Real-Time Workshop® Release Notes

The Real-Time Workshop 6.3 Release Notes describe the changes introduced in the latest version of Real-Time Workshop®. The following topics are discussed in these Release Notes:

The Real-Time Workshop Release Notes also provide information about earlier versions of the product, in case you are upgrading from a version that was released prior to Version 6.3.

- Chapter 10, “Real-Time Workshop 4.1 Release Notes”
- Chapter 11, “Real-Time Workshop 4.0 Release Notes”

Contents

1 Real-Time Workshop 6.3 Release Notes

2 Real-Time Workshop 6.2.1 Release Notes

3 Real-Time Workshop 6.2 Release Notes

4 Real-Time Workshop 6.1 Release Notes

5 Real-Time Workshop 6.0 Release Notes

6 Real-Time Workshop 5.1.1 Release Notes

7 Real-Time Workshop 5.1 Release Notes

8 Real-Time Workshop 5.0.1 Release Notes

9 Real-Time Workshop 5.0 Release Notes

10 Real-Time Workshop 4.1 Release Notes

11 Real-Time Workshop 4.0 Release Notes

Real-Time Workshop 6.3 Release Notes

New Features and Enhancements

Real-Time Workshop Version 6.3 introduces the following new features and enhancements.

- "New rtw_precompile_libs Function" on page 1-2
- "Support for Subsystem Latch Enhancements" on page 1-3
- "Support for Variable Transport Delay Enhancements" on page 1-3
- "C++ Target Language Support for Real-Time Windows Target and External Mode " on page 1-4
- "Rapid Simulation Target Enhanced for Use with Distributed Computing Toolbox" on page 1-4
- "Simulink Model and MATLAB Desktop Window Interaction Enhanced" on page 1-4
- "Documentation Enhancements" on page 1-5

New rtw_precompile_libs Function

Real-Time Workshop 6.3 introduces a new M-file function, `rtw_precompile_libs`, which you can use to

- Precompile new or updated S-function libraries (MEX-files) for a model. By precompiling S-function libraries, you can optimize system builds. Once your precompiled libraries exist, Real-Time Workshop can omit library compilation from subsequent builds. For models that use numerous libraries, the time savings for build processing can be significant.
- Recompile precompiled libraries included as part of the Real-Time Workshop product, such as `rtwlib` or `dsplib`. You might consider doing this if you need to customize compiler settings for various platforms or environments.

For details on using `rtw_precompile_libs`, see "Precompiling S-Function Libraries" in the Real-Time Workshop documentation.

Support for Subsystem Latch Enhancements

This release of Real-Time Workshop supports Simulink® latch enhancements for triggered and function-call subsystems discussed in "Input Port Latching Enhancements" in the Simulink Release Notes.

- A renamed Inport block option is available for triggered subsystems. **Latch (buffer) input** was renamed to **Latch input by delaying outside signal** to better reflect the option's purpose.
- A new option, **Latch input by copying inside signal**, was added for the Inport block for use with function-call subsystems.

If you select **Latch input by copying inside signal** for a function-call subsystem, Real-Time Workshop

- Preserves latches in generated code regardless of any optimizations that might be set
- Places the code for latches at the start of a subsystem's output/update function

For more detail, also see the description of the Inport block.

Support for Variable Transport Delay Enhancements

Real-Time Workshop 6.3 supports new Simulink enhancements to the Variable Transport Delay block. Prior to R14SP3, the block performed a variable time delay function. The block has been enhanced to support both variable time and variable transport delays with a new parameter **Select delay type**:

- For instances of the block in existing models, **Select delay type** is set to `Variable time delay` to preserve the block's variable time delay behavior. In such cases, you can use the block as is, or consider changing the parameter settings for transport delay behavior.
- The Simulink Library Browser now offers a Variable Time Delay block and Variable Transport Delay block, which are instances of the original Variable Transport Delay block. Both blocks have the delay type parameter, which is preset depending on the type of block you include. In addition, for the Variable Time Delay block, you can select a parameter for handling zero

delays. For the Variable Transport Delay block, you can specify a fixed buffer size and absolute tolerance.

For more detail, see the descriptions of the Variable Time Delay and Variable Transport Delay blocks.

C++ Target Language Support for Real-Time Windows Target and External Mode

Real-Time Workshop 6.3 supports

- C++ code generation for Real-Time Windows target
- The use of external mode with executables it generates from C++ source files

For more information on C++ target language support, see "Support for C and C++ Code Generation" in the Real-Time Workshop documentation.

Rapid Simulation Target Enhanced for Use with Distributed Computing Toolbox

The Rapid Simulation (RSim) target has been enhanced such that RSim executables that specify a variable step solver do not check out a Simulink license when run by a worker executing a task created by the Distributed Computing Toolbox.

Simulink Model and MATLAB Desktop Window Interaction Enhanced

In R14SP3, the interaction between Simulink model and MATLAB® desktop windows during code generation has been enhanced such that the window layering and input focus during code generation on Windows systems now matches that of Linux systems.

Prior to R14SP3, if you had a Simulink model window on top of the MATLAB desktop window on a Windows system, the MATLAB desktop window would move on top of the model window when you generated code for that model. When code generation was complete, the MATLAB desktop window would retain input focus. This behavior intentionally differed from the behavior on Linux systems, which kept the model window on top.

Documentation Enhancements

The following areas of the Real-Time Workshop documentation have been corrected or improved:

- **Help** button on **Real-Time Workshop** pane and subpanes of the Configuration Parameters dialog box — displays help that is specific to the pane or subpane that is active
- Example index — expanded
- Feature limitations — moved to a new section, "Limitations" on page 1-7 in the Release Notes
- Model reference tutorial
- "Code Generation and the Build Process" — reorganized to reflect workflow and make key topics more accessible
- "Controlling the Location and Names of Libraries During the Build Process" — added as a new topic
- "Tunable Expressions in Masked Subsystems"
- "Profiling Generated Code" — added as a new topic
- "Reusable Code and Referenced Models"
- "Sharing Utility Functions"
- "Data Transfer Assumptions" for rate transitions
- "Writing Noninlined S-Functions"
- "Build Support for S-Functions"
- "Checksums and the S-Function Target" — added as a new topic
- "Specifying a New Signal Data File for a From File Block" when running a rapid simulation
- "Generating ASAP2 and C-API Files" — added as a new topic

- "Simulink Block Support" — new reference appendix listing Real-Time Workshop and Real-Time Workshop Embedded Coder block support for blocks available in Simulink
- Target Language Compiler documentation

Limitations

C++ Target Language Limitations

- Microsoft Visual C/C++ and GNU C/C++ have been fully tested with Real-Time Workshop Version 6.3 and are fully supported on 32-bit Windows and Linux platforms. However, Version 6.3 provides Beta C++ support only for the Watcom, Borland®, and Intel® C/C++ compilers. These compilers have not yet been fully evaluated for C++ compatibility with MathWorks products.
- Real-Time Workshop provides Beta support for C++ code generation for all blockset products. C++ code generation for the blockset products has not yet been fully evaluated.
- Real-Time Workshop does not support C++ code generation for the following:

  Embedded Target for Infineon C166® Microcontrollers
  Embedded Target for Motorola® MPC555
  Embedded Target for Motorola® HC12
  Embedded Target for OSEK/VDX®
  Embedded Target for TI C2000™ DSP

Embedded Target for TI C6000™ DSP
SimDriveline
SimMechanics
SimPowerSystems
xPC Target

- When using the model reference feature, you cannot generate C code for the parent model and C++ code for models that refer to the parent model. However, you can generate C or C++ for both the parent and referring models, or C++ code for the parent model and C code for referring models.
- The following Real-Time Workshop Embedded Coder dialog box fields currently do not accept the `.cpp` extension. However, a `.cpp` file will be generated if you specify a filename without an extension in these fields, with C++ selected as the target language for your generated code.
  - **Data definition filename** field on the **Data Placement** pane of the Configuration Parameters dialog box
  - **Definition file** field for an **mpt data object** in the Model Explorer

  These restrictions on specifying `.cpp` will be removed in a future release.

Tunable Expression Limitations

Currently, there are certain limitations on the use of tunable variables in expressions. When an expression described below as not supported is encountered during code generation, a warning is issued and a nontunable expression is generated in the code. The limitations on tunable expressions are

- Complex expressions are not supported, except where the expression is simply the name of a complex variable.
- The use of certain operators or functions in expressions containing tunable operands is restricted. Restrictions are applied to four categories of operators or functions, classified in the following table:

<table>
<tr><th>Category</th><th>Operators or Functions</th></tr>
<tr><td>1</td><td><code>+ - .* ./ < > <= >= == ~= & |</code></td></tr>
<tr><td>2</td><td><code>* /</code></td></tr>
</table>

Category	Operators or Functions
3	`abs`, `acos`, `asin`, `atan`, `atan2`, `boolean`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `int8`, `int16`, `int32`, `log`, `log10`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `uint8`, `uint16`, `uint32`
4	`: .^ ^ [] {} .  \ .\ ' .'  ; ,`

The rules applying to each category are as follows:

- Category 1 is unrestricted. These operators can be used in tunable expressions with any combination of scalar or vector operands.
- Category 2 operators can be used in tunable expressions where at least one operand is a scalar. That is, scalar/scalar and scalar/matrix operand combinations are supported, but not matrix/matrix.
- Category 3 lists all functions that support tunable arguments. Tunable arguments passed to these functions retain their tunability. Tunable arguments passed to any other functions lose their tunability.
- Category 4 operators are not supported.

Note The “dot” (structure membership) operator is not supported. This means that expressions that include a structure member are not tunable.

- Expressions that include variables that were declared or modified in mask initialization code are *not* tunable.
- The Fcn block does not support tunable expressions in code generation.
- Model workspace parameters can take on only the `Auto` storage class, and thus are not tunable. To tune parameters in referenced models globally, declare `Simulink.Parameter` objects for them in the MATLAB workspace (not in model workspaces).

Limitations on Specifying Data Types in the Workspace Explicitly

When you explicitly specify a data type other than `double` in the workspace, blocks typecast the parameter to the appropriate data type. This is an issue for blocks that use pointer access for their parameters. Blocks cannot use pointer access if they need to typecast the parameter before using it (because of a data type mismatch). Two possible solutions to this problem are

- Remove the explicit data type specification in the workspace for parameters used in such blocks.
- Modify the block so that it uses the parameter with the same data type as specified in the workspace. For example, the Lookup Table block uses the data types of its input signal to determine the data type that it uses to access the `X-breakpoint` parameter. You can prevent the block from typecasting the run-time parameter by converting the input signal to the data type used for `X-breakpoints` in the workspace. (Similarly, the output signal is used to determine the data types used to access the lookup table's `Y` data.)

Code Reuse Limitations

Real-Time Workshop uses a checksum to determine whether subsystems are identical. You cannot reuse subsystem code if instances of a subsystem differ in the following ways:

- A port used by multiple instances of a subsystem has different sample times, data types, complexity, frame status, or dimensions across the instances
- Subsystems contain identical blocks with different names or parameter settings

Some of these situations can arise even when subsystems are copied and pasted within or between models or are manually constructed to be identical. If you select `Reusable function` and Real-Time Workshop determines that code for a subsystem cannot be reused, it generates a separate function that is not reused. The code generation report can show that the separate function is reusable, even if it is used by only one subsystem. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, you should choose `Auto` for the **RTW system code** option.

The presence of certain blocks in a subsystem can also prevent its code from being reused. These are

- Scope blocks (with data logging enabled)
- S-Function blocks that fail to meet certain criteria
- To File blocks
- To Workspace blocks

Model Referencing Limitations

This section summarizes major limitations on the use of model referencing with some features of Real-Time Workshop and products based on Real-Time Workshop. For example, models must meet certain conditions to reference other models or be referenced by other models. See “Model Referencing Limitations” in the Simulink Release Notes for a more complete list of model reference limitations.

The following limitations are specific to code generation:

- When using the model reference feature, you cannot generate C code for the top model and C++ code for referenced models. However, you can generate C or C++ for both the top and referenced models or C++ code for the top model and C code for referenced models.
- When using the data logging feature, note that
  - To Workspace and Scope blocks in models referenced by a top model do not log data when you run code generated from the top model.
  - A top model can perform data logging to MAT-files whether or not it refers to other models. However, code generated for referenced models does not log data to MAT-files regardless of the target specified. If data logging is enable for a referenced model, Real-Time Workshop disables the option during code generation and reenables it after the build is complete.
- The S-function target and GRT `malloc` target do not support model referencing.

- You cannot build a subsystem module by right-clicking a subsystem if the subsystem contains Model blocks unless the model is configured to use an ERT target.
- Real-Time Workshop cannot generate stand-alone executables for models that refer to models that include noninlined S-functions.
- A referenced model cannot use noninlined S-functions generated by Real-Time Workshop.
- Configuration parameters of a top model and its reference models must meet specific conditions. For details, see “Possible Incompatibilities Between Top and Referenced Models” in the Real-Time Workshop documentation.
- You must clear the **Load initial state** option on the **Data Import/Export** pane of the Configuration Parameters dialog box when building a target for a referenced model. However, you can select this option for the top model.
- If you generate code for a model’s atomic subsystems as reusable functions, the functions can have inputs or outputs connected to a referenced model’s root Inport or Outport blocks, however, they can affect code reuse. For details, see “Reusable Code and Referenced Models” in the Real-Time Workshop documentation.
- If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See “Supporting Model Referencing” in the Real-Time Workshop Embedded Coder documentation.

External Mode Limitations

In general, you cannot change a parameter if doing so results in a change in the structure of the model. For example, you cannot change

- The number of states, inputs, or outputs of any block
- The sample time or the number of sample times
- The integration algorithm for continuous systems
- The name of the model or of any block
- The parameters to the Fcn block

If you cause any of these changes to the block diagram, then you must rebuild the program with newly generated code.

However, you can change parameters in transfer function and state space representation blocks in specific ways:

- The parameters (numerator and denominator polynomials) for the Transfer Fcn (continuous and discrete) and Discrete Filter blocks can be changed (as long as the number of states does not change).
- Zero entries in the State-Space and Zero Pole (both continuous and discrete) blocks in the user-specified or computed parameters (that is, the A, B, C, and D matrices obtained by a zero-pole to state-space transformation) cannot be changed once external simulation is started.
- In the State-Space block, if you specify the matrices in the controllable canonical realization, then all changes to the A, B, C, D matrices that preserve this realization and the dimensions of the matrices are allowed.

Note Opening a dialog box for a source block causes Simulink to pause. While Simulink is paused, you can edit the parameter values. You must close the dialog box to have the changes take effect and allow Simulink to continue.

If the Simulink block diagram does not match the external program, Simulink displays an error informing you that the checksums do not match (that is, the model has changed since you generated code). This means that you must rebuild the program from the new block diagram (or reload the correct one) to use external mode.

If the external program is not running, Simulink displays an error informing you that it cannot connect to the external program.

Noninlined S-Function Parameter Type Limitations

Parameters to S-functions can be of the following types only:

- Double precision
- Characters in scalars, vectors, or 2-D matrices

For more flexibility in the type of parameters you can supply to S-functions or the operations in the S-function, inline your S-function and consider using an `mdlRTW` S-function routine.

