removed for Altium TASKING" on page 98
- "Support Removed for Infineon® C166®" on page 98
- "Support Ending for Green Hills® MULTI® in a Future Release" on page 98
- "Support Ending for Freescale MPC5xx in a Future Release" on page 98

64-bit Version of Embedded Coder Supports Analog Devices VisualDSP++ and Texas Instruments Code Composer Studio 3.3 and 4.0

Installing MATLAB & Simulink on a 64-bit Windows computer automatically installs the 64-bit versions of your MathWorks® products, including Embedded Coder software. Now, you can use the 64-bit version of Embedded Coder software with the following 32-bit IDEs/tool chains:

- Texas Instruments Code Composer Studio 3.3
- Texas Instruments Code Composer Studio 4.0
- Analog Devices VisualDSP++ 5.0 (update 8)

Previously, you had to install the 32-bit versions of your MathWorks products to use Embedded Coder software with these IDEs.

For more information, see http://www.mathworks.com/hardware-support/texas-instruments.html and http://www.mathworks.com/hardware-support/analog-devices.html.

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

Support Added for Wind River VxWorks 6.8

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

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

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

For more information, see Serial Communication Interface (SCI) for Texas Instruments C2000, Example — Performing a Model Block PIL Simulation via SCI Using Makefiles, and the fuelsys_pil demo.

New Target Function Library for Intel IPP/SSE (GNU)

This release adds a new Target Function Library (TFL), `Intel IPP/SSE (GNU)`, for the GCC compiler. This library includes the Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE) code replacements.

For more information, see Code Replacement Library (CRL) and Embedded TargetsDesktop Targets.

Support Added for Single Instruction Multiple Data (SIMD) with ARM Cortex-A8, ARM Cortex-A9 , and Intel Processors

This release adds support for the Single Instruction Multiple Data (SIMD) capabilities of the ARM Cortex-A8, ARM Cortex-A9 , and Intel® processors. The use of SIMD instructions increases throughput compared to traditional Single Instruction Single Data (SISD) processing.

The following TFLs (code replacement libraries) optimize generated code for SIMD:

- `GCC ARM Cortex-A8` — The GCC compiler and the ARM Cortex-A8 embedded processor
- `GCC ARM Cortex-A9` — The GCC compiler and the ARM Cortex-A9 embedded processor
- `Intel IPP/SSE (GNU)` — The GCC compiler and the Intel Performance Primitives (IPP) and Streaming SIMD Extensions (SSE)

The performance of the SIMD-enabled executable depends on several factors, including:

- Processor architecture of the target
- Optimized library support for the target
- The type and number of TFL replacements in the generated algorithmic code

Evaluate the performance of your application before and after using the TFL.

To use SIMD capabilities, enable the corresponding TFLs as described in Code Replacement Library (CRL) and Embedded TargetsDesktop Targets.

Support Removed for Altium TASKING

Support for the Altium® TASKING IDE has been removed from the Embedded Coder product.

Support Removed for Infineon C166

Support for the Infineon® C166® processor family has been removed from the Embedded Coder product.

Support Ending for Green Hills MULTI in a Future Release

Support for the Green Hills MULTI IDE will end in a future release of the Embedded Coder product.

Support Ending for Freescale MPC5xx in a Future Release

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

Saturation Control of Stateflow Data

A new property for Stateflow charts, **Saturate on integer overflow**, enables you to control the behavior of data with signed integer types when overflow occurs. This check box appears in the Chart properties dialog box.

Check Box	When to Use This Setting	Overflow Handling	Example of a Result
Selected	Overflow is possible for data in your Stateflow chart and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer saturates to –128 or +127.
Cleared	You want to optimize efficiency of the generated code.	The behavior depends on the C compiler you use for generating code.	The number 130 does not fit in a signed 8-bit integer and wraps to –126.

Arithmetic operations in the chart for which you can enable saturation protection are:

- Unary minus: `–a`
- Binary operations: `a + b`, `a – b`, `a * b`, `a / b`, `a ^ b`
- Assignment operations: `a += b`, `a –= b`, `a *= b`, `a /= b`

For new charts, this check box is selected by default. When you open charts saved in previous releases, the check box is cleared to maintain backward compatibility.

For more information, see Handling Integer Overflow for Chart Data in the Stateflow User's Guide.

Custom Storage Class Properties for Managing Data Ownership and Definition

Compatibility Considerations: Yes

In R2011b, use the **Owner** and **Definition File** properties of custom storage classes to manage the definition and ownership of `mpt` data objects in generated code.

Previously, you could include the data definitions in generated code but could not specify the model that defined the data. Now, Embedded Coder creates the data definitions in the generated code according to the **Owner** property.

The **Owner** property of a custom storage class specifies the model that owns and defines the data in the generated code. The **Definition File** property specifies a name for the data definition file that Embedded Coder generates.

Compatibility Considerations

- If your legacy code exports data definitions to generated code and you now specify the **Owner** property, your generated code might have duplicate data definitions. This duplication causes a link error. In this case, remove the data definitions from the legacy code.
- If your legacy code does not export data definitions to generated code and you now specify the **Owner** property, your generated code might not contain data definitions. This mismatch causes a link error. In this case, add the missing data definitions to your legacy code.

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

When generating code with model reference, you can export shared data declarations to a specific header file in a shared directory.

Specify a data declaration header file in the following ways:

- For a data object: In the **Code generation** options section of the data object dialog
- For a model: In the **Code Generation > Code Placement** section of the **Configuration Parameters** dialog

Specify the option to use a **Shared location** in the field **Shared code placement** in **Code Generation > Interface** section of the **Configuration Parameters** dialog.

Target Function Library Code Replacement Enhancements

R2011b provides the following enhancements to code replacement using target function libraries (TFLs).

Code Replacement Tool for Creating and Managing TFL Tables

R2011b provides the Code Replacement Tool, which helps you create and manage the code replacement tables that make up a TFL. You can:

- Create a new code replacement table or import existing tables.
- Add, modify, and delete table entries. Each table entry represents a potential code replacement for a single function or operator. You can manage multiple tables together and copy and paste entries between tables.
- Validate tables and table entries.
- Save code replacement tables as MATLAB files.
- Generate the customization file you use to register your code replacement tables with code generation software.