S-Function Target Limitations

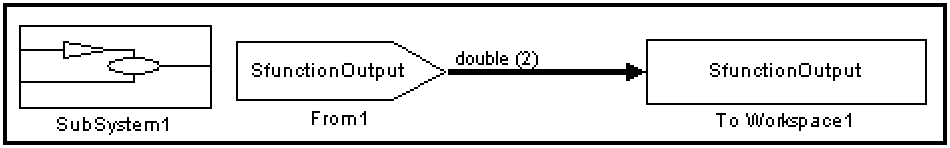
- "Goto and From Block Limitations" on page 1-14
- "Model Reference and Updating Limitations" on page 1-16
- "Unsupported Blocks" on page 1-16

Goto and From Block Limitations

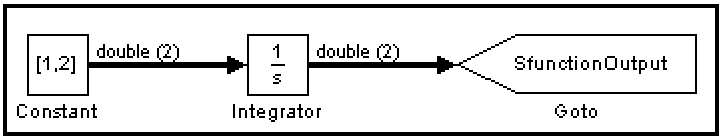
When using the S-function target, Real-Time Workshop restricts I/O to correspond to the root model's Inport and Outport blocks (or the Inport and Outport blocks of the Subsystem block from which the S-function target was generated). No code is generated for Goto or From blocks.

To work around this restriction, you should create your model and subsystem with the required Inport and Outport blocks, instead of using Goto and From blocks to pass data between the root model and subsystem. In the model that incorporates the generated S-function, you would then add needed Goto and From blocks.

As an example of this restriction, consider the model shown below, and its subsystem, `Subsystem1`, shown in the subsequent figure. The Goto block in `Subsystem1`, which has global visibility, passes its input to the From block in the root model.

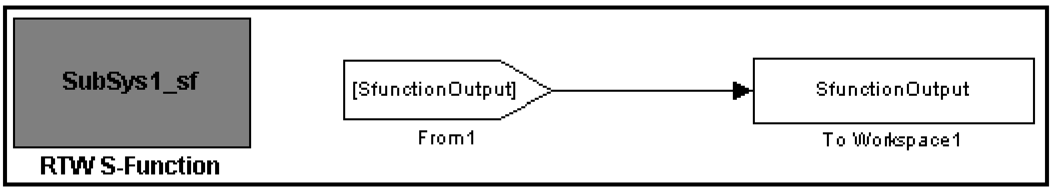


Root Model with From Block



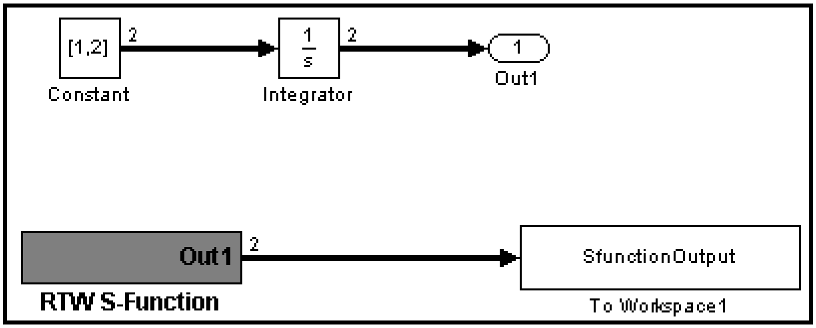
Subsystem1 with Goto Block

If `SubSystem1` is built as an S-Function using the S-Function target, and plugged into the original model (as shown in the following figure), a warning is issued when the model is run because the generated S-function does not implement the Goto block.



Generated S-Function Replaces Subsystem1

A workaround is shown in the next figure. A conventional Outport block is used in `Subsystem1`. When the generated S-function is plugged into the root model, its output is connected to the To Workspace block.



Use of Outport in Generated S-Function

Model Reference and Updating Limitations

The following limitations apply to building and regenerating S-function targets:

- You cannot build models that contain Model blocks using the Real-Time Workshop S-function target. This also means that you cannot build a subsystem module by right-clicking (or by using **Tools > Real-Time Workshop > Build subsystem**) if the subsystem contains Model blocks. This restriction applies only to GRT S-functions, not to ERT S-functions.
- If you modify the model that generated an S-Function block, Real-Time Workshop does not automatically rebuild models containing the generated S-Function block. This is in contrast to the practice of automatically rebuilding models referenced by Model blocks when they are modified (depending on the Model Reference **Rebuild options** configuration setting).
- Handwritten S-functions without corresponding TLC files must contain exception-free code. For more information on exception-free code, see "Exception Free Code" in the Writing S-Functions Simulink documentation.

Unsupported Blocks

The S-function format does not support the following built-in blocks:

- MATLAB Fcn block
- S-Function blocks containing any of the following:

  - M-file S-functions (unless you supply a TLC file for C code generation)
  - Fortran S-functions (unless you supply a TLC file for C code generation)
  - C/C++ MEX S-functions that call into MATLAB
- Scope block
- To Workspace block

Rapid Simulation Target Limitations

The rapid simulation (RSim) target is subject to the following limitations:

- The RSim target does not support algebraic loops.
- The RSim target does not support MATLAB function blocks.
- The RSim target does not support noninlined M-file, Fortran, or Ada S-functions.
- If an RSim build includes referenced models (by using Model blocks), these models must be set up to use fixed-step solvers for code to be generated for them. The top model, however, can use a variable-step solver as long as all blocks in the referenced models are discrete.
- In certain cases, changing block parameters can result in structural changes to your model that change the model checksum. An example of such a change would be changing the number of delays in a DSP simulation. In such cases, you must regenerate the code for the model.
- Variable-step solver support for RSim is not available on Windows platforms when you use the following compilers:
  - Watcom C/C++ compiler
  - Borland C/C++ compiler

C-API Limitations

The C-API feature has the following limitations.

- The following code formats are not supported:
  - S-function
  - Simulink Accelerator

- For ERT-based targets, the C-API requires that support for floating-point code be enabled.
- The following signals are not supported:
  - External inputs
  - External outputs
  - Local block outputs
- Parameters local to Stateflow are not supported.
- The following custom storage class objects are not supported:
  - Objects without the package `csc_registration` file are not supported.
  - `BitPackBoolean` objects, grouped custom storage classes, and objects defined by using macro are not supported.
- Customized data placement is disabled when you are using the C-API. The interface looks for global data declaration in *`model`*`.h` and *`model`*`_private.h`. Declarations placed in any other file by customized data placement result in code that does not compile.

Note Custom Storage Class objects take effect in code generation only if you use the ERT target and clear the **Ignore custom storage classes** check box on the Configuration Parameters dialog box.

Simulink Block Limitations

For information on limitations on the use of Simulink blocks for code generation, see "Simulink Block Support" in the Real-Time Workshop documentation. A table lists blocks available in Simulink and provides caveats, limitations, and suggestions regarding their use with Real-Time Workshop and Real-Time Workshop Embedded Coder.

Major Bug Fixes

To view major bug fixes made in Release 14SP3 for Real-Time Workshop, use the Bug Reports interface on the MathWorks Web site.

Note If you are not already logged in to Access Login, when you link to the Bug Reports interface (see below), you will be prompted to log in or create an Access Login account.

After you are logged in, use this Bug Fixes link. You will see the bug report for Real-Time Workshop. The report is sorted with fixed bugs listed first, and then open bugs.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

Bug fixes made before Release 14SP2 are not included in the Bug Reports interface. For earlier bug fixes, see

- R14SP1 Major Bug Fixes
- R14 Major Bug Fixes
- R13SP1 Major Bug Fixes
- R13+ Major Bug Fixes
- R13 Major Bug Fixes
- R12SP1 Bug Fixes

Upgrading from an Earlier Release

If you are upgrading from an earlier release, you should review the following notes:

- "Customizations to Built-In Blocks" on page 1-20
- "Use slbuild Instead of rtwgen" on page 1-20
- "CustomStorageClass and StorageClass Properties Initialized Differently" on page 1-20
- "rem Function No Longer Supports Tunable Arguments" on page 1-21
- "Hardware Configuration for Pre-Version 6 Models" on page 1-21

Customizations to Built-In Blocks

The MathWorks recommends that you not customize built-in blocks provided as part of the Simulink product even though the capability exists to do so. Such customizations might not be applied during the code generation process and can lead to unpredictable results.

Use slbuild Instead of rtwgen

The Target Language Compiler documentation for R14SP2 and earlier recommends using the `rtwgen` and `tlc` commands together to create targets and generate code. The `rtwgen` command is not intended for direct use, and upgrading Real-Time Workshop may cause code that uses the command to fail. Existing code should change to use `slbuild` rather than `rtwgen`, and new code should use `slbuild` exclusively. The syntax for `slbuild` is

```
slbuild('model'[,'TargetType'])
```

Use of the `tlc` command is unaffected by this change.

CustomStorageClass and StorageClass Properties Initialized Differently

In Release 14, Real-Time Workshop merged functionality of custom storage classes into the standard `Simulink.Parameter` and `Simulink.Signal` classes. As a result, when you instantiate the `Simulink.CustomParameter` and `Simulink.CustomSignal` classes, the `CustomStorageClass` and `StorageClass` properties do not get initialized the same way they did in Release 13.

In Release 13, the properties were initialized as

```
CustomStorageClass = 'BitField' (1st item on the list)
 StorageClass = 'Custom'
```

Starting in Release 14, the properties are initialized as

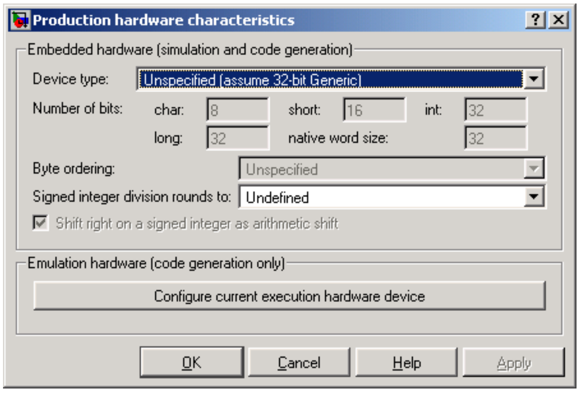
```
CustomStorageClass = 'Default' (1st item on the list)
StorageClass = 'Auto' (custom storage class is ignored)
```

rem Function No Longer Supports Tunable Arguments

As of R14SP3, the `rem` function no longer supports tunable parameters when used with Real-Time Workshop. If you use tunable parameters with this function, Real-Time Workshop inlines the equivalent numeric value into the generated code in place of the tunable expression.

Hardware Configuration for Pre-Version 6 Models

When you open a preexisting model that has not been saved using the current version of Simulink, and select **Hardware** in the Configuration Parameters dialog box, the following set of controls appears:



All but one of the parameters below the **Device type** menu are grayed out. This is because these characteristics have been preset for the default target (`32-bit Generic`), as well as for several dozen known target processors that you can select from that menu.

In the event that none of the choices listed in the **Device Type** drop-down menu is appropriate for your intended hardware target, you can select `Custom`, and then set values for the hardware characteristics. Selecting any other option disables them. The hardware characteristics that you can specify are

- **Number of bits** — Text fields that specify the number of bits used to represent types **char**, **short**, **int**, and **long**. The values specified should be consistent with the word sizes as defined in the compiler's `limits.h` header file.
- **Byte ordering** — Specifies whether the target hardware uses `Big Endian` (most significant byte first) or `Little Endian` (least significant byte first) byte ordering. If left as `Unspecified`, Real-Time Workshop generates code to determine the endianness of the target; this is the least efficient option.
- **Shift right on a signed integer as arithmetic shift** — ANSI C leaves the behavior of right shifts on negative integers as implementation dependent. Use this control to specify how Real-Time Workshop implements right shifts on signed integers in generated code.

  The option is selected by default. If your C or C++ compiler handles right shifts as arithmetic shifts, this is the preferred setting.

  - When the option is selected, Real Time Workshop generates simple efficient code whenever the Simulink model performs arithmetic shifts on signed integers.
  - When the option is unselected, Real Time Workshop generates fully portable but less efficient code to implement right arithmetic shifts.

Known Software and Documentation Problems

To view important open bugs in Release 14SP3 for Real-Time Workshop, use the Bug Reports interface on the MathWorks Web site.

Note If you are not already logged in to Access Login, when you link to the Bug Reports interface (see below), you will be prompted to log in or create an Access Login account.

After you are logged in, use this Bug Reports link. You will see the bug report for Real-Time Workshop. The report is sorted with fixed bugs listed first, and then open bugs. You can select the **Status** column to list the open bugs first.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

2

Real-Time Workshop 6.2.1 Release Notes

Major Bug Fixes

To view major bug fixes made in Release 14SP2+ for Real-Time Workshop, use the Bug Reports interface on the MathWorks Web site.

Note If you are not already logged into Access Login, when you link to the Bug Reports interface (see below), you will be prompted to log in or create an Access Login account.

After you are logged in, use this Bug Fixes link. You will see the bug report for Real-Time Workshop. The report is sorted with fixed bugs listed first, and then open bugs.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

Bug fixes made before Release 14SP2+ are not included in the Bug Reports interface. For earlier bug fixes, see

- R14SP1 Major Bug Fixes
- R14 Major Bug Fixes
- R13SP1 Major Bug Fixes
- R13+ Major Bug Fixes
- R13 Major Bug Fixes
- R12SP1 Bug Fixes

Real-Time Workshop 6.2 Release Notes

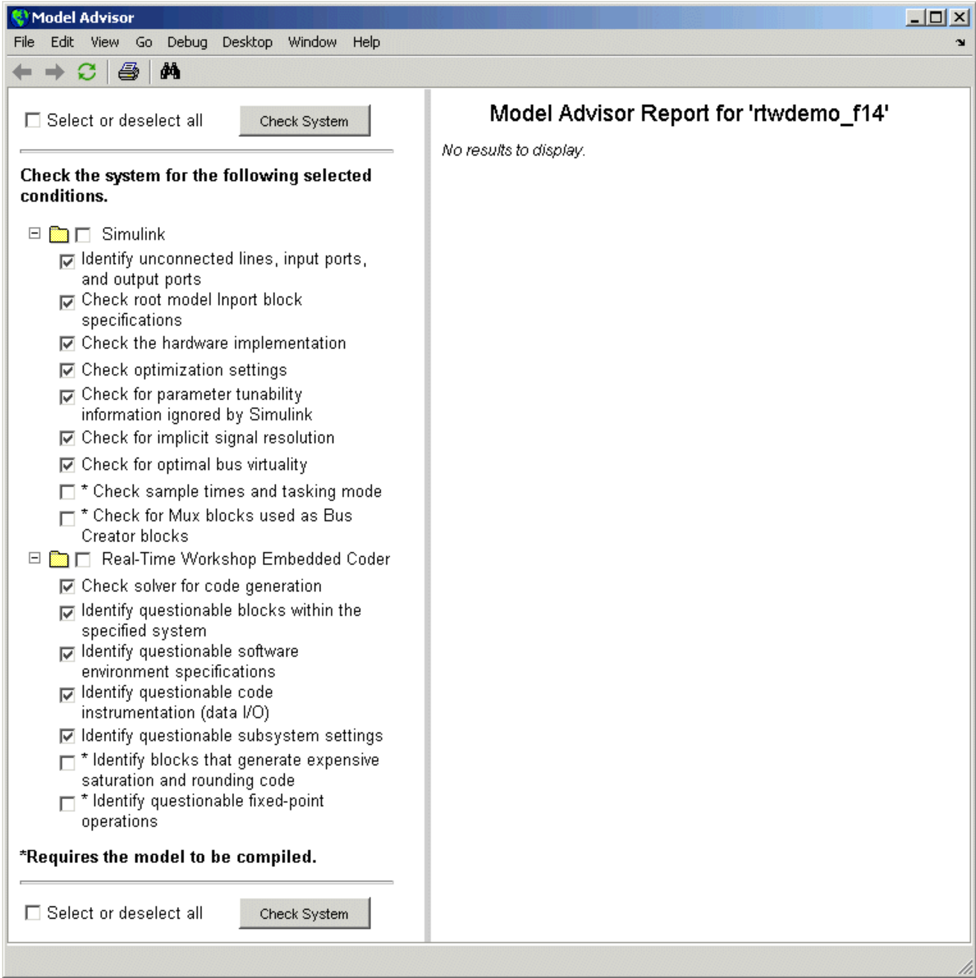
New Features and Enhancements

Real-Time Workshop Version 6.2 introduces the following new features and enhancements.

Model Advisor Enhancements

The Model Advisor analyzes Simulink models for optimal use of Simulink for simulation and code generation. You can customize the analysis and resulting report by selecting the checks that you want the Model Advisor to perform. Real-Time Workshop 6.2 enhances the Model Advisor by adding several new checks and grouping checks based on their application for simulation or code generation.

The Model Advisor dialog box now appears as follows:



Rate Transition Block Enhancements

The Rate Transition block has been enhanced to support:

- Automatic insertion for transitions to or from asynchronous tasks. If you select the **Automatically handle data transfers between tasks** on the

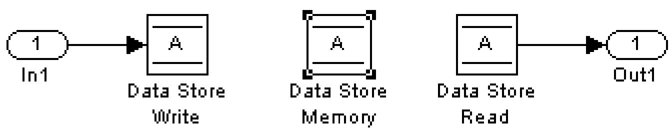
Solvers pane of the Configuration Parameters dialog, Simulink detects rate transitions and inserts Rate Transition blocks automatically to handle them for asynchronous and periodic tasks. Prior to Version 6.2, automatic block insertion for asynchronous tasks was not supported. For details, see “Rate Transition Block Options”.

- Automatic insertion for single-tasking execution mode. If you select the **Automatically handle data transfers between tasks**, Simulink detects rate transitions inserts Rate Transition blocks automatically for models that execute in single-tasking or multitasking mode. Prior to Version 6.2, automatic block insertion for single-tasking execution mode was not supported. For details, see “Rate Transitions and Asynchronous Blocks”.
- Asynchronous rates when no priority is specified. You can set the block to one of two modes—unprotected or data integrity with no determinism. Prior to Version 6.2, the Rate Transition block did not ensure data integrity for asynchronous rates when the priority was not set. For details, see “Rate Transitions and Asynchronous Blocks”.

Data Store Read Block Enhancement

The code that Real-Time Workshop generates for the Data Store Read block has been optimized. Prior to this release, the code generated for this block would copy the value of the block to a temporary variable. This release of Real-Time Workshop eliminates the use of the temporary variable, if possible.

Consider the following model:



A section of the code generated for this model, using an earlier version of Real-Time Workshop would appear as follows:

```
/* local block i/o variables */

real_T rtb_DataStoreRead;

/* DataStoreWrite: '/Data Store Write' incorporates:
```

```
 *   Inport: '/In1'
 */
mdsm_opt_DWork.A = mdsm_opt_U.In1;

/* DataStoreRead: '/Data Store Read' */
rtb_DataStoreRead = mdsm_opt_DWork.A;

/* Outport: '/Out1' */
mdsm_opt_Y.Out1 = rtb_DataStoreRead;
```

Note the value of `mdsm_opt_DWork.A` is stored in the temporary variable `rtb_DataStoreRead`.

The following code fragment shows the comparable section of code generated by this release of Real-Time Workshop. The temporary variable `rtb_DataStoreRead` is no longer used.

```
/* DataStoreWrite: '/Data Store Write' incorporates:
 *  Inport: '/In1'
 */
mdsm_opt_DWork.A = mdsm_opt_U.In1;

/* Outport: '/Out1' incorporates:
 *  DataStoreRead: '/Data Store Read'
 */
mdsm_opt_Y.Out1 = mdsm_opt_DWork.A;
```

C++ Target Language Support

Real-Time Workshop 6.2 introduces support for generating C++ code. The primary use for this feature is to facilitate integration of generated code with legacy or custom user code written in C++.

The following sections discuss

- “Configuring Your Compiler” on page 3-6
- “Selecting Target Language in Configuration Parameter Dialog” on page 3-6
- “Integrating C and C++ Code” on page 3-7
- “C++ Target Language Limitations” on page 3-8

For a demo, enter `sfcndemo_cppcount` in the MATLAB Command Window. For a Stateflow example, enter `sf_cpp`.

Configuring Your Compiler

Before you use the new C++ target language support, you might need to configure Real-Time Workshop to use the appropriate compiler. For example, on Windows the default compiler is the `lcc` C compiler shipped with MATLAB. If you do not configure Real-Time Workshop to use a C++ compiler before you select the C++ language option, the following build error message appears:

```
The specified Real-Time Workshop target is configured to generate
C++, but the C-only compiler, LCC, is the default compiler.  To
specify a C++ compiler, enter 'mex -setup' at the command prompt.
To generate C code, click (Open) to open the Configuration
Parameters dialog and set the target language to C.
```

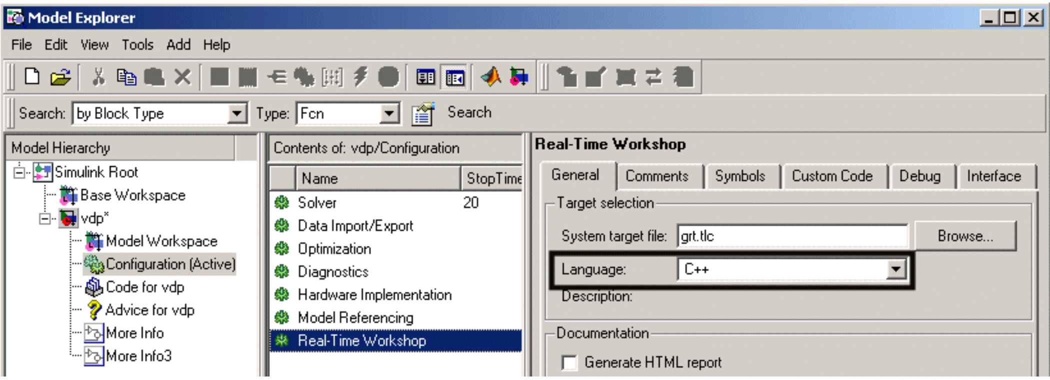
To configure the compiler, use the `mex -setup` function. Alternatively, you can modify your target's template make file (TMF) to include an environment variable provided by the compiler vendor that specifies the correct compiler.

See "Choosing and Configuring a Compiler" in the Real-Time Workshop documentation for more information.

Selecting Target Language in Configuration Parameter Dialog

To select the target language for the code Real-Time Workshop generates,

1. Open the **Configuration Parameters** dialog in the Model Explorer or from the **Simulation** menu.
2. Open the general **Real-Time Workshop** pane.
3. In the **Target selection** section, select `C` or `C++` from the **Language** menu.



Real-Time Workshop generates `.c` or `.cpp` files, depending on your selection, and places the files in your build directory.

Integrating C and C++ Code

Real-Time Workshop includes a variety of mechanisms for integrating generated code with legacy or custom code. A summary of these mechanism is available in "Integrating Legacy and Custom Code".

If you need to integrate legacy or custom C code with generated C++ code or vice versa, you must modify your legacy or custom code to be language compatible with the generated code. Options for making the code language compatible include

- Writing or rewriting the legacy or custom code in the same language as the generated code.
- If the generated code is in C++ and your legacy or custom code is in C, for each C function, create a header file that prototypes the function, using the following format:

```
#ifdef __cplusplus
extern "C" {
#endif
int my_c_function_wrapper();
#ifdef __cplusplus
}
#endif
```

The prototype serves as a function wrapper. The value __cplusplus is defined if your compiler supports C++ code. The linkage specification extern "C" specifies C linkage with no name mangling.

- If the generated code is in C and your legacy or custom code is in C++, include an extern "C" linkage specification in each .cpp file. For example, the following shows the a portion of the C++ code in the file my_func.cpp:

```
extern "C" {

int my_cpp_function();
{ .
  .
  .}
}
```

C++ Target Language Limitations

- Microsoft Visual C/C++ and GNU C/C++ have been fully tested and are fully supported on 32–bit Windows and Linux platforms. However, Version 6.2 provides Beta C++ support only for the Watcom, Borland®, and Intel® C/C++ compilers. These compilers have not yet been fully evaluated for compatibility with MathWorks products.
- Real-Time Workshop provides Beta support for C++ code generation for all blockset products. C++ code generation for the blockset products has not yet been fully evaluated.
- Real-Time Workshop does not support C++ code generation for the following:

  Embedded Target for Infineon C166® Microcontrollers
  Embedded Target for Motorola® MPC555
  Embedded Target for Motorola® HC12
  Embedded Target for OSEK/VDX®
  Embedded Target for TI C2000™ DSP
  Embedded Target for TI C6000™ DSP
  Real-Time Windows Target
  SimDriveline
  SimMechanics
  SimPowerSystems

xPC Target

- Real-Time Workshop does not support the use of external mode with executables it generates from C++ source files.
- When using the Model Reference feature, you cannot generate C code for the parent model and C++ code for models that refer to the parent model. However, you can generate C or C++ for both the parent and referring models, or C++ code for the parent model and C code for referring models.
- The following Real-Time Workshop Embedded Coder dialog box fields currently do not accept the `.cpp` extension. However, a `.cpp` file will be generated if you specify a filename without an extension in these fields, with C++ selected as the target language for your generated code.
  - **Data definition filename** field on the **Data Placement** pane of the Configuration Parameters dialog box
  - **Definition file** field for an **mpt data object** in the Model Explorer

  These restrictions on specifying `.cpp` will be removed in a future release.

Support for Open Watcom 1.3 Compiler

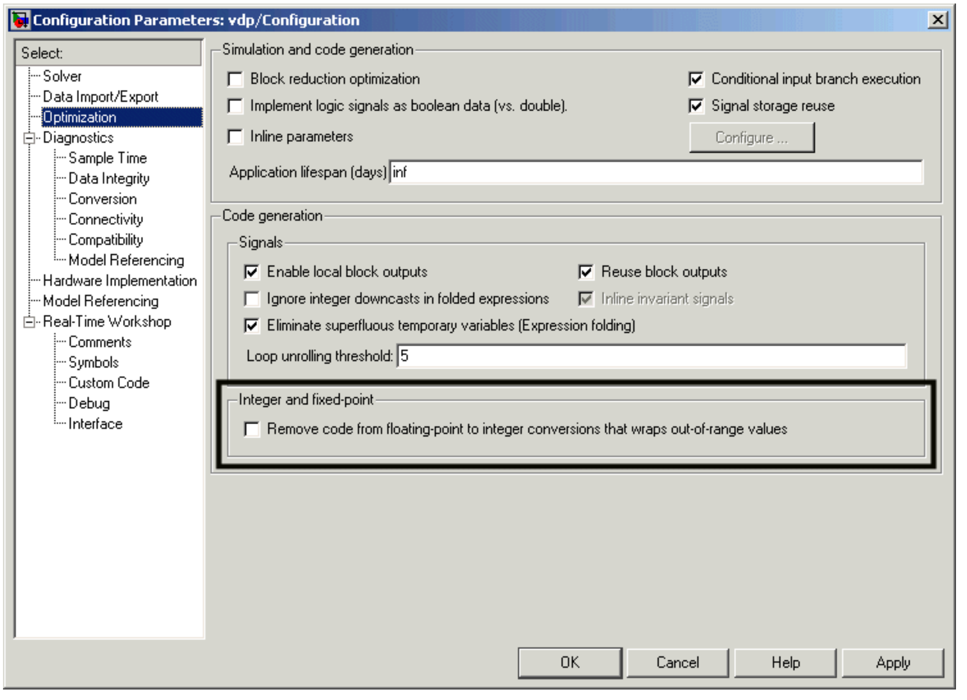
Real-Time Workshop 6.2 provides Beta support for the Open Watcom 1.3 compiler. The compiler has not yet been fully evaluated for compatibility with MathWorks products. However, the support files necessary for you to use the compiler with MATLAB and the MATLAB Compiler are available. To configure the compiler, use the `mex -setup` function. Full support will be available in a future release.

New Configuration Option for Optimizing Floating-Point to Integer Data Type Conversions

A new option has been added to the **Optimization** pane of the Configuration Parameters dialog box that you can use to increase the efficiency of generated code that represents floating-point to integer or fixed-point data type conversions. The option removes code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur. This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.

Note Enabling this option affects code generation results only for out-of-range values and hence cannot cause code generation results to differ from simulation results for in-range values.

The new option, **Remove code from floating-point to integer conversions that wraps out-of-range values**, appears in the **Integer and fixed-point** section of the dialog box pane, as shown below.



Consider using this option if code efficiency is critical to your application and the following conditions are true for at least one block in the model.

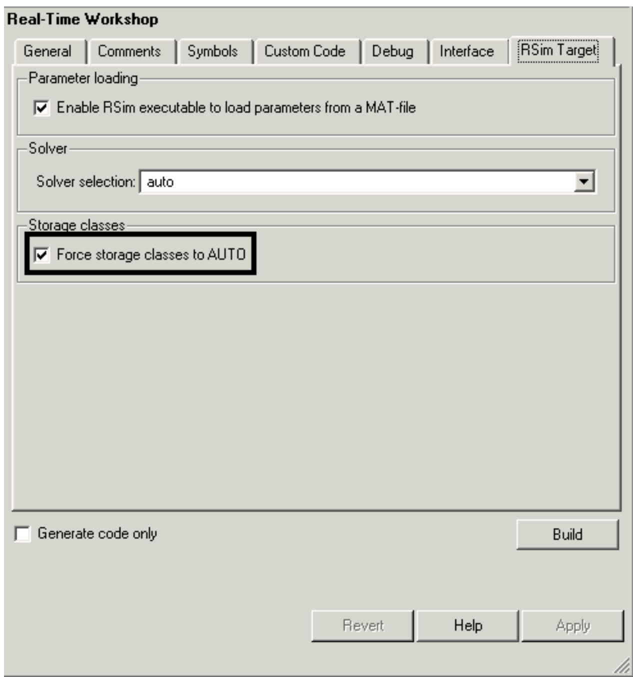
- Computing the block's outputs or parameters involves converting floating-point data to integer or fixed-point data
- The block's **Saturate on integer overflow** option is disabled

Task Priority Block Parameters Renamed for Consistency

The **Effective priorities** parameter for the Async Interrupt block and **Task priority** parameter for the Task Sync block are renamed **Simulink task priority**. In both cases, the Rate Transition block uses the parameter to generate the appropriate high-to-low or low-to-high priority transition code.

New RSim Target Configuration Option

A new option, **Force storage classes to AUTO**, has been added to the **Real-Time Workshop/RSim Target** pane of the Configuration Parameters dialog box. The option is on by default and forces all storage classes to `Auto`. If your application requires the use of other storage classes, such a `ExportedGlobal` or `ImportedExtern`, turn this option off. The new option appears in the **Storage Classes** section as shown below.



LibManageAsyncCounter Function Added to asynclib.tlc Library

The function `LibManageAsyncCounter` has been added to the `asynclib.tlc` TLC library. This function determines whether an asynchronous task needs a counter and manages its own timer.

Enhanced Documentation on Integrating Legacy and Custom Code with Generated Code

Documentation on integrating legacy and custom code with generated code has been enhanced.