Each code replacement table contains one or more table entries. Each table entry represents a potential replacement, during code generation, of a single function or operator by a custom implementation. For each table entry, you provide:

- **Mapping Information**, which relates a conceptual view of the function or operator (similar to the Simulink block view of the function or operator) to a custom implementation of that function or operator.
- **Build Information**, which provides header, source, or link information required for building the custom implementation.

You can open the Code Replacement Tool in the following ways:

- Go to the **Interface** pane of the Configuration Parameters dialog box and click the **Custom** button, which is located to the right of the **Target function library** parameter.
- Use the MATLAB command `crtool`.

For more information about creating code replacement tables for TFLs, see Create and Manage Code Replacement Tables Using the Code Replacement Tool.

Ability to Align Data Objects to TFL-Specified Boundaries to Boost Code Performance

R2011b provides the ability to align data objects passed into a TFL replacement function to a specified boundary. This allows you to take advantage of target-specific function implementations that require data to be aligned in order to optimize application performance. To configure data alignment for a function implementation:

1. Specify the data alignment requirements in a TFL table entry. Alignment can be specified separately for each implementation function argument or collectively for all function arguments.
2. Specify the data alignment capabilities and syntax for one or more compilers, and include the alignment specifications in a TFL registry entry in an `sl_customization.m` or `rtwTargetInfo.m` file.

For more information on specifying data alignment requirements and compiler alignment attributes, see Configure Data Alignment for Function Implementations.

For additional examples of configuring data alignment for function implementations, see the demo `rtwdemo_tfl_script`.

Support for Replacing Element-wise Matrix Multiply

TFLs support several nonscalar operators for replacement with custom library functions in generated model code. R2011b adds support for replacing element-wise matrix multiplication operations (`.*` operator in element-wise mode) with custom implementations. For more information, see Map Nonscalar Operators to Target-Specific Implementations.

Code Generation Enhancements

Redundant Condition Checks

Multiple checks of the same condition are difficult to avoid in modeling. For example, a common modeling pattern is Switch blocks sharing the same condition check. Previously, the generated code for multiple Switch blocks produced multiple `if` statements.

```
if (cond) {
   true_statement1;
} else {
   false_statement1; }
if (cond) {
   true_statement2;
} else {
   false_statement2;
}
```

In R2011b, the generated code combines these condition checks. For example, the generated code for Switch blocks with a common condition combines these multiple `if` statements.

```
if (cond) {
   true_statement1;
   true_statement2;
}
else {
   false_statement1;
   false_statement2;
}
```

This optimization reduces code size and execution time. As a result, other optimizations for condition expressions or merged branches are enabled which reduce data copies and RAM usage.

Loop Fusion

R2011b provides more precise data dependency analysis of the data and signals of a nested Simulink bus. This enhancement enables more loop fusion in the generated code which reduces code execution time and ROM, and improves code readability.

Invariant Condition Check Lifting

When a condition check is invariant to the enclosing loop and you specify loops to be unrolled, the code generator lifts the check out of the loop. This enhancement reduces ROM, enables additional optimizations, and improves execution speed and code readability. For more information on loop unrolling, see Configure Loop Unrolling Threshold.

Parameter Pooling for Stateflow and Interpreted MATLAB Function Blocks

Parameter pooling now occurs for Simulink matrix constants used as Stateflow graphical function arguments. This enhancement reduces RAM and ROM, and improves thread safety.

Readability Improvement for Reusable Subsystem Input and Output

The generated code for reusable subsystem input and output now eliminates redundant operators and unnecessary parentheses. This enhancement improves code readability.

Enhanced Code Generation Optimization Using Minimum and Maximum Values

The **Optimize using specified minimum and maximum values** code generation option now takes into account the minimum and maximum values specified for `Simulink.Parameter` objects even if the object is part of an expression. For example, consider a Gain block with a gain parameter specified as an expression such as `k1 + 5`, where `k1` is a `Simulink.Parameter` object with `k1.min = -10` and `k1.max = 10`. If minimum and maximum values of the parameter specified in the parameter dialog box are `0` and `20`, the range calculated for this parameter expression is `0` to `15`.

For more information, see Optimize Generated Code Using Specified Minimum and Maximum Values.

New Model Advisor Check for Code Efficiency of Logic Blocks

The Simulink Model Advisor includes the following new check for code efficiency of logic blocks: Check output types of logic blocks. The following blocks in the Simulink Logic and Bit Operations library can use `boolean` or another setting for the output data type:

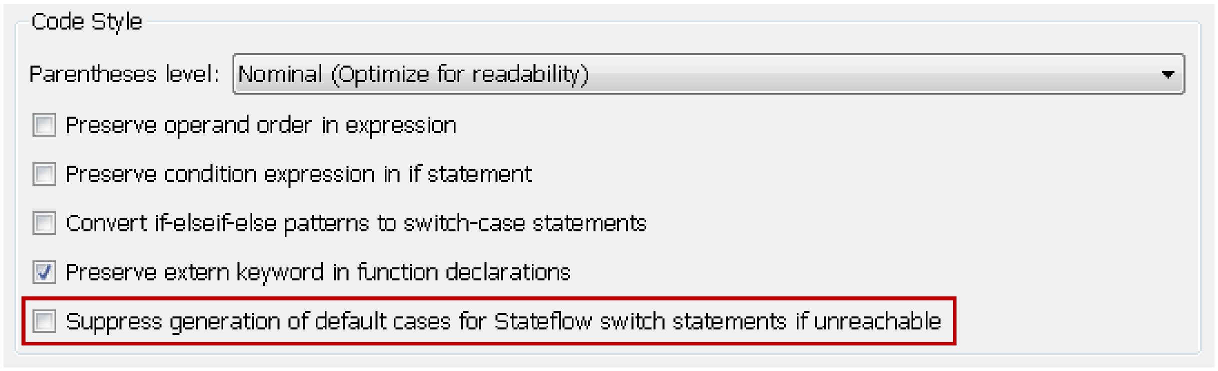
- Compare To Constant
- Compare To Zero
- Detect Change
- Detect Decrease
- Detect Fall Negative
- Detect Fall Nonpositive
- Detect Increase
- Detect Rise Nonnegative
- Detect Rise Positive
- Interval Test
- Interval Test Dynamic
- Logical Operator
- Relational Operator