- A new section, "Integrating Legacy and Custom Code", summarizes the mechanisms available for integrating code generated by Real-Time Workshop into an existing code base or integrating existing code into code generated by Real-Time Workshop. In the later scenario, integration can be either block based or model based. The new summary can help you evaluate and choose a mechanism that best meets your application requirements and directs you to other areas of the documentation for implementation details.
- The section "Using the rtwmakecfg.m API" discusses new fields in the `rtwmakecfg.m` API that support the Real-Time Workshop build process for S-functions.
- A new section, "Build Support for S-Functions", discusses the different ways of adding build information to the Real-Time Workshop build process.

Documentation Improvements

The following areas of the Real-Time Workshop documentation have been corrected or improved:

- Integrating custom and legacy code
- References to and screen captures showing new and modified Configuration Parameter dialog box options
- Descriptions of `MaxStackSize` and `MaxStackVariableSize` variables
- Limitations on tunable expressions
- Limitation on Stateflow outputs (removed)

- Symbolic naming conventions for signals in generated code as documented in “Working with Data Structures”
- Parameter tuning using MATLAB commands
- How to avoid parameter configuration conflicts related to storage classes
- Example for user-defined block state names
- Parameter configuration quick reference diagram (was missing from HTML output)
- Data type considerations for tunable workspace parameters
- Definitions of top model and reference model in the context of model referencing
- Deletion of user `*.c` files from the Real-Time Workshop build directory
- Conditions that need to be met for a block to be considered for dead code elimination
- Writing S-functions that specify sample time inheritance
- Use of `ssSetNeedAbsoluteTime` or `ssSetNeedElapseTime` in S-functions for accessing timers
- Optimizing with expression folding
- References to the Data Object Wizard (DOW) in the context of using ASAP2
- C API for S-Functions
- External mode parameter descriptions

Major Bug Fixes

To view major bug fixes made in Release 14SP2 for Real-Time Workshop, use the Bug Reports interface on the MathWorks Web site.

Note If you are not already logged into Access Login, when you link to the Bug Reports interface (see below), you will be prompted to log in or create an Access Login account.

After you are logged in, use this Bug Fixes link. You will see the bug report for Real-Time Workshop. The report is sorted with fixed bugs listed first, and then open bugs.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

Bug fixes made before Release 14SP2 are not included in the Bug Reports interface. For earlier bug fixes, see

- R14SP1 Major Bug Fixes
- R14 Major Bug Fixes
- R13SP1 Major Bug Fixes
- R13+ Major Bug Fixes
- R13 Major Bug Fixes
- R12SP1 Bug Fixes

Real-Time Workshop 6.1 Release Notes

Changes from the Previous Release

The behavior of source block dialog has changed. Note that opening a dialog for a source block causes Simulink® to pause. While Simulink is paused, you can edit the parameter values. You must close the dialog to have the changes take effect and allow Simulink to continue.

Major Bug Fixes

Real-Time Workshop 6.1 includes several bug fixes made since Version 6.0. You can see a list of the important Version 6.1 bug fixes.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

If you are upgrading from a version earlier than Version 6.1, you should also review major bug fixes for all versions between the version currently installed and Version 6.1.

Real-Time Workshop 6.0 Release Notes

New Features

This section introduces the new features and enhancements added in the Real-Time Workshop 6.0 since Real-Time Workshop 5.0.1. The new features are organized into the following categories:

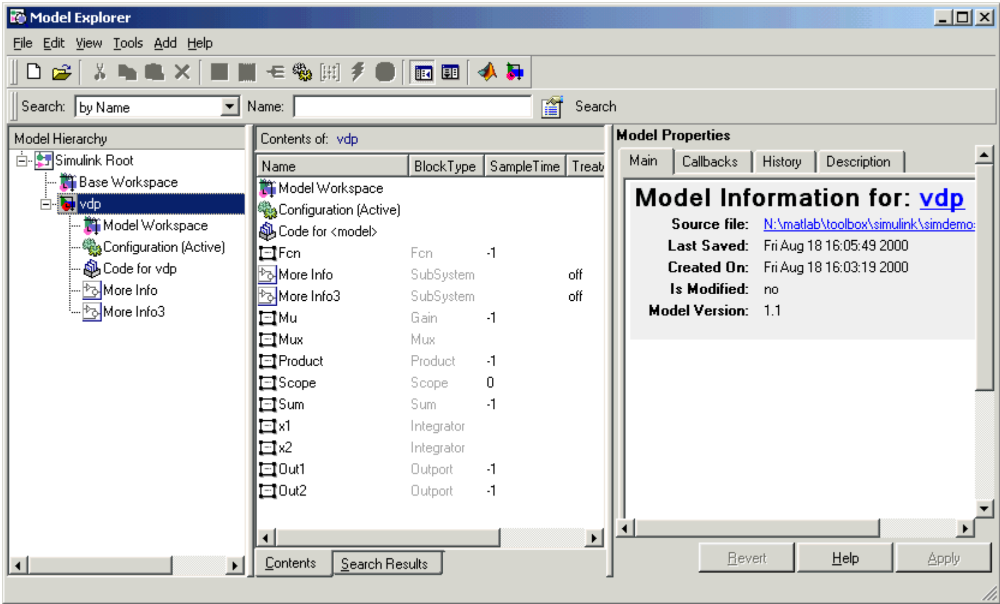
- "User Interface and Configuration Enhancements" on page 5-2
- "Model Referencing (Model Block) Enhancements" on page 5-9
- "Signal, Parameter Handling and Interfacing Enhancements" on page 5-10
- "External Mode Enhancements" on page 5-18
- "Code Customization Enhancements" on page 5-23
- "Timing-Related Enhancements" on page 5-29
- "GRT and ERT Target Unification" on page 5-34

User Interface and Configuration Enhancements

- "New Model Explorer and Configuration Parameters Dialogs for Controlling Code Generation" on page 5-2
- "Generated Code Report Integrated into Model Explorer" on page 5-5
- "Model Advisor Helps You to Configure and Optimize Any Target" on page 5-7
- "Real-Time Workshop Now Supports Intel Compiler" on page 5-8

New Model Explorer and Configuration Parameters Dialogs for Controlling Code Generation

This release of Simulink features a new user interface for simulation and code generation, called Model Explorer, which replaces the **Simulation Parameters** dialog. When you select **Model Explorer** from the **Tools** menu, the Model Explorer opens in a new window containing three panes, as shown below:



The Model Explorer features three resizable, scrolling panes:

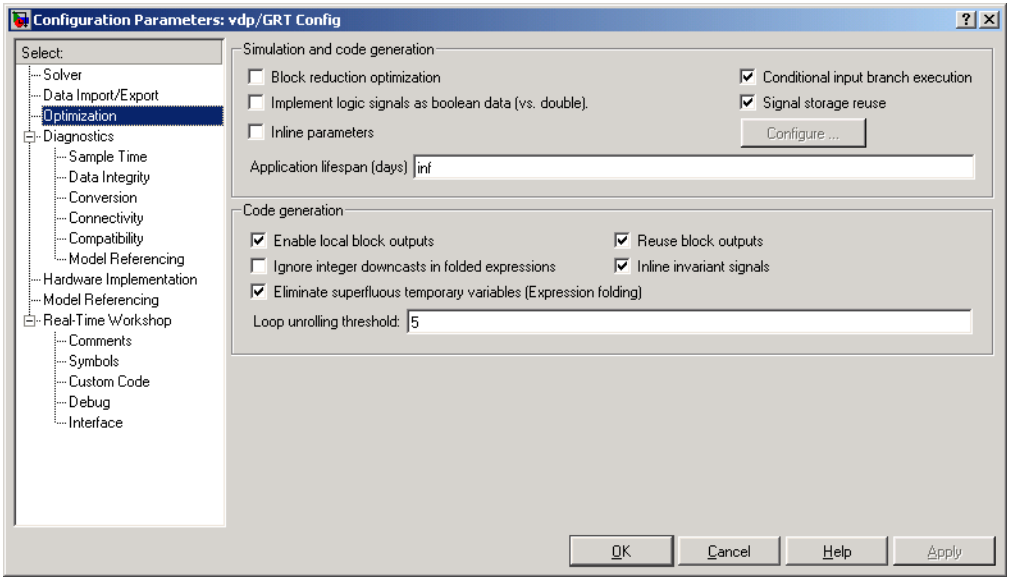
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane

For more information on the Model Explorer, see the Simulink documentation.

You can also control configurations with the standalone **Configuration Parameters** dialog. To activate this interface, a model must be open. You can summon this interface in any of three equivalent ways:

- Choose **Configuration Parameters** from the Simulation menu.
- Choose **Real-Time Workshop -> Options** from the **Tools** menu.
- Type **Ctrl+E**.

The **Configuration Parameters** dialog with the **Optimization** pane selected is shown below:



Code Generation Configuration Components. Code generation aspects of configuration sets fall under the following configuration components and sections within them:

- The Data Import/Export Configuration Component
- The Optimization Configuration Component
- The Diagnostics Configuration Component
- The Hardware Configuration Component
- Real-Time Workshop
  - Real-Time Workshop/General Tab
  - Real-Time Workshop/Debug Tab
  - Real-Time Workshop/Comments Tab
  - Real-Time Workshop/Interface Tab (specific to the current Real-Time Workshop target)
- The Model Referencing Configuration Component

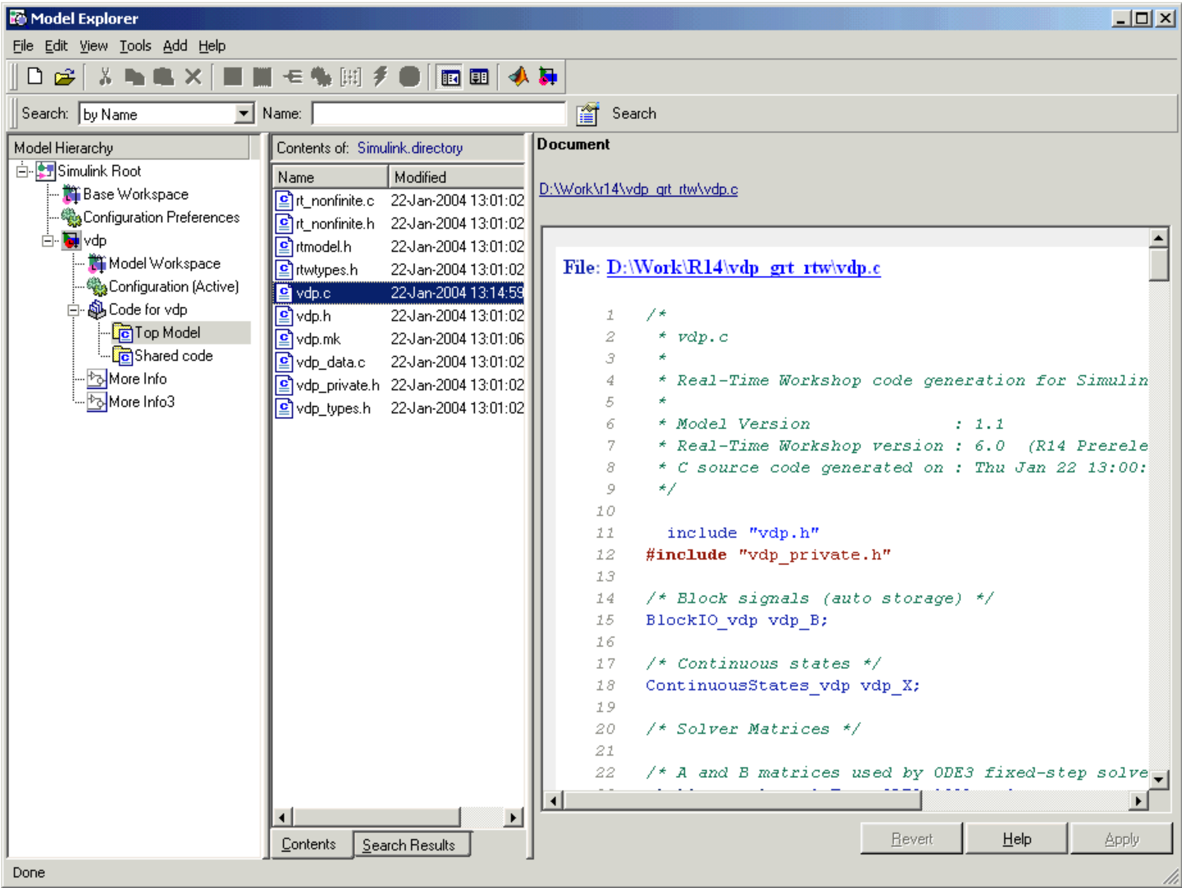
Note The controls visible in configuration component panes can change according to the target you are using and the options you select. This means that the interfaces shown below may not be identical to those that you see in a given context.

For information on configuration parameter options relevant to code generation, see "Adjusting Simulation Configuration Parameters for Code Generation".

Generated Code Report Integrated into Model Explorer

You can now browse files generated by Real-Time Workshop, Real-Time Workshop Embedded Coder, and other products directly in the Model Explorer. This capability supplements HTML code generation reporting, which was available in earlier releases.

When you generate code, or open a model that has generated code for its current target configuration in your working directory, The Hierarchy (left) Pane of Model Explorer contains a node named `Code for` *`model`*. Under that node are other nodes, typically called `Top Model` and `Shared Code`. Clicking on `Top Model` displays in the Content (middle) Pane a list of source code files in the build directory of each model that is currently open. The figure below shows code for the `vdp` model.



In this example, the file `./vdp_grt_rtw/vdp.c` is being viewed. To view any file in the Content Pane, click it once.

The views in the Dialog (right) Pane are read-only. The code listings there contain hyperlinks to functions and macros in the generated code. A hyperlink for the file being viewed sits above it. Clicking it opens that file in a text editing window where you can modify its contents. This is not something you typically do with generated source code, but in the event you have placed custom code files in the build directory, you can edit them as well in this fashion.

If an open model contains Model blocks, and if generated code for any of these models exists in the current `slprj` directory, nodes for the referenced models appear in the Hierarchy pane one level below the node for the top model. Such referenced models do not need to be open for you to browse and read their generated source files.

The node directly underneath the `Top Model` node is named `Shared Code`. It collects files in the appropriate `./slprj/`*`target`*`/_sharedutils` subdirectory, containing shared fixed-point utility code, if any exists.

The structure and contents of `slprj` directories are described in "“Project Directory Structure for Model Reference Targets” in the Real-Time Workshop documentation.