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

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

Control of Default Case Generation for Switch Statements in Generated Code for Stateflow Charts

You can specify whether or not to generate default cases for switch statements in the generated code for Stateflow charts. This optimization works on a per-model basis and applies to the code generated for a state that has multiple substates. Use the following check box on the **Code Generation > Code Style** pane of the Configuration Parameters dialog box:


Code Style

Parentheses level: Nominal (Optimize for readability)

- [ ] Preserve operand order in expression
- [ ] Preserve condition expression in if statement
- [ ] Convert if-elseif-else patterns to switch-case statements
- [x] Preserve extern keyword in function declarations
- [ ] Suppress generation of default cases for Stateflow switch statements if unreachable

Check Box	When to Use This Setting	Format of Switch Statements
Selected	Provide better code coverage by checking that every branch in the generated code is falsifiable.	Exclude the default case when it is unreachable.
Cleared	Check for MISRA C® compliance and provide a fallback in case of RAM corruption.	Include a default case.

For new models, this check box is cleared by default. When you open models saved in previous releases, the check box is also cleared to maintain backward compatibility.

For more information, see Code Generation Pane: Code Style in the Embedded Coder Reference documentation.

Improvement to Build Process for Conflicting Identifiers

Previously, if your model contained two referenced models with the same input (or output) port names, the model might not build because of potentially conflicting identifiers. The failure to build happens when the generated identifiers exceed the Maximum identifier length. In R2011b, the build process is improved to handle more cases when two referenced models have the same input (or output) port names. For more information, see Model Referencing Considerations.

Update to Code Generation Verification Class `cgv.Config`

Compatibility Considerations: Yes

Compatibility Considerations

The `Connectivity` cgv.Config parameter has the following updates:

- `pil` replaces the `custom` value. In R2011b, you can use `custom` without producing a warning or error message.
- The `tasking` value is not available. Specifying `tasking` produces an error.

License Names Not Yet Updated for Coder Product Restructuring

The Simulink Coder and Embedded Coder license name strings stored in `license.dat` and returned by the `license ('inuse')` function have not yet been updated for the R2011a coder product restructuring. Specifically, the `license ('inuse')` function continues to return `'real-time_workshop'` for Simulink Coder and `'rtw_embedded_coder'` for Embedded Coder, as shown below:

```
>> license('inuse')
matlab
matlab_coder
real-time_workshop
```

```
rtw_embedded_coder
simulink
>>
```

The license name strings intentionally were not changed, in order to avoid license management complications in situations where Release 2011a or higher is used alongside a preR2011a release in a common operating environment. MathWorks plans to address this issue in a future release.

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

New and Enhanced Demos

The following demos have been enhanced in R2011b:

Demo...	Now...
`rtwdemo_pmsmfoc_script`	Shows how you can perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine
`rtwdemo_sil_pil_script`	Incorporates code execution profiling
`rtwdemo_tfl_script`	Shows how you can align nonscalar data passed into a target function library (TFL) code replacement function
`fuelsys_pil`	Incorporates using serial communication interface to communicate during PIL simulation

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2011b Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2011b`
`&keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2011b&product=EC`

R2011a

Version: 6.0

New Features: Yes

Bug Fixes: Yes

Coder Product Restructuring

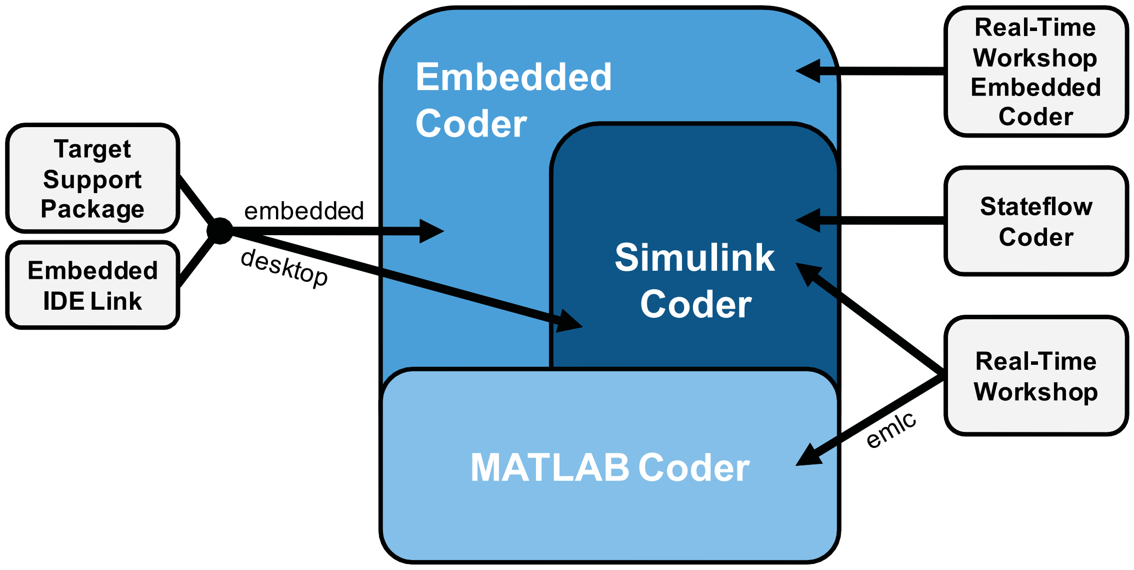
Compatibility Considerations: Yes

Product Restructuring Overview

In R2011a, the Embedded Coder product replaces the Real-Time Workshop® Embedded Coder product. Additionally,

- The Simulink Coder product combines and replaces the Real-Time Workshop and Stateflow Coder products
- The Real-Time Workshop facility for converting MATLAB code to C/C++ code, formerly referred to as Embedded MATLAB® Coder, has migrated to the new MATLAB Coder product.
- The previously existing Embedded IDE Link™ and Target Support Package™ products have been integrated into the new Simulink Coder and Embedded Coder products.

The following figure shows the R2011a transitions for C/C++ code generation related products, from the R2010b products to the new MATLAB Coder, Simulink Coder, and Embedded Coder products.



Resources for Upgrading from Real-Time Workshop Embedded Coder

If you are upgrading to Embedded Coder from Real-Time Workshop Embedded Coder, review information about compatibility and upgrade issues at the following locations:

- *Release Notes for Embedded Coder* (latest release), "Compatibility Summary" section
- On the MathWorks web site, in the Archived documentation, select R2010b, and view the following tables, which are provided in the release notes for Real-Time Workshop Embedded Coder: *Compatibility Summary for Real-Time Workshop Embedded Coder Software*:

  This table provides compatibility information for releases up through R2010b.
- If you use the Embedded IDE Link or Target Support Package capabilities that now are integrated into Simulink Coder and Embedded Coder, go to the Archived documentation and view the corresponding tables for Embedded IDE Link or Target Support Package:

- *Compatibility Summary for Embedded IDE Link* (R2010b)
- *Compatibility Summary for Target Support Package* (R2010b)

You can also refer to the rest of the archived documentation, including release notes, for the Real-Time Workshop, Stateflow Coder, Embedded IDE Link, and Target Support Package products.

Migration of Embedded MATLAB Coder Features to MATLAB Coder

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

Migration of Embedded IDE Link and Target Support Package Features to Simulink Coder and Embedded Coder

In R2011a, the capabilities formerly provided by the Embedded IDE Link and Target Support Package products have been integrated into Simulink Coder and Embedded Coder. The following table summarizes the transition of the Embedded IDE Link and Target Support Package supported hardware and software into Coder products.

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
Embedded IDE Link	Altium TASKING		x
	Analog Devices VisualDSP++		x
	Eclipse IDE	x	x
	Green Hills MULTI		x
	Texas Instruments Code Composer Studio		x

Former Product	Supported Hardware and Software	Simulink Coder	Embedded Coder
Target Support Package	Analog Devices Blackfin		x
	ARM		x
	Freescale MPC5xx		x
	Infineon C166		x
	Texas Instruments C2000		x
	Texas Instruments C5000™		x
	Texas Instruments C6000		x
	Linux OS	x	x
	Windows OS	x	
	VxWorks RTOS		x

Interface Changes Related to Product Restructuring

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

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

Simulink Graphical User Interface Changes

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

Compatibility Considerations

In the Help browser **Contents** pane, Embedded Coder is now listed with the products for MATLAB, because Embedded Coder now supports both MATLAB Coder and Simulink Coder workflows.

Data Management Enhancements and Changes

Compatibility Considerations: Yes

- "Memory Section Enhancements" on page 117
- "No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects" on page 117
- "Parts of Data Class Infrastructure Not Available" on page 118
- "No Longer Generating Pragma for Data Defined with Built-In Storage Class ExportedGlobal, ImportedExtern, or ImportedExternPointer" on page 119
- "Simulink.CustomParameter and Simulink.CustomSignal Data Classes To Be Deprecated in a Future Release" on page 120

Memory Section Enhancements

- `Pragmas` are now added to data and function declarations (prior to R2011a they were added to definitions only); at compile time, this makes the compiler aware of memory locations for functions and data, potentially optimizing generated code
- New function category is available for shared utilities on the **Code Generation > Memory Sections** pane: Shared utility
- Referenced models can have a memory section that is different from that of the top model for the `InitTerm` and `Execute` function categories

No Longer Able to Set RTWInfo or CustomAttributes Property of Simulink Data Objects

You can not set the `RTWInfo` or `CustomAttributes` property of a Simulink data object from the MATLAB Command Window or a MATLAB script. Attempts to set these properties generate an error.

Although you cannot set `RTWInfo` or `CustomAttributes`, you can still set subproperties of `RTWInfo` and `CustomAttributes`.

Compatibility Considerations

Operations from the MATLAB Command Window or a MATLAB script, which set the data object property `RTWInfo` or `CustomAttributes`, generate an error.

For example, a MATLAB script might set these properties by copying a data object as shown below:

```
a = Simulink.Parameter;
b = Simulink.Parameter;
b.RTWInfo = a.RTWInfo;
b.RTWInfo.CustomAttributes = a.RTWInfo.CustomAttributes;
 .
 .
 .
```

To copy a data object, use the object's `deepCopy` method.

```
a = Simulink.Parameter;
b = a.deepCopy;
.
.
.
```

Parts of Data Class Infrastructure Not Available

Simulink has been generating warnings for usage of the following data class infrastructure features for several releases. As of R2011a, the features are not supported.

- Custom storage classes not captured in the custom storage class registration file (`csc_registration`) – *warning displayed since R14SP2*
- Built-in custom data class attributes `BitFieldName` and `FileName+IncludeDelimiter` – *warning displayed since R2008b*

Instead of...	Use...
`BitFieldName`	`StructName`
`FileName+IncludeDelimiter`	`HeaderFile`

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

Compatibility Considerations

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

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

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

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

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

For Built-In Storage Class...	Generate `pragma` Around...
`ExportedGlobal`	Data definition and declaration
`ImportedExtern`	Data declaration
`ImportedExternPointer`	Data declaration

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

Compatibility Considerations

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

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

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

Compatibility Considerations

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

AUTOSAR Enhancements

The following enhancements are available in R2011a.

Calibration Parameters

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

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

For more information, see Calibration Parameters and Configure Calibration Parameters in the Embedded Coder documentation.

Multiple Runnables from Virtual Subsystems

Previously, if a wrapper subsystem had virtual subsystems containing function-call subsystems, you could not export the function-call subsystems as AUTOSAR runnables from the wrapper subsystem level. Now, within a wrapper subsystem, you can group function-call subsystems into virtual subsystems and generate runnables for these function-call subsystems. See Configure Multiple Runnables and Export AUTOSAR Software Component in the Embedded Coder documentation.