Model Advisor Helps You to Configure and Optimize Any Target

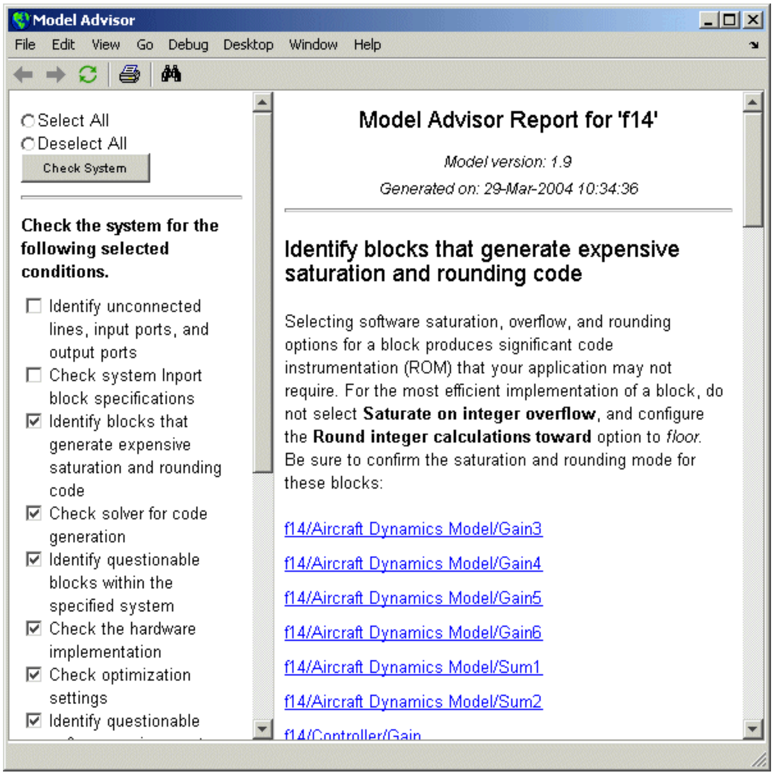
The Model Advisor (formerly called Model Assistant) is a tool that helps you configure any model to optimally achieve your code generation objectives. Using it, you can quickly configure a model for code generation, and identify aspects of your model that impede production deployment or limit code efficiency. Clicking the icon labeled **Advice on *`model`*** in the **Model Hierarchy** pane launches the Model Advisor From Model Explorer. This node is directly below the **Code for model** node, as the above figure shows. Clicking the **Advice** node causes the **Dialog** pane to be labeled **Model Advisor**, and to contain a link, **Start model advisor**. When you click that link, Model Advisor opens a separate HTML window with a set of button and check box controls.

Another way to invoke Model Advisor is to type

```
ModelAdvisor('model')
```

specifying the name of model, at the MATLAB® prompt. If the model (assumed to be on the MATLAB path) is not currently open, Model Advisor will open it.

The following figure illustrates a Model Advisor report:



See "Using the Model Advisor" in the Real-Time Workshop documentation for more information.

Real-Time Workshop Now Supports Intel Compiler

Real Time Workshop now includes support for the Intel compiler (version 7.1 for Microsoft Windows). Note that the Intel compiler requires Microsoft Visual C/C++ version 6.0 or newer to be installed.

Model Referencing (Model Block) Enhancements

- "Including Models as Blocks in Simulations and in Generated Code" on page 5-9
- "Model Reference Demos" on page 5-10

Including Models as Blocks in Simulations and in Generated Code

The new *Model block* from the Simulink library allows one model to include another model as if it were a block. This feature, called *model reference*, works by generating code for included models that the parent model executes from a binary library file. In this release, Model reference works on all Unix and Linux platforms (using the `gcc` compiler), and on Windows PC platforms (using the `lcc` and Visual C++ compilers).

We call models that include Model blocks *top models*. Model referencing uses *incremental loading*; when a top model is opened, any models it references will not be loaded into memory until they are needed or the user opens them.

Note In order to take advantage of incremental model loading, models called from Model blocks must be saved at least once with the current version of Simulink (Release 14). Top and referenced models must have **Inline parameters** set `on`.

When running simulations, models are included in other models by generating code for them in a project directory and creating a static library file called a *simulation target* (sometimes referred to as a SIM target). When Real-Time Workshop generates code for referenced models, it follows a parallel process to create whatever target (e.g., GRT) you have specified (sometimes generically referred to as Real-Time Workshop targets). The generated code is also stored in the project directory, although generated code for parent models is stored (as in previous releases) in a build directory at the same level as the model reference project directory.

In addition to incremental loading, the model referencing mechanism employs *incremental code generation*. This is accomplished by comparing the date, and optionally, the structure of model files of referenced models with those for their generated code to determine whether it is necessary to regenerate model reference targets. You can also force or prevent code generation via the diagnostic setting for **Rebuild options** in the **Model Referencing Configuration Parameters** dialog.

Model Reference Demos

You can learn more about how model blocks work and generate code by running demos. For the full demo suite, at the MATLAB prompt type

```
mdlrefdemos
```

The suite contains three separate demos:

- `mdlref_basic` — General demonstration of using Model blocks
- `mdlref_paramargs` — Passing parameters to referenced models
- `mdlref_bus` — Using bus objects to communicate signals to referenced models
- `mdlref_conversion` — Automatically converting atomic subsystems in models to models called with Model blocks.

For more information on generating code for referenced models, including using `mdlref_conversion`, see "Generating Code from Models Containing Model Blocks" and "Generating Code for a Referenced Model" in the Real-Time Workshop documentation.

Signal, Parameter Handling and Interfacing Enhancements

- “Minimizing Memory Requirements for Parameters and Data During Code Generation” on page 5-17
- “New Options for Controlling Resolution of Signal Objects for Named Signals and States” on page 5-18

New C-API for Accessing Model Block Outputs and Parameters Data

C-API is a target-based Real-Time Workshop feature that provides access to global block outputs and global parameters in the generated code. Using C-API, you can build target applications that log signals, monitor signals and tune parameters while the generated code executes.

In previous releases, to access model parameters via the C-API, a model-specific parameter mapping file, `model_pt.c` was generated. Similarly, to access the `BlockSignals`, `model_bio.c` is generated. The new C-API improves the efficiency and capability of the interface while reducing its code size. In addition, the new API will provide support for:

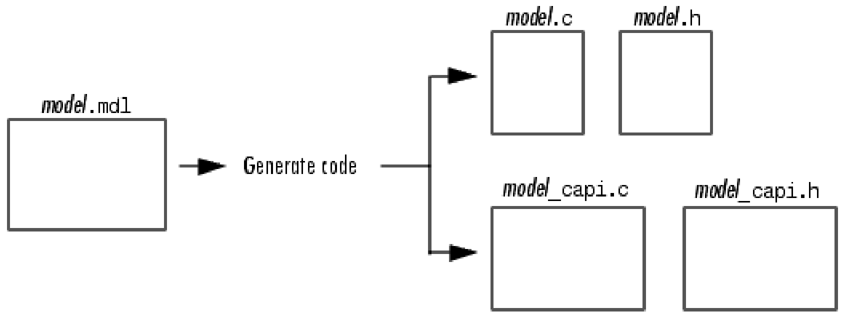
- Referenced models
- Fixed point
- Complex data
- Reusable code

The new interface eliminates redundant fields and also improves consistency between signal and parameter structures. For example, previously the data name was `char_T*` for signals but was `uint_T` for parameters.

The C-API is designed to provide a smaller memory footprint. This is achieved by mapping information common to signals and parameters in smaller structures. An index into the structure map is provided in the actual signal or parameter structure. This allows the sharing of data across signals and parameters.

When you select the C-API feature and generate code, Real-Time Workshop generates two additional files, *`model`*`_capi.c` and *`model`*`_capi.h`, where "model" is the name of the model. Real-Time Workshop places the two C-API files in the build directory, based on settings on the **Configuration**

Parameters dialog. The *model*_capi.c file contains information about global block signals and global parameters defined in the generated code. The *model*_capi.h file is an interface header file between the model source code and the generated C-API. You can use the information in these C-API files to create your application. The generated files are illustrated below.



Compatibility Considerations. The old C API will continue to be available, but at some point will be eliminated. The following table compares the files in the two versions:

CAPI Files	New C-API Files	Old C-API Files
Data structure interface	Unified interface for signals and parameters: /rtw/c/src/rtw_capi.h	Signals Interface: /rtw/c/src/bio_sig.h Parameters Interface: /rtw/c/src/pt_info.h
RTModel C API Interface	/rtw/c/src/ rtw_modelmap.h	/rtw/c/src/mdl_info.h
TLC files	/rtw/c/tlc/mw/ capi.tlc	/rtw/c/tlc/mw/biosig.tlc /rtw/c/tlc/mw/ptinfo.tlc

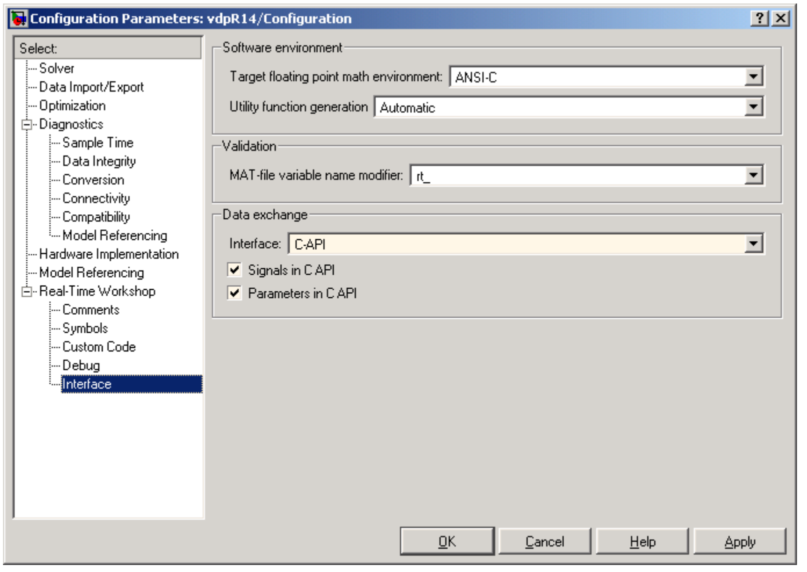
The file rtw_modelmap.h defines structures for mapping data from the rtModel structure. The file rtw_capi.h provides macros for accessing the rtModel.

Note Because the data structures used for the different APIs can conflict, you can generate either C-API or External Mode interface code, but not both at once. The same holds true for ASAP2 interface code, a third data exchange option available for ERT and GRT targets.

Generating the C-API Files. There are two ways to select the C-API feature: Using the **Configuration Parameters** dialog or directly from the MATLAB command line.

To select the C-API with the **Configuration Parameters** dialog

1. In the open model, select **Configuration Parameters** on the **Simulation** menu.
2. Click **Interface** under **Real-Time Workshop** on the left pane.
3. Select `C-API` in the **Interface** field. The **Signals in C API** and **Parameters in C API** check boxes appear, as shown below.
4. If you want to generate C-API for global block outputs, select the **Signals in C API** check box. If you want to generate C-API for global block and model parameters, select the **Parameters in C API** check box. If you select both check boxes, the default, both signals and parameters will appear in the C-API.
5. Click **Apply**.
6. Click **Real-Time Workshop** in the left pane. The **Generate code** button appears in the right pane.
7. Click **Generate Code**.



Activating the C-API from the MATLAB Command Line. From the MATLAB command line you can select or clear the two C-API check boxes on the **Configuration Parameters** dialog using the `uset_param` command. Type one or more of the following commands on the MATLAB command line as desired, where *`modelname`* is the one-word name of the model:

To select **Signals in C API**, type

```
uset_param(modelname,'RTWCAPISignals','on')
```

To clear **Signals in C API**, type

```
uset_param(modelname,'RTWCAPISignals','off')
```

To select **Parameters in C API**, type

```
uset_param(modelname,'RTWCAPIParams','on')
```

To clear **Parameters in C API**, type

```
uset_param(modelname,'RTWCAPIParams','off')
```

Using the C-API in an Application. The C-API provides you with the flexibility of writing your own application code to interact with the signals and parameters. Your target-based application code is compiled with the Real-Time Workshop generated code into an executable. The target-based application code accesses the C-API structure arrays in the `model_capi.c` file. You may have host-based code that interacts with your target-based application code. Or, you may have other target-based code that interacts with your target-based application code. The `rtw_modelmap.h` file provides macros for accessing the structures in these arrays, and their members.

For more details, see "C-API for Interfacing with Signals and Parameters" in the Real-Time Workshop documentation.

Back-propagating Auto, Test-pointed Signal Labels Through Subsystem Output Ports

If a signal exiting an output port of a subsystem has non-auto storage class, the label on that signal is internally propagated backwards into the subsystem so that the code generated for the subsystem uses that signal label which is defined outside the subsystem. Before this release, signal labels were not back-propagated when the signal's storage class was `auto` and it also was test-pointed. Signal labels are now also back-propagated the if the signal is test-pointed.

Declaring Wide Signals, States, and Parameters as ImportedExternPointer

If your model declares the storage class of a signal, state, or parameter as `ImportedExternPointer`, your code must define an appropriate pointer variable. In Version 6, whenever the signal state, or parameter is wide, the variable must be defined as a pointer to an array. In previous versions, an array of pointers was assumed. Here are the changes:

Width	Previous Versions	Version 6
scalar	`double *x1`	`double *x1`
wide	`double *x2[]`	`double *x2`

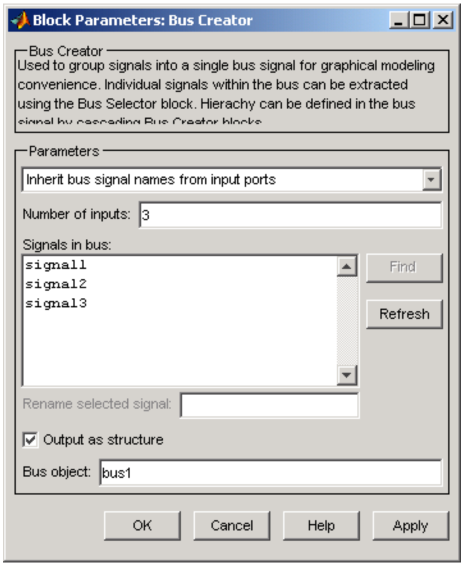
The legacy code could define and initialize data as follows:

```
double x1_data;
double *x1 = &x1_data;
double x2_data[10];
double *x2 = x2_data;
```

This change enables wide data declared as `ImportedExternPointer` to occupy contiguous memory locations, making this storage class useful in more contexts than previously possible.

Bus Creator Blocks Now Can Emit Structures

In the past, the output of a Bus Creator block could not be assigned a storage class. If its new parameter **Output as structure** is selected, the output of the block can be assigned a storage class. This will enable bus signals to occupy contiguous memory. When this parameter is selected, a Simulink Bus object must be specified. You can make and modify bus objects (class `Simulink.Bus`) using the Bus Editor. Type `buseditor` at the MATLAB prompt. An example Bus Creator dialog for a block that outputs a three-element structure is shown below.



For additional details on working with bus and other Simulink data objects, see the "Working with Data" in the Simulink documentation.

Minimizing Memory Requirements for Parameters and Data During Code Generation

When Real-Time Workshop generates code, it creates an intermediate representation of your model (called `model.rtw`), which the Target Language Compiler parses to transform block computations, parameters, signals, and constant data into a high-level language, (e.g., C). Parameters and data are normally copied into the `model.rtw` file, whether they originate in the model itself or come from variables or objects in a workspace.

Models which have large amounts of parameter and constant data (such as lookup tables) can tax memory resources and slow down code generation because of the need to copy their data to `model.rtw`. You can improve code generation performance by limiting the size of data that is copied by using a `set_param` command line option, described below.

Data vectors such as those for parameters, lookup tables, and constant blocks whose sizes exceed a specified value are not copied into the `model.rtw` file. In place of the data vectors, Real-Time Workshop places a special reference key in the intermediate file that enables the Target Language Compiler to access the data directly from Simulink when it is needed and format it directly into the generated code. This results in maintaining only one copy of large data vectors in memory.

You can specify the maximum number of elements that a parameter or other data source can have for Real-Time Workshop to represent it literally in the model.rtw file. Whenever this threshold size is exceeded, Real-Time Workshop writes a reference to the data to the `model.rtw` file, rather than its values. To do this, type the following `set_param` function in the MATLAB Command Window:

```
set_param(0, 'RTWDataReferencesMinSize', size)
```

Provide an integer value for size that specifies the number of data elements above which reference keys are to be used in place of actual data values. The default value for size is 10 elements.