Support for Code Descriptor Elements

The AUTOSAR standard specifies that the XML description of an AUTOSAR Software Component implementation must contain code descriptor elements to describe generated source files and include header files. This feature allows AUTOSAR authoring tools that import software components to automate the building process for source code.

Previously, the software did not generate the software component implementation file (`*modelname*_implementation.arxml`) with these code descriptor elements. Now, when you build a Simulink model for an AUTOSAR target, the software generates a `CODE-DESCRIPTORS` element within the `SWC_IMPLEMENTATION` element. The `CODE-DESCRIPTORS` element contains `XFILE` elements that provide descriptions of the generated code.

For example, if you build the model `rtwdemo_autosar_counter`, the generated file `rtwdemo_autosar_counter_implementation.arxml` has the following `SWC_IMPLEMENTATION` element:

```
....
<SWC-IMPLEMENTATION>
  <SHORT-NAME>rtwdemo_autosar_counter</SHORT-NAME>
  <CODE-DESCRIPTORS>
    <CODE>
      <SHORT-NAME>Code</SHORT-NAME>
      <TYPE>SRC</TYPE>
      <XFILES>
```

```
          <XFILE>
            <SHORT-NAME>rtwdemo_autosar_counter_c</SHORT-NAME>
            <CATEGORY>GeneratedFile</CATEGORY>
            <URL>rtwdemo_autosar_counter_autosar_rtw\rtwdemo_autosar_counter.c</URL>
            <TOOL>Embedded Coder</TOOL>
            <TOOL-VERSION>5.6</TOOL-VERSION>
          </XFILE>
          <XFILE>
            <SHORT-NAME>rtwdemo_autosar_counter_h</SHORT-NAME>
            <CATEGORY>GeneratedFile</CATEGORY>
            <URL>rtwdemo_autosar_counter_autosar_rtw\rtwdemo_autosar_counter.h</URL>
            <TOOL>Embedded Coder</TOOL>
            <TOOL-VERSION>5.6</TOOL-VERSION>
          </XFILE>
            ...
        </XFILES>
      </CODE>
    </CODE-DESCRIPTORS>
    <CODE-GENERATOR>Embedded Coder 5.6 (R2011a) 26-Aug-2010</CODE-GENERATOR>
    <PROGRAMMING-LANGUAGE>C</PROGRAMMING-LANGUAGE>
  </SWC-IMPLEMENTATION>
  ....
```

SIL and PIL Enhancements

Code Execution Profiling

You can collect execution time measurements in a specified base workspace variable during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. At the end of the simulation, you can view or analyze the measurements within the MATLAB environment. This feature allows you to collect an execution time profile for each task within your generated code.

The software supports code execution profiling for all types of SIL or PIL simulations except the SIL block.

For more information, see Code Execution Profiling in the Embedded Coder documentation.

PIL Block Parameter Tuning

R2011a supports parameter tuning for the PIL block, which allows you to change tunable workspace parameters between or during simulations without regenerating code. This feature also includes support for tunable structure parameters. For more information, see I/O Support and Tunable Parameters and SIL/PIL.

Top-Model SIL/PIL and PIL Block Parameter Initialization

R2011a supports automatic definition and initialization of parameters with imported storage classes. For more information, see I/O Support and Imported Data Definitions.

Model Block Parameter Tuning and Model Initialization

Previously, the software did not support the following features for Model block SIL/PIL:

- Simplified initialization mode
- Tunable structure parameters

R2011a now supports these features. For more information, see Configuration Parameters Support, I/O Support, and Tunable Parameters and SIL/PIL.

Code Generation Enhancements

Improved Code for Data Store Memory In-place Assignment

Previously, the generated code for a Data Store Memory block used data copies to perform data store assignments. The generated code now eliminates the data copies and performs an in-place assignment. This improvement generates less code, uses less memory, and provides faster execution.

Improvements to Target Function Library Replacements

Enhancements to Target Function Library Replacements (TFL) include:

- If multiple TFL replacements occur within a function, temporary variables are now reused instead of creating extra temporary variables. This enhancement reduces the stack size during TFL replacement.
- During TFL replacement, if unnecessary temporary variables are introduced when block output is not the returned value of the function but one of the input arguments, code generation now removes the temporary variable. This enhancement improves execution speed and requires less memory.

For more information, see Introduction to Code Replacement Libraries.

Improved Loop Fusion

Code generation now includes the following:

- An improved loop fusion algorithm that reduces data copies. This enhancement decreases stack size, ROM consumption, and code generation time.
- Selectively fuses loops when the loop count is larger than the Loop unrolling threshold. In these cases, loop unrolling allows the code generator to perform more optimizations. In addition, the code generator groups the statements together to assign values to the elements of a signal or parameter array, which improves data access and code readability.

Improved Array Indexing

The generated code is optimized for more efficient array indexing. When a complex instruction is used repeatedly in an array index, the instruction is replaced with a temporary variable to perform the calculation more efficiently. This enhancement improves execution speed and reduces code size.

Improvement on Matrix Parameter Pooling

For matrix parameters with the same flattened value, the generated code now pools the matrix parameters even when they have different shapes. This enhancement reduces ROM consumption.

Readability Improvements Involving Data References

For references to the root inport and outport, as well as DWork, unnecessary parentheses are removed from the generated code. This enhancement produces more readable code.

Code Generation Verification (CGV) API Updates

Compatibility Considerations: Yes

Support for Adding Multiple Callback Functions

In R2011a, the cgv.CGV class includes new methods to add callback functions. These methods replace the `cgv.CGV.addCallback` method which added only a pre-execution callback function. Now, the new methods allow CGV to invoke callback functions at several stages of the cgv.CGV.run execution. The new methods are:

- cgv.CGV.addHeaderReportFcn adds a callback function invoked before executing input data in the `cgv.CGV` object.
- cgv.CGV.addPreExecReportFcn adds a callback function invoked before executing each input data file in the `cgv.CGV` object.
- cgv.CGV.addPreExecFcn adds a callback function invoked before executing each input data file in the `cgv.CGV` object.
- cgv.CGV.addPostExecReportFcn adds a callback function invoked after executing each input data file in the `cgv.CGV` object.
- cgv.CGV.addPostExecFcn adds a callback function invoked after executing each input data file in the `cgv.CGV` object.
- cgv.CGV.addTrailerReportFcn adds a callback function invoked after executing input data in the `cgv.CGV` object.

New Functionality Added to the cgv.CGV Class

The `cgv.CGV` class now includes the following methods:

- cgv.CGV.activateConfigSet activates the configuration set of a model.

- cgv.CGV.addBaseline adds a file of baseline data for comparison.
- cgv.CGV.copySetup creates a copy of a `cgv.CGV` object.
- cgv.CGV.setMode specifies the mode of execution (`sim`, `sil`, or `pil`).
- cgv.CGV.copySetup returns the status of the execution of the `cgv.CGV` object.

The `cgv.CGV` class now includes the following properties:

- `Name`
- `Description`

Compatibility Considerations

Previously, the `cgv.CGV` class included parameters that you set to perform automatic configuration checks of your model. In R2011a, `cgv.CGV` class does not performs automatic configuration checks. Instead, you can use the cgv.Config class to perform a manual configuration check of your model. Before calling `cgv.CGV.run`, perform a manual configuration check of your model. Otherwise, an error might occur later in the process. For more information, see Programmatic Code Generation Verification.

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

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

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

Parameter	What Happens When You Use Parameter?	Use This Parameter Instead	Compatibility Considerations
`CheckInterface` parameter from `cgv.CGV`	Warns if set to `off` Errors if set to `on`	`CheckOutports` parameter in `cgv.Config`	To check your model before running CGV, pass the `CheckOutports` parameter to the constructor for `cgv.Config`. Then call the cgv.Config.configModel method to adjust the model configuration.
`tasking` and `custom` values removed from the `Connectivity` parameter of `cgv.CGV`	Errors	`pil`, a new value for the `cgv.CGV` `Connectivity` parameter	Replace calls to the `cgv.CGV` constructor using the parameter-value arguments, `('Connectivity', 'tasking')` or `('Connectivity', 'custom')`, with `('Connectivity, 'pil')`.

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

Parameter	What Happens When You Use Parameter?	Compatibility Considerations
`CheckOutports` parameter added to `cgv.Config`	Defaults to `on`. Compiles the model. Then checks that the model outport configuration is compatible with the `cgv.CGV` object.	If your script fixes errors reported by `cgv.Config`, you can set `CheckOutports` to `off`.
`LogMode` parameter from `cgv.Config`	Change in behavior	If you do not give a value for `LogMode`, logging changes are not made to the configuration parameters.

MISRA-C Code Generation Objective

The Code Generation Advisor now includes a new objective for MISRA-C:2004 guidelines. To set the new objective, open the Configuration Parameters dialog box and select the **Code Generation** pane. In the Code Generation Advisor section, click the **Set objectives** button to open the Code Generation Advisor dialog box. In the **Available objectives** list, select `MISRA-C:2004 guidelines` and click the select button (arrow pointing right) to move the objective to the **Selected objectives** list. For more information on setting objectives, see Application Objectives.

New Model Advisor Check for Code Efficiency of Lookup Table Blocks

The Simulink Model Advisor includes the following new check for code efficiency of lookup table blocks: Identify lookup table blocks that generate expensive out-of-range checking code. By default, the following blocks generate code that checks for out-of-range breakpoint inputs:

- 1-D Lookup Table
- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

Similarly, the Interpolation Using Prelookup block generates code that checks for out-of-range index inputs. Running this Model Advisor check helps you identify lookup table blocks that generate out-of-range checking code for breakpoint or index inputs.

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

Enhanced Code Generation Optimization

The **Optimize using specified minimum and maximum values** code generation option now takes into account the minimum and maximum values specified for:

- A `Simulink.Parameter` object provided that it is used on its own. It does not use these minimum and maximum values if the object is part of an expression. For example, if a Gain block has a gain parameter specified as `K1`, where `K1` is defined as a `Simulink.Parameter` object in the base workspace, the optimization takes the minimum and maximum values of `K1` into account. However, if the Gain block has a gain parameter of `K1+5` or `K1+K2+K3`, where `K2` and `K3` are also `Simulink.Parameter` objects, the optimization does not use the minimum and maximum values of `K1`, `K2` or `K3`.
- Design ranges specified on block outputs in a conditionally-executed subsystem, except for the block outputs that are directly connected to an Outport block.

For more information, see Optimize Generated Code Using Specified Minimum and Maximum Values.

Target Function Library Replacement Based on Computation Method for Reciprocal Sqrt, Sine, and Cosine

Target function libraries (TFLs) now support the ability to control replacement of certain math functions using their computation method as a distinguishing attribute. For example,

- The rSqrt block can be configured to use either of two computation methods, `Newton-Raphson` or `Exact`.
- The Trigonometric Function block, with **Function** set to `sin` or `cos`, can be configured to use either of two approximation methods, `CORDIC` or `None`.

You can configure TFL table entries to replace these functions for one or all of the available computation methods. For example, you could replace only `Newton-Raphson` instances of the `rSqrt` function.

For more information, see Replace Math Functions Based on Computation Method in the Embedded Coder documentation.

Target Function Library Support for abs, min, max, and sign functions

Embedded Coder software now supports target function library customization control for fixed-point `abs`, `min`, `max`, and `sign` functions.

For more information, see Register Code Replacement Libraries.

C++ Encapsulation Allowed for Referenced Models in For Each Subsystems

In previous releases, due to a code generation limitation, code could not be generated for a For Each Subsystem block under the following conditions:

- The For Each Subsystem block directly or indirectly contains a Model block.
- The Model block references a model for which C++ encapsulation is selected.

R2011a removes this limitation. You can now generate code for a For Each Subsystem in which a referenced model uses C++ encapsulation.

Improved Code Generation for Portable Word Sizes

In the software-in-the-loop (SIL) simulation work flow, the model option **Enable portable word sizes** allows you to take code intended for a specific target platform and compile and run the same code on a MATLAB host platform that uses different processor word sizes. R2011a enhances the code generated for portable word sizes by inserting explicit casts to help protect against integral promotion differences and other behavior differences between host and target. This potentially can reduce the incidence of numerical differences due to host/target behavior differences. For more information, see Configure Hardware Implementation Settings for SIL and Portable Word Sizes Limitations in the Embedded Coder documentation.