New Options for Controlling Resolution of Signal Objects for Named Signals and States

In prior releases, Real-Time Workshop attempted to resolve all signal objects in a model. Checking all named signals and states was inefficient, complicated error checking, and now has the potential to cause problems for incremental code generation for referenced models. To address these concerns, the current release provides following enhancements:

- Ports and blocks with discrete state now have a setting to indicate whether or not the port/block requires that a signal label be resolved.
- Models have a parameter to control signal resolution. This option is located on the **Diagnostics/Data Integrity** pane of the **Configuration Parameters** dialog.
- A utility function, `disableautosignalresolution`, is provided to assist users in converting existing models (that depended on implicit signal label resolution) to the new, more efficient approach.

External Mode Enhancements

External Mode Changes May Impact Customized Makefiles and Static Main files

The grt, ert, grt_malloc, rsim, rtwin, and tornado targets support external mode. For each of these targets, the template makefiles and the system target files have been changed. In addition, the `main()` files for each target have also been modified (although `ert` may have a dynamic main, which will not be affected). If you have customized any of these static files or their makefiles, you will need to merge your version with those in the current release if you intend to support external mode.

The file *matlabroot*/rtw/ext_mode/common/ext_main.c has also changed slightly. In function ExtCommMain, the line

```
ES = (ExternalSim *)plhs
```

was changed to

```
ES = (ExternalSim *)plhs[0]
```

For xPC, the same change was made in function mexFunction in the file *matlabroot*/toolbox/rtw/targets/xpc/internal/xpc/src/ext_main.c.

If you created your own custom ext_main.c file, you need to merge this change to be compatible with the corresponding change to Simulink.

Floating Scopes Now Work in External Mode

It is now possible to utilize Floating Scope blocks in External mode. A new section in the **External Mode** panel, **Floating scope**, contains:

- **Enable data uploading** check box, which functions as an "arm trigger" button for floating scopes. When the target is disconnected it controls whether or not to "arm when connect" the floating scopes. When already connected it acts as a toggle button to arm/cancel trigger.
- **Duration** edit field, which specifies the duration for floating scopes. By default it is set to auto, which picks up the value specified in the signal and triggering GUI (which by default is 1000).

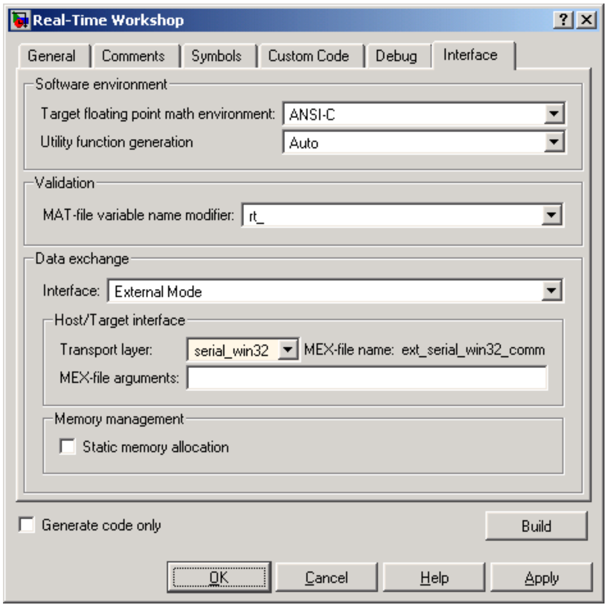
The behavior of wired Scope blocks is unchanged.

Serial Transport Mechanism for External Mode on Windows

Real-Time Workshop now provides code to implement both the client and server side using serial as well as TCP/IP protocols. You can use the socket-based external mode implementation provided by Real-Time Workshop with the generated code, provided that your target system supports TCP/IP. Otherwise, use or customize the serial transport layer option provided.

This design makes it possible for different targets to use different transport layers. The GRT, GRT malloc, ERT, RSim, and xPC targets support host/target communication via TCP/IP and RS232 (serial) and TCP/IP communication. Note that serial transport is implemented only for Windows 32-bit architectures.

To use serial data communications, you need to first instruct Real-Time Workshop to generate support code for external mode. Do this by selecting the **Interface** pane (which is sometimes labelled to specify the current target) of the **Real-Time Workshop Configuration Parameters** dialog. First choose `External mode` from the **Interface** menu in the **Data exchange** section of the dialog. Next, in the **Host/target interface** subsection that then appears, select `serial_win32` for the **Transport layer**, as illustrated below:



The above picture shows the default serial MEX-file interface, `ext_serial_win32_comm`, selected. You can configure Real-Time Workshop to override this with your own serial interface mechanism. See the Real-Time Workshop documentation for details.

The **MEX-file arguments** edit field lets you specify parameters to the external interface MEX-file for communicating with executing targets. For TCP/IP interfaces, `ext_comm` allows three optional arguments:

- The network name of your target
- Verbosity level
- A TCP/IP server port number

For serial transport, optional arguments to `ext_serial_win32_comm` are:

- Verbosity level (`0` or `1`)
- Serial port ID (e.g., `1` for `COM1`, etc.) to be used on the host machine
- Baud rate (selected from the set `1200`, `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, `115200`, with a default baud rate of `57600`).

When you start the target program using a serial connection, you must specify the port ID to use to connect it to the host. Do this by including the `-port` command line option, e.g.,

```
mytarget.exe -port 2 -w
```

If the target program is executing on the same machine as the host and communications is through a loopback serial cable, the target's port ID must differ from that of the host (as specified in the **MEX-file arguments** edit field).

Upgrading Custom Transport Layers for External Mode to Single-Channel Architecture

In earlier releases External Mode had separate logical channels for messages and data. In the TCP/IP example source files, these channels were implemented as separate sockets. Now there is only one logical channel (socket), which handles both data and messages (both of which are now called packets).

Most users will not notice this change. If, however, you have created your own custom transport layer for External Mode, you will have to modify it for the single-channel architecture. Here is a summary of the changes that you may need to make:

On the target side (see example files in `matlabroot/rtw/c/src/`):

- The function `ExtWaitForStartMsgFromHost()` has been renamed `ExtWaitForStartPktFromHost()`.
- The functions `ExtSetHostData()` and `ExtSetHostMsg()` have been merged into `ExtSetHostPkt()`.
- The function `ExtGetHostMsg()` has been renamed `ExtGetHostPkt()`.

On the host side (see example files in `matlabroot/rtw/ext_mode`):

- The functions `ExtTargetDataPending()` and `ExtTargetMsgPending()` have been merged into `ExtTargetPktPending()`.
- The functions `ExtGetTargetData()` and `ExtGetTargetMsg()` have been merged into `ExtGetTargetPkt()`.
- The function `ExtSetTargetMsg()` has been renamed `ExtSetTargetPkt()`.

For complete instructions, see “Creating an External Mode Communication Channel”in the Real-Time Workshop documentation.

New Static Memory Allocation Option for External Mode Code Generation

Code for external mode can now be generated that uses only static memory allocation ("malloc-free" code). The **Static memory allocation** check box, found on the GRT and ERT Target configuration component, enables this feature and activates an edit field in which you can specify the size of the static memory buffer used by external mode. The default value is 1,000,000 bytes. Should you enter too small a value for your application, external mode will issue an out-of-memory error when it tries to allocate more memory than you allowed. In such cases, the value in the **Static memory buffer size** field should be increased and the code should be regenerated. To determine how much memory you need to make available, enable verbose mode on the target (by including `OPTS="-DVERBOSE"` on the `make` command line). As it executes, external mode will display the amount of memory it tries to allocate and the amount of memory available to it each time it attempts an allocation. Should an allocation fail, this console log can be used to adjust the size entered in the **Static memory buffer size** field.

Code Customization Enhancements

Source Code for User S-Functions Now Is Easier to Include

In prior releases, Real-Time Workshop sometimes failed to find S-function source files during a build, even if they were on the MATLAB path, thus aborting the build with an error. This happened because there were no rules dynamically added to the generated makefile for handling the directories in which the S-function MEX-files were located.

Now, Real-Time Workshop adds an include path to the generated makefiles whenever it finds a file named `s-function-name.h` in the same directory that the S-function MEX-file is in. This directory must be on the MATLAB path.

Similarly, Real-Time Workshop will add a rule for the directory when it finds a file `s-function-name.c` (or `.cpp`) in the same directory as the S-function MEX-file.

This enhancement removes the need to copy the S-function source file into the MATLAB current directory or to create an `rtwmakecfg.m` file in the S-function's directory.

Custom Code Block Library Enhancements

The Custom Code Block library has been reinstated into the Real-Time Workshop library. The library has been simplified, so that now the same blocks can be used in subsystems as in top-level models (with minor exceptions). Custom Code blocks enable users to incorporate their own code fragments to specific functions in the source code and header files generated by Real-Time Workshop. The user code can be included in Real-Time Workshop target code generated for referenced models (via Model blocks).

Note that custom code that you include in a configuration set is ignored when building Accelerator, S-Function, and Model Reference Simulation Targets.

Combining User C++ Files with Generated Code

It is now possible to incorporate user C++ files into both Real-Time Workshop and Stateflow builds. Note that Real-Time Workshop itself does not generate C++ code; it simply enables them to be called and incorporated into an executable. For examples of how to use this capability, see the following demos:

- `sf_cpp.mdl` — accessible through **Stateflow Demos** in the Help Browser.
- `sfcndemo_cppcount.mdl` — (in the `sfundemos` demo suite, accessible from Help Browser under **Simulink->Features->S-Function examples**.)

Preventing User Source Code from Being Deleted from Build Directories

In Release 13, the behavior of Real-Time Workshop regarding handling of user source files in the build directory changed. Previously, any `.c` or `.h` files that the user had placed in the build directory were not deleted when rebuilding targets. Now all foreign source files are by default deleted, but can be preserved by following the guidelines given below.

If you put a `.c` or `.h` source file in a build directory, and you want to prevent Real-Time Workshop from deleting it during the TLC code generation process, insert the string `target specific file` in the first line of the `.c` or `.h` file. For example,

```
/*  COMPANY-NAME target specific file
*
```

```
*   This file is created for use with the
*   COMPANY-NAME target.
*   It is used for ...
*/
...
```

Make sure `target specific file` is spelled correctly, and occupies the first line of the source file.

In addition, flagging user files in this manner prevents post-processing them to indent them along with generated source files. Auto-indenting occurred in previous releases to build directory files with names having the pattern *`model`*`_*.c` (where `*` could be any string). The indenting is harmless, but can cause differences to be detected by source control software that might trigger unnecessary updates.

Designating Target-Specific Math Functions

Target configurations can expressly specify which floating-point math library to use when generating code. Real-Time Workshop uses a switchyard called the Target Function Library (TFL) to designate compiler-specific versions of math functions. The mappings created in the TFL allow for C run-time library support specific to a compiler.

Real-Time Workshop provides three different TFLs:

- `ansi_tfl_tmw.mat` — The ANSI-C library (default)
- `iso_tfl_tmw.mat` — Extensions for ISO-C/C99
- `gnu_tfl_tmw.mat` — Extensions for GNU

You choose among them by setting the **Target floating point math environment** pull-down in the **Software Environment** section of the **Interface** tab of the **Real-Time Workshop Configuration Parameters** dialog. This enables you to specify different run-time libraries for different configuration sets within a given model.

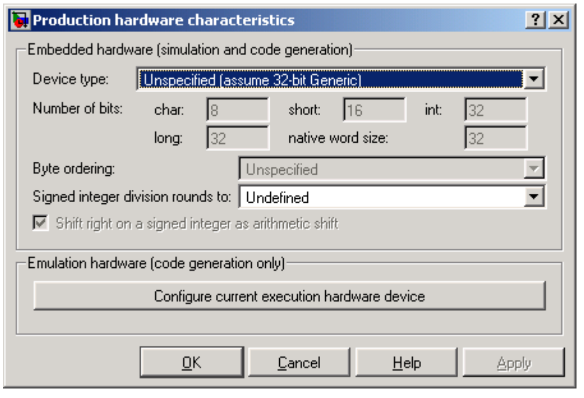
Selecting `ANSI-C` provides the ANSI-C set of library functions. For example, selecting `ANSI-C` would result in generated code that calls `sin()` whether the input argument is double precision or single precision. However, if

ISO-C is selected, the call would instead be to the function sinf(), which is single-precision. If your compiler supports the ISO-C math extensions, selecting the ISO-C library can result in more efficient code.

Hook Files Describing Hardware Characteristics Are Deprecated

Real-Time Workshop now provides a menu that includes more than 20 target processors for the purpose of identifying hardware characteristics such as word lengths. In the previous release, this information was stored in user-supplied *hook files*, which are now deprecated.

When you open a preexisting model that has not been saved using the current version of Simulink, and select **Hardware** in the Configuration Parameters dialog box, the following set of controls appears:



All but one of the parameters below the **Device type** menu are grayed out. This is because these characteristics have been preset for the default target (32-bit Generic), as well as for several dozen known target processors that you can select from that menu.

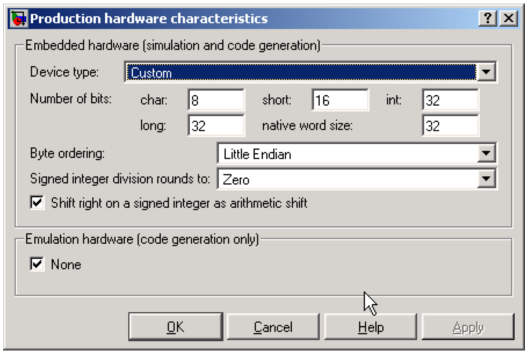
Real-Time Workshop only reads existing hook files when a model created by Version 5 (Release 13) is built for the first time in Version 6 without the user having first specified characteristics of the **Current code generation execution hardware device** on the **Configuration Parameters Hardware Implementation** pane. If you build a model in this under-specified state, Real-Time Workshop will scan the current directory, then the MATLAB path, for an existing hook file with the name `target_rtw_info_hook.m`. If the file is found, its instructions override the defaults in that section. You can subsequently respecify any characteristic freely. If at any point prior to building the target code you do specify **Current code generation execution hardware device**, hook files will be ignored, as hardware characteristics are now configured.

When a preexisting (pre-Version 6) model is opened, the Hardware configuration dialog box displays a **Configure current execution hardware device** button. This button disappears after being pressed once. When code is generated (**Ctrl+B**) for the target the model specifies,

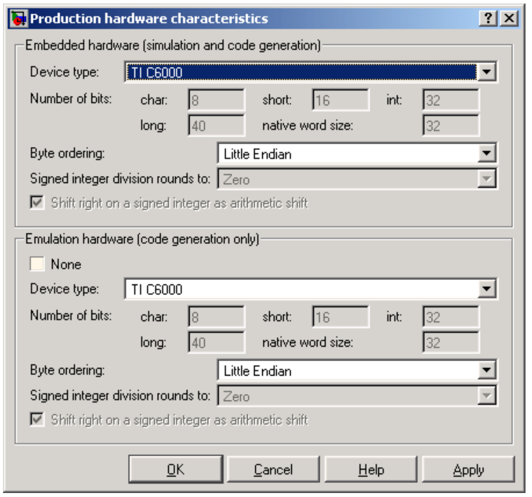
- If the target has a hook file, and the **Configure current execution hardware device** button has not yet been pressed,
  - The hook file is executed and configures the fields specifying current code generation execution hardware device.
  - A warning is issued to the user that the hook file was used.
  - The **Configure current execution hardware device** button on the Hardware configuration dialog box is permanently removed for that model (assuming that you save the model).
- If the target has a hook file and the **Configure current execution hardware device** button has been pressed (removing it),
  - Code is generated for the target using the hardware characteristics for the current code generation execution hardware device (which can default to those of the final embedded hardware device).
  - The hook file for the target is ignored, and is from now on;
  - A warning is issued that a hook file exists but was not used;
- If the target has no hook file, no message to that effect is issued, and the current code generation execution hardware device, if left unspecified, defaults to MATLAB host computer for target device information. A message is displayed during code generation to indicate this

This second group of **Hardware Implementation** pane controls governs how hardware characteristics are handled in generated code. They do not appear unless Real-Time Workshop is installed. Their appearance varies depending on whether hardware configuration characteristics were previously specified for the model or not. If they were not, you see a button (as illustrated in the first of the two above figures) labeled **Configure current execution hardware device**. This button never again appears for this model once code has been generated and the model has been saved.