Improved Comments in the Generated Code

R2011a provides improvements to comment generation for better readability and understanding of the generated code. Specifically, comments are located closer to the referring code and reflect the intent of the code. An end comment is now included at the end of a control flow block of code. For information on customizing comments in the generated code, see Configure Code Comments in Embedded System Code.

Replacement Data Types and Simulation Mode for Referenced Models

To replace built-in data type names with user-defined data type names in the generated code for a referenced model, you must set the **Simulation mode** parameter for the Model block to one of the following:

- `Normal`
- `Software-in-the-loop (SIL)`
- `Processor-in-the-loop (PIL)`

For more information, see Data Types and Referenced Model Simulation Modes in the Simulink documentation.

Changes for Embedded IDEs and Embedded Targets

Compatibility Considerations: Yes

- "Feature Support for Embedded IDEs and Embedded Targets" on page 133

Feature Support for Embedded IDEs and Embedded Targets

The Embedded Coder software provides the following features as implemented in the former Target Support Package and former Embedded IDE Link products:

- Automation Interface
- Processor-in-the-Loop (PIL) Simulation
- Execution Profiling
- Execution Profiling during PIL Simulation
- Stack Profiler
- External Mode

- Schedulers and Timing
- Makefile Generation (XMakefile)
- Target Function Library (TFL) Optimization
- Multicore Deployment for Rate Based Multithreading

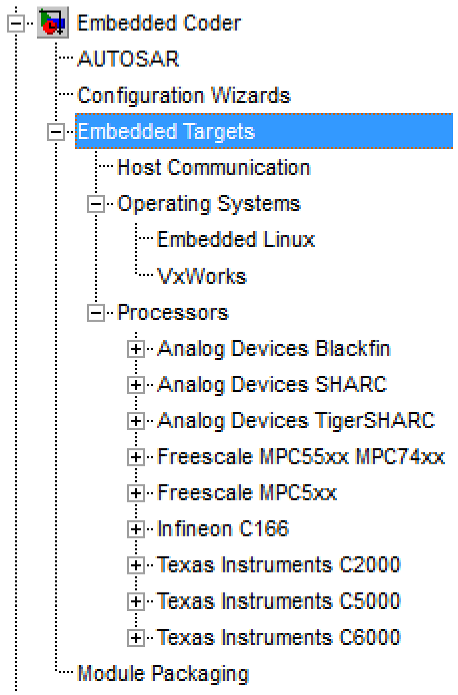
Note You can only use these features in the 32-bit version of your MathWorks products. To use these features on 64-bit hardware, install and run the 32-bit versions of your MathWorks products.

Execution Profiling during PIL Simulation

During Processor-in-the-loop (PIL) simulation, you can profile synchronous tasks in code running on the target. For more information, see Execution Profiling during PIL Simulation

Location of Blocks for Embedded Targets

Blocks from the former Target Support Package product and Embedded IDE Link product now reside under Embedded Coder in the Embedded Targets block library, as shown.



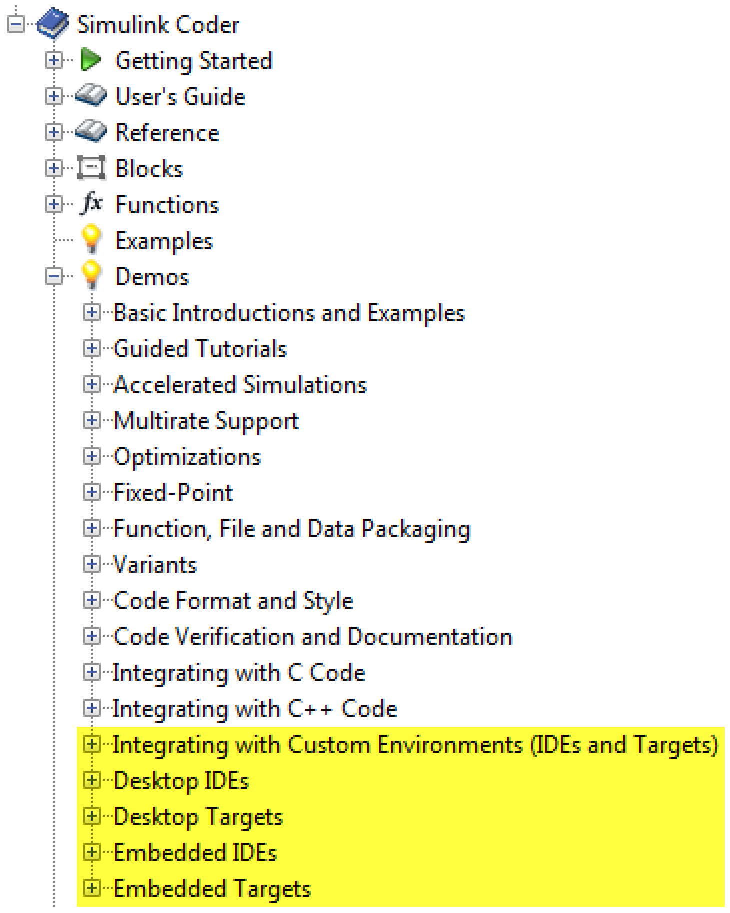
Embedded Targets includes the following types of blocks:

- Host Communication
- Operating Systems
  - Embedded Linux
  - VxWorks
- Processors
  - Analog Devices Blackfin
  - Analog Devices SHARC
  - Analog Devices TigerSHARC
  - Freescale MPC55xx MPC74xx
  - Freescale MPC5xx
  - Infineon C166

- Texas Instruments C2000
- Texas Instruments C5000
- Texas Instruments C6000

Location of Demos for Embedded IDEs and Embedded Targets

Demos from the former Target Support Package product and Embedded IDE Link product now reside under Simulink Coder product help. Click the expandable links, as shown.



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 Embedded Linux and

VxWorks. This feature improves performance by taking advantage of multicore hardware resources.

Also see the Running Target Applications on Multicore Processors user's guide topic.