When you click the **Configure current execution hardware device** button, it is replaced by a check box labeled **None**. This box is selected by default, as shown in the following figure.



If you deselect this box, controls appear for that section that are identical to the controls for the **Embedded Hardware** section above, as shown below (in this example the `TI-C6000` processor is selected).



Timing-Related Enhancements

Application Lifespan Option Optimizes Timer Data Storage

The **Application lifespan (days)** field on the **Optimization** pane of the **Configuration Parameters** dialog lets you specify how long an application which contains blocks that depend on elapsed time should be able to execute before timer overflow. Specifying it determines the word size used by timers in the generated code, and can lower RAM usage.

Application lifespan, when combined with the step size of each task, determinates data type of integer absolute time for each task, as follows:

- If your model does not require absolute time, this option affects neither simulation nor the generated code.
- If your model requires absolute time, this option optimizes the word size used for storing integer absolute time in generated code. This will ensure that timers will not overflow within the lifespan you specify. If you set Application lifespan to `Inf`, two `uint32` words are used.
- If your model contains fixed-point blocks that require absolute time, this option affects both simulation and generated code.

Using 64 bits to store timing data enables models with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. To run a model with a step size of one millisecond (0.001 seconds) for one day would require a 32-bit timer (but it could continue running for 49 days). **Application lifespan** was an ERT-only option in prior releases.

Enabling the Rapid Simulation Target to Time Out

The Rapid Simulation (RSim) Real-Time Workshop target now has a timeout execution option. Use this option to enable the RSim executable to abort if it is taking excessive time. This can happen, for example, in some models when zero crossings are frequent and minor step size is small.

To cause an executing RSim to timeout after `n` seconds, use the `-L` command line option followed by n. For example, suppose you created an RSim executable for the vdp demo; you can run the executable as follows:

```
vdp -L 20
```

After vdp (or vdp.exe) executes for 20 seconds the following will happen:

On Windows platforms, the program will terminate with the message:

```
Exiting program, time limit exceeded
Logging available data ...
```

On Unix platforms the message will be

```
** Received SIGALRM (Alarm) signal @ Fri Jul 25 15:43:23 2003
** Exiting model  'vdp' @ Fri Jul 25 15:43:23 2003
```

You do not need to do anything to your model or its Real-Time Workshop configuration to use this feature. However, you must generate the RSim executable using Version 6.0 or later of Real-Time Workshop for the `-L` flag to be recognized.

New Asynchronous Block Library

The new VxWorks block library (`vxlib1`) allows you to model and generate code for asynchronous event handling, including servicing of hardware generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under a real-time operating system (RTOS).

Although the blocks in the library target a particular RTOS (VxWorks Tornado), full source code and documentation are provided so that you can develop blocks supporting asynchronous event handling for your target RTOS.

The new VxWorks block library supports a superset of the functions of the older Interrupt Templates library. The new library is easier to use, since special Asynchronous Read and Write blocks are no longer required to handle rate transitions.

Note The older Interrupt Templates library (`vxlib`) is obsolete. It is provided only to allow models created prior to Real-Time Workshop 6.0 to continue to operate. If you have models that use `vxlib` blocks, The MathWorks recommends that you change them to use vxlib1 blocks.

The revised "Asynchronous Support" chapter of the Real-Time Workshop User's Guide describes the VxWorks library blocks in detail, including a detailed description of the C and TLC implementations of the Async Interrupt and Task Synchronization blocks.

Summary of VxWorks Library Blocks. The blocks in the library are:

- Async Interrupt block: Generates interrupt-level code. Each output of the Async Interrupt block is associated with a user-specified VxWorks VME interrupt. When an output is connected to the control input of a triggered subsystem such as a function-call subsystem, the generated subsystem code is called from an interrupt service routine (ISR).

- Task Synchronization block: a function-call subsystem that spawns an independent VxWorks task that calls the function-call subsystem connected to its output. The Task Synchronization block is designed to work in conjunction with the Async Interrupt block connected its control input.
- Protected Rate Transition block: The Protected Rate Transition block that is configured to ensure data integrity during data transfers between blocks running at different priorities.
- Unprotected Rate Transition block: The Unprotected Rate Transition block is configured to operate in unprotected/non-deterministic mode during data transfers between blocks running at different priorities.

  Note that the Protected and Unprotected Rate Transition blocks are provided as a convenience. You can use the built-in Simulink Rate Transition block for the same purpose. Rate Transition blocks can be used with any target.

Accessing the VxWorks Library. The VxWorks library (`vxlib1`) is part of the Real-Time Workshop library. You can access the VxWorks library by opening the Simulink Library Browser, clicking the plus sign to the left of the **Real-Time Workshop** entry, and clicking **VxWorks**.

Alternatively, type the following MATLAB command to open the VxWorks library directly:

```
vxlib1
```

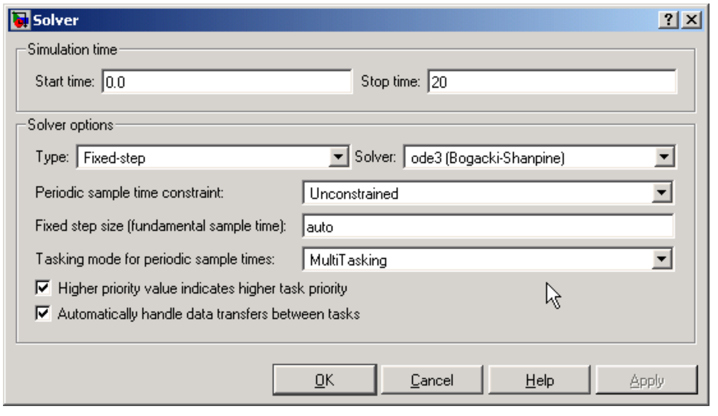
Rate Transition Block Improvements

Since Release 13, the Simulink Signal Attributes library has included a built-in block to handle sample rate transitions (in previous releases rate transitions were handled by Zero-order Hold and Unit Delay blocks, which still exist). The updated Rate Transition block automatically detects whether transitions must be slow-to-fast or fast-to-slow, and acts appropriately. Accordingly, its block parameters dialog no longer includes a setting for **Data transfer type**. The four remaining block parameters are:

- **Ensure data integrity during transfer** check box
- **Ensure deterministic data transfer** check box
- **Outport sample time** text field
- **Initial condition** text field

All Rate Transition blocks in a model will be updated to the new block when the model is saved in Version 6.

When a model using a fixed-step solver is set up for multitasking, Simulink can auto-insert rate transitions between periodic tasks that run at different rates and transfer data. Note that the auto-insertion feature does not apply to transitions to or from non-periodic (asynchronous) tasks. You can control whether or not auto-insertion can happen with the **Automatically handle data transfers between tasks** check box on the **Solver** pane, as shown below.



Simulink configures auto-inserted blocks to insure both data integrity and deterministic data transfer. As mentioned above, they only are inserted when a model is set up for multitasking. Auto-inserted rate transition blocks are non-graphic, thus they do not appear on the block diagram. Nevertheless, they do affect simulation and do affect code generated by Real-Time Workshop, implemented as semaphores or double buffers, depending on the constraints being observed.

Enhanced Absolute and Elapsed Time Computation

Certain blocks require the value of either *absolute* time (i.e., the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). The Real-Time Workshop now provides more efficient time computation services to blocks that request absolute or elapsed time. These timer services are available to all targets that support the real-time model (`rtModel`) data structure. Improvements in the implementation of absolute and elapsed timers include

- Timers are implemented as unsigned integers in generated code.
- In multirate models, at most one timer is allocated per rate, on an as-needed basis. If no blocks executing at a given rate require a timer, no timer is allocated to that rate. This minimizes memory allocated for timers and significantly reduces overhead involved in maintaining timers.
- Allocation of elapsed time counters for use of blocks within triggered subsystems is minimized, further reducing memory usage and overhead.
- Real-Time Workshop provides S-function and TLC APIs that let you access timers for use in your S-functions, in both simulation and code generation.

For more information see “Timing Services” in the Real-Time Workshop documentation.

Improved Single-Tasking Code Generation

New efficiencies in code generation no longer require code generated for single-tasking models to test for sample hits in the base rate task. The code fragment below is an example of such a test in prior versions.

```
if (rtmIsSampleHit(S,0,tid)) { ...
  }
```

Since the base rate task always has a sample hit, such tests are not needed. Elimination of this test improves the runtime performance of the generated code.

GRT and ERT Target Unification

- “Code Format Unification” on page 5-35

- "Compatibility Issues for GRT-Based Targets" on page 5-37
- "Real-Time Workshop and Real-Time Workshop Embedded Coder Feature Set Comparison" on page 5-40
- "Symbol Formatting Options Replaced" on page 5-43

An important goal for both Real-Time Workshop and Real-Time Workshop Embedded Coder in release 14 has been *target unification*. Target unification includes enhancements to the underlying technology and feature sets of both products, such that:

- Both products use a common backend generated code format. This enhancement, termed *code format unification*, has a number of implications (see "Code Format Unification" on page 5-35).
- The set of features common to both products is expanded. Some features and efficiencies formerly exclusive to Real-Time Workshop Embedded Coder and the Embedded Real-Time (ERT) target are now generally available via the Generic Real-Time (GRT) target. Conversely, the Real-Time Workshop Embedded Coder now supports some features that were previously available only via the GRT target (for example, support of continuous-time blocks and noninlined S-functions).

  In general, the GRT and ERT targets have many more common features, but the ERT target offers additional controls for common features.
- Conversion from GRT-based targets to ERT-based targets is simplified.
- The ERT and GRT targets are fully backward-compatible with existing applications.

This note provides a high-level overview and comparison of feature set enhancements and compatibility issues that result from target unification in Real-Time Workshop 6.0 and Real-Time Workshop Embedded Coder 4.0.

Code Format Unification

Before discussing *code format unification*, it is necessary to review the distinction between a target and a code format.

A target (such as the ERT target) is an environment for generating and building code intended for execution on a certain hardware or operating system platform. A target is defined at the top level by a system target file, which in turn invokes other target-specific files.

A code format (such as `Embedded-C` or `RealTime`) is one property of a target. The code format controls decisions made at several points in the code generation process. These include whether and how certain data structures are generated (for example, `SimStruct` or `rtModel`), whether or not static or dynamic memory allocation code is generated, and the calling interface used for generated model functions. In general, the `Embedded-C` code format is more efficient than the `RealTime` code format. `Embedded-C` code format provides more compact data structures, a simpler calling interface, and static memory allocation. These characteristics make the `Embedded-C` code format the preferred choice for production code generation.

In prior releases, only the ERT target and targets derived from the ERT target used the `Embedded-C` code format. Non-ERT targets used other code formats (e.g., `RealTime` or `RealTimeMalloc`).

In release 14, the GRT target uses the `Embedded-C` code format for backend code generation. This includes generation of both algorithmic model code and supervisory timing and task scheduling code. The GRT target (and derived targets) generates a `RealTime` code format wrapper around the `Embedded-C` code. This wrapper provides a calling interface that is backward-compatible with existing GRT-based custom targets. The wrapper calls are compatible with the main program module of the GRT target (`grt_main.c`). Note that this use of wrapper calls incurs some calling overhead; the pure `Embedded-C` calling interface generated by the ERT target is more highly optimized.

The calling interface generated by the ERT target is described in the "Data Structures and Program Execution" chapter of the Real-Time Workshop Embedded Coder documentation. The calling interface generated by the GRT target is described in the "Program Architecture" chapter of the Real-Time Workshop documentation.

Since the GRT target now uses the `Embedded-C` code format for backend code generation, many `Embedded-C` optimizations are available to all Real-Time Workshop users. In general, the GRT and ERT targets have many more common features, but the ERT target offers additional controls for common features. The availability of features is now determined by licensing, rather than being tied to code format.

Code format unification simplifies the conversion of GRT-based custom targets to ERT-based targets. See "Compatibility Issues for GRT-Based Targets" on page 5-37 for a description of target conversion issues.

Compatibility Issues for GRT-Based Targets

If you have developed a GRT-based custom target, it is simple to make your target ERT-compatible. By doing so, you can take advantage of many efficiencies.

There are several approaches to ERT compatibility:

- If your installation is not licensed for Real-Time Workshop Embedded Coder, you can convert a GRT-based target as described in "Converting Your Target to Use rtModel" on page 5-38. This enables your custom target to support all current GRT features, including backend Embedded-C code generation.
- You can create an ERT-based target, but continue to use your customized version of `grt_main.c` module. To do this, you can configure the ERT target to generate a GRT-compatible calling interface, as described in "Generating GRT Wrapper Code from the ERT target" on page 5-39. This lets your target support the full ERT feature set, without changing your GRT-based runtime interface. This approach requires that your installation be licensed for Real-Time Workshop Embedded Coder.
- If your installation is licensed for Real-Time Workshop Embedded Coder, you can re-implement your custom target as a completely ERT-based target, including use of an ERT generated main program. This approach lets your target support the full ERT feature set, without the overhead caused by wrapper calls.

Note If you intend to use custom storage classes (CSCs) with a custom target, you must use an ERT-based target. See "Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation for detailed information on CSCs.

For details on how GRT targets are made call-compatible with previous versions of Real-Time Workshop, see "The Real-Time Model Data Structure" in the Real-Time Workshop documentation.

Converting Your Target to Use rtModel. The real-time model data structure (`rtModel`) encapsulates model-specific information in a much more compact form than the `SimStruct`. Many ERT-related efficiencies depend on generation of `rtModel` rather than `SimStruct`, including:

- Integer absolute and elapsed timing services
- Independent timers for asynchronous tasks
- Generation of improved C-API code for signal and parameter monitoring

To take advantage of such efficiencies, you must update your GRT-based target to use the `rtModel` (unless you already did so for release 13). The conversion requires changes to your system target file, template makefile, and main program module.

The following changes to the system target file and template makefile are required to use `rtModel` instead of `SimStruct`:

- In the system target file, add the following global variable assignment:

```
%assign GenRTModel = TLC_TRUE
```

- In the template makefile, define the symbol `USE_RTMODEL`. See one of the GRT template makefiles for an example.

The following changes to your main program module (i.e., your customized version of `grt_main.c`) are required to use `rtModel` instead of `SimStruct`:

- Include `rtmodel.h` instead of `simstruc.h`.
- Since the `rtModel` data structure has a type that includes the model name, define the following macros at the top of main program file:

```
#define EXPAND_CONCAT(name1,name2) name1 ## name2

#define CONCAT(name1,name2) EXPAND_CONCAT(name1,name2)
```

```
#define RT_MODEL CONCAT(MODEL,_rtModel)
```

- Change the `extern` declaration for the function that creates and initializes the `SimStruct` to:

```
extern RT_MODEL *MODEL(void);
```

- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 14 version of `grt_main.c`.
- Change all function prototypes to have the argument `'RT_MODEL'` instead of the argument `'SimStruct'`.
- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`, `rt_UpdateDiscreteEvents`, `rt_UpdateDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. Change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass into them.