Windows-Based Code Generation and Remote Build On Linux Target (BeagleBoard)

You can generate a makefile project on a Windows host machine, transfer the makefile project to an remote target running Linux, such as a BeagleBoard, and then build the executable on the remote target.

Changes to Frame-Based Processing

Signal processing applications often process sequential samples of data at once as a group, rather than one sample at a time. MathWorks documentation refers to the former as *frame-based processing* and the latter as *sample-based processing*. A *frame* is a collection of samples of data, sequential in time. To perform frame-based processing in MathWorks products, you must have a DSP System Toolbox license.

Historically, Simulink-family products that can perform frame-based processing propagate frame-based signals throughout a model. The frame status is an attribute of the signals in a model, just as data type and dimensions are attributes of a signal. The Simulink engine propagates the frame attribute of a signal with a frame bit, which can either be on or off. When the frame bit is on, Simulink interprets the signal as frame-based, and displays it as a double line, rather than as a single line.

Beginning in R2010b, MathWorks started to change the handling of frame-based processing significantly. In the future, signal attributes will not include frame status. Instead, individual blocks will control whether they treat data inputs as frames or as samples.

To transition to this new paradigm, blocks that can perform sample- and frame-based processing contain a new **Input processing** parameter that specifies the processing behavior. You can set **Input processing** to `Columns as channels (frame based)` or `Elements as channels (sample based)`. The third option, `Inherited (this choice will be removed - see release notes)`, is a temporary selection. This third option helps you migrate your existing models from the old paradigm to the new paradigm.

In R2011a, the following Embedded Coder blocks received a new **Input processing** parameter:

- C62X Real Forward Lattice All-Pole IIR
- C62X Complex FIR
- C62X General Real FIR
- C62X Real IIR
- C64X Real Forward Lattice All-Pole IIR

Compatibility Considerations

When you load an existing model in R2011a, blocks with the new **Input processing** parameter shows a setting of `Inherited (this choice will be removed - see release notes)`. This setting enables your existing models to work as expected until you upgrade them. Upgrade your models as soon as possible.

To upgrade your existing models, use the `slupdate` function. This function detects blocks that have **Input processing** set to `Inherited (this choice will be remove - see release notes)`. The function asks you whether to upgrade each block. If you select yes, the function detects the status of the frame bit on the input port of the block. If the frame bit is `1` (frames), the function sets the **Input processing** parameter to `Columns as channels (frame based)`. If the bit is `0` (samples), the function sets the parameter to `Elements as channels (sample based)`.

A future release will remove the frame bit and the `Inherited (this choice will be removed - see release notes)` option. At that time, if you have not updated the model, the software automatically sets the **Input**

processing parameter. The software uses the library default setting of the block to select either `Columns as channels (frame based)` or `Elements as channels (sample based)`. If the library default setting does not match the parameter setting in your model, your model will produce unexpected results. Additionally, after the removal of the frame bit, you will no longer be able to upgrade your models using the `slupdate` function. Therefore, upgrade your existing modes using `slupdate` as soon as possible.

New Support for Analog Devices Blackfin BF50x and BF51x Processors

You can now generate code for the following embedded processors when you use Embedded Coder software:

- BF504
- BF504F
- BF506F
- BF512
- BF514
- BF516
- BF518

Generate Optimized Fixed-Point Code for ARM Cortex-M3, Cortex-A8, and Cortex-A9 Processors

You can use new Target Function Libraries (TFLs) to generate efficient fixed-point code for the ARM Cortex-M3, Cortex-A8, and Cortex-A9 processors. These TFLs include GCC compiler extensions and intrinsic functions that optimize the code Embedded Coder generates for these processors.

Support for Versions 5.0.6 and 5.1.6 of Green Hills MULTI

Support for Green Hills MULTI software now includes versions 5.0.6 and 5.1.6. For additional information about supported versions, see the Support for Green Hills MULTI topic online.

Support for Texas Instruments Delfino C2834x Processors

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

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

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

- C2000 CAN Calibration Protocol
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Input
- C280x/C2802x/C2803x/C28x3x/c2834x GPIO Digital Output
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Receive
- C280x/C2802x/C2803x/C28x3x/C2834x I2C Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SCI Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Receive
- C280x/C2802x/C2803x/C28x3x/c2834x SPI Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x Software Interrupt Trigger
- C28x Watchdog
- C280x/C2803x/C28x3x/c2834x eCAN Receive
- C280x/C2803x/C28x3x/c2834x eCAN Transmit
- C280x/C2802x/C2803x/C28x3x/c2834x eCAP
- C280x/C2802x/C2803x/C28x3x/c2834x ePWM
- C280x/C2803x/C28x3x/c2834x eQEP

Ending Support for Altium TASKING in a Future Release

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

Ending Support for Freescale MPC5xx in a Future Release

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

Ending Support for Infineon C166 in a Future Release

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

Removed Methods and Arguments

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

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

Changes to ver Function Product Arguments

Compatibility Considerations: Yes

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

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

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

Compatibility Considerations

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

New and Enhanced Demos

The following demos have been added in R2011a:

Demo...	**Shows How You Can...**
`coderdemo_tfl`	Use target function libraries (TFLs) to replace operators and functions in code generated by MATLAB Coder.
`rtwdemo_code_coverage_script`	Generate model coverage and code coverage reports, and use these reports to compare model coverage and code coverage results for parts of a model.
`rtwdemo_pmsmfoc_script`	Perform system-level simulation and algorithmic code generation using Field-Oriented Control for a Permanent Magnet Synchronous Machine.

The following demos have been enhanced in R2011a:

Demo...	**Now...**
`vipstabilize_fixpt_beagleboard`	Uses the new Video Capture block to simulate or capture a video input signal in the Video Stabilization demo.

Check bug reports for issues and fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at `www.mathworks.com/support/bugreports/`. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

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

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

Search R2011a Bug Reports

Known Bugs for Incorrect Code Generation:
`www.mathworks.com/support/bugreports/?product=ALL&release=R2011a`
`&keyword=Incorrect+Code+Generation`

All Known Bugs for This Product:
`www.mathworks.com/support/bugreports/?release=R2011a&product=EC`