  See the Release 14 version of `grt_main.c` for the list of arguments passed into each function.
- Modify all macros that refer to `SimStruct` to now refer to `rtModel`. `SimStruct` macros begin with the prefix `ss`, whereas `rtModel` macros begin with the prefix `rtm`. For example, change `ssGetErrorStatus` to `rtmGetErrorStatus`.

Generating GRT Wrapper Code from the ERT target. The Real-Time Workshop Embedded Coder supports the **GRT compatible call interface** option. When this option is selected, the Real-Time Workshop Embedded Coder generates model function calls that are compatible with the main program module of the GRT target (`grt_main.c`). These calls act as wrappers that interface to ERT (`Embedded-C` format) generated code.

This option provides a quick way to use ERT target features with a GRT-based custom target that has a main program module based on `grt_main.c`.

See "Code Generation Options and Optimizations" in the Real-Time Workshop Embedded Coder documentation for detailed information on the **GRT compatible call interface** option.

Real-Time Workshop and Real-Time Workshop Embedded Coder Feature Set Comparison

The approach you take to ERT compatibility will depend on the feature set required by your custom target. The following table will help you decide whether or not you require features licensed for Real-Time Workshop Embedded Coder.

For detailed information about these features, see the Real-Time Workshop and Real-Time Workshop Embedded Coder documentation.

Comparison of Features Licensed with Real-Time Workshop Versus Real-Time Workshop Embedded Coder

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
`rtModel` data structure	Full `rtModel struct` generated.	`rtModel` is optimized for the model. Suppression of error status field, data logging fields, and in the `struct` is optional.
Custom storage classes (CSCs)	Code generation ignores CSCs; objects assigned a CSC default to `Auto` storage class.	Code generation with CSCs supported.
HTML code generation report	Basic HTML code generation report.	Enhanced report with additional detail and hyperlinks to the model.
Symbol formatting	Symbols (for signals, parameters etc.) are generated in accordance with hard coded default.	Detailed control over generated symbols.

Comparison of Features Licensed with Real-Time Workshop Versus Real-Time Workshop Embedded Coder (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
User-defined maximum identifier length for generated symbols	Supported	Supported
Generation of terminate function	Always generated.	Option to suppress terminate function.
Combined output/update function	Separate output/update functions are generated.	Option to generate combined output/update function.
Optimized data initialization	Not available.	Options to suppress generation of unnecessary initialization code for zero-valued memory, I/O ports, etc.
Comments generation	Basic options to include or suppress comment generation.	Options to include Simulink block descriptions, Stateflow object descriptions, and Simulink data object descriptions in comments.
Module Packaging Features (MPF)	Not supported.	Extensive code customization features. See the Real-time Workshop Embedded Coder documentation.

Comparison of Features Licensed with Real-Time Workshop Versus Real-Time Workshop Embedded Coder (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
Target-optimized data types header file	Requires full `tmwtypes.h` header file.	Generates optimized `rtwtypes.h` header file, including only the necessary definitions required by the target.
User-defined types	User defined types default to base types in code generation.	User defined data type aliases are supported in code generation.
Simplified call interface	Non-ERT targets default to GRT interface.	ERT and ERT-based targets generate simplified interface.
Rate grouping	Not supported	Supported
Auto-generation of main program module	Not supported; static main program module provided.	Automated and customizable generation of main program module supported. Static main program also available.
MAT-file logging	No option to suppress MAT-file logging data structures.	Option to suppress MAT-file logging data structures.
Reusable (multi-instance) code generation with static memory allocation	Not supported.	Option to generate reusable code.
Software constraint options	Support for floating point, complex, and non-finite numbers always enabled.	Options to enable or disable support for floating point, complex, and non-finite number.

Comparison of Features Licensed with Real-Time Workshop Versus Real-Time Workshop Embedded Coder (Continued)

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
Application life span	User-specified; determines most efficient word size for integer timers. Defaults to `inf`.	User-specified; determines most efficient word size for integer timers.
Software-in-the-loop (SIL) testing	Model reference simulation target can be used for SIL testing.	Additional SIL testing support via auto-generation of Simulink S-Function block.
ANSI-C code generation	Supported	Supported
ISO-C code generation	Supported	Supported
GNU-C code generation	Supported	Supported
Generate scalar inlined parameters	Not supported	Supported
MAT-file variable name modifier	Supported	Supported
Data exchange: C-API, External Mode, ASAP2	Supported	Supported

Symbol Formatting Options Replaced

This note discusses changes in the way that symbols are generated for

- Signals and parameters that have `Auto` storage class
- Subsystem function names that are not user-defined
- All Stateflow names

The following options, all related to formatting generated symbols, have been removed from the Real-Time Workshop GUI and replaced by a default symbol formatting specification.

- **Prefix model name to global identifiers**
- **Include System Hierarchy Number in Identifiers**
- **Include data type acronym in identifier**

The components of a generated symbol now include the root model name, followed by the name of the generating object (signal, parameter, state, etc.), followed by a unique *name mangling* string that is generated (if required) to resolve potential conflicts with other generated symbols.

Note that the length of generated symbols is limited by the **Maximum identifier length** parameter specified on the **Real-Time Workshop / Symbols** pane of the **Configuration Parameters** dialog. The default length is 31 characters. When there is a potential name collision between two symbols, a name mangling string is generated. The string has the minimum number of characters required to avoid the collision. The other symbol components are then inserted. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other components, they are truncated. To avoid this outcome, it is good practice to

- Avoid name collisions in general. One way to do this is to avoid using default block names (for example, `Gain1`, `Gain2...`) when there are many blocks of the same type in the model. Also, whenever possible, make subsystems atomic and reusable.
- Where possible, increase the **Maximum identifier length** to accommodate the length of the symbols you expect to generate.

Model Referencing Considerations. Within a model that uses model referencing, there can be no collisions between the names of the constituent models. When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate full the root model name and the name mangling string (if any). A code generation error occurs if **Maximum identifier length** is not large enough.

When a name conflict occurs between a symbol within the scope of a higher-level model and a symbol within the scope of a referenced model, the symbol from the referenced model is preserved. Name mangling is performed on the symbol from the higher-level model.

Note that the Real-Time Workshop Embedded Coder provides a **Symbol format** field that lets you control the formatting of generated symbols in much greater detail. See "Code Generation Options and Optimizations" in the Real-Time Workshop Embedded Coder documentation for more information.

Major Bug Fixes

Real-Time Workshop 6.0 includes several bug fixes made since Version 5.1.1. This section describes the particularly important Version 6.0 bug fixes.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

If you are upgrading from a version earlier than Version 6.0, you should also review major bug fixes for all versions between the version currently installed and Version 6.0.

Upgrading from an Earlier Release

This section discusses issues involved with upgrading to Real-Time Workshop 6.0 from Version 5.0.

If you are upgrading from Version 5.0, you should also see Version 5.1.1 "Major Bug Fixes" on page 6-3, Version 5.1 "Major Bug Fixes" on page 8-4, and Version 5.01 "Major Bug Fixes" on page 8-4.

If you are upgrading from Version 4.0, see Chapter 11, "Real-Time Workshop 4.0 Release Notes".

Global Data Identifiers for Targets Now Incorporate Model Name

Global data structures, such as `rtB`, `rtP` and `rtY` now have new identifiers in ERT and GRT generated code. For GRT, these names now include the model name followed by `_B`, `_P`, `_Y`, etc. (ERT targets provide you with flexible naming options; see "Symbol Formatting Options Replaced" on page 5-43). The construction of identifiers was changed to prevent name clashes when code for models containing Model blocks is generated and linked. If you are interfacing external code to any Simulink global data, you will probably need to use the GRT compatible calling interface for ERT-based targets (see "Generating GRT Wrapper Code from the ERT target" on page 5-39 for more information). The GRT interface enables you to access global data using the deprecated symbols via a set of macros that map old-style to new-style identifiers. See “Backwards Compatibility of Code Formats” in the Real-Time Workshop documentation for details.

Selecting a Target Programmatically

Simulink models store model-wide parameters and target-specific data in *configuration sets* in Release 14. Every configuration set contains a component that defines the structure of a particular target and the current values of target options. Some of this information is loaded from a system target file when you select a target using the System target file browser. You can configure models to generate alternative target code by copying and modifying old or adding new configuration sets and browsing to select a new target. Subsequently, you can interactively select an active configuration from among these sets (only one configuration set can be active at a given time). Scripts that automate target selection therefore need to emulate this process. They can do this via a new function, `switchTarget`.

The following example shows how to select a target properly, without referencing the deprecated `RTWSystemTargetFile` or `rtwOptions`. The example first obtains a handle (`cs`) to the active configuration set. Next, it stores string variables that correspond to the required Real-Time Workshop system target file, Template makefile, and `make` command settings. The system target file is then selected by passing the `cs` object and the `stf` string to the `switchTarget` function.

```
cs = getActiveConfigSet(model);
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc  = 'make_rtw';
switchTarget(cs,stf,[]);
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

For related information, see “Accessing the rtwOptions Structure Correctly” on page 5-48.

Accessing the rtwOptions Structure Correctly

A new function, `getActiveConfigSet`, provides safe access to option settings stored in the active configuration set. `getActiveConfigSet` returns an object through which you can access properties of the model's active configuration set. The following example shows how to call `getActiveConfigSet` in order to turn the ERT option **Single output/update function** off.

```
cs = getActiveConfigSet(model);
set_param(cs, 'CombineOutputUpdateFcns', 'off');
```

In prior releases, it was possible to access code generation options and other model parameters stored in the `rtwOptions` data structure directly, by using `get_param` and `set_param` calls. In the following code excerpt, for example, the value of the ERT **Single output/update function** option is changed from on to off.

```
options = get_param(model, 'RTWOptions');
strrep(options, 'CombineOutputUpdateFcns=1', 'CombineOutputUpdateFcns=0');
set_param(model, 'RTWOptions', options);
```

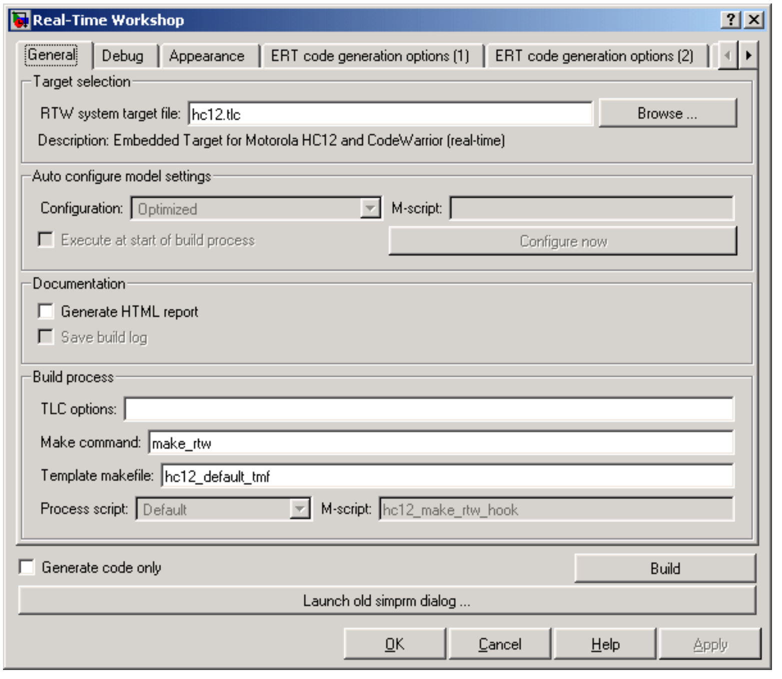
If you have written code that accesses the `rtwOptions` structure directly, as in the above example, you should update your code to use `getActiveConfigSet` instead. Due to changes in underlying data structures, code that accesses `rtwOptions` directly as above will no longer work correctly.

An alternative and more flexible method for automatic configuration of model options is available to users of the Real-Time Workshop Embedded Coder. See "Auto-Configuring Models for Code Generation" in the Real-Time Workshop Embedded Coder documentation for more information.

Defining and Displaying Custom Target Options

For release 14, extensive improvements and revisions have been made in the appearance and layout of code generation options and other target-specific options for Real-Time Workshop targets. If you have developed a custom target, The MathWorks recommends that you take advantage of the Model Explorer to present target options to end users. This requires some modifications to your custom system target file. If you do not want to make these modifications, a mechanism for using the old-style **Simulation Parameters** dialog is available for backwards compatibility.

As an example of what users would see if you do not upgrade, below is an Real-Time Workshop component dialog for the Embedded Target for Motorola® HC12 product, before its system target file was converted to fully use configuration dialogs.



Instead of one Real-Time Workshop/Target tab, this dialog has four: **ERT Code Generation options** 1 through 3, **External mode options**, and **Code Warrior options** (not all are visible in the figure). Targets that have not been updated to use configuration sets will display similar dialogs. In addition, there is a **Launch old simprm dialog** button at the bottom of the dialog. Targets that use the **Simulation Parameters** dialog to handle callbacks will work without updating for Model explorer only if the user uses this button and then builds from the **Simulation Parameters** dialog. Note that configuration set dialogs can issue callbacks but handle them differently than did the **Simulation Parameters** dialog.

See the Real-Time Workshop Embedded Coder 4.0 release notesChapter 11, "Real-Time Workshop 4.0 Release Notes" for details.

SelectCallback Function for System Target Files

The Release 14 API for system target file callbacks provides a new function for use in system target files. `SelectCallback` is associated with the target rather than with any of its individual options. If a `SelectCallback` function is implemented for the target, it is triggered once, when the user selects the target via the System Target File browser.

To implement this callback, use the `SelectCallback` field of the `rtwgensettings` structure. The following code installs a `SelectCallback` function:

```
rtwgensettings.SelectCallback =
  ['custom_open_callback_handler(hDlg, hSrc)'];
```

The arguments to the `SelectCallback` function (`hDlg`, `hSrc`) are handles to private data used by the callback API functions. These handles are restricted to use in system target file callback functions. They should be passed in without alteration, as in this example:

```
slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant', 'on');
```

If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See "Model Reference Compatibility for Custom Targets" on page 5-54 for an example.

Supporting the Shared Utilities Directory in the Build Process

The shared utilities directory (`slprj/`*`target`*`/_sharedutils`) typically stores generated utility code that is common between a top-level model and the models it references. You can also force the build process to use a shared utilities directory for a standalone model. See "Project Directory Structure for Model Reference Targets" in the Real-Time Workshop documentation for details.

If you want your target to support compilation of code generated in the shared utilities directory, several updates to your template makefile (TMF) are required. Note that support for the shared utilities directory is a necessary, but not sufficient, condition for supporting Model Reference builds. See "Model

Reference Compatibility for Custom Targets" on page 5-54 to learn about additional updates that are needed for supporting Model Reference builds.

The exact syntax of the changes can vary due to differences in the make utility and compiler/archive tools used by your target. The examples below are based on the GNU make utility. You can find the following updated TMF examples for GNU and Microsoft Visual C make utilities in the GRT and ERT target directories:

- GRT: `matlabroot/rtw/c/grt/`
  - `grt_lcc.tmf`
  - `grt_vc.tmf`
  - `grt_unix.tmf`
- ERT: `matlabroot/rtw/c/ert/`
  - `ert_lcc.tmf`
  - `ert_vc.tmf`
  - `ert_unix.tmf`

Use the GRT or ERT examples as a guide to the location, within the TMF, of the changes and additions described below.

Note The ERT-based TMFs contain extra code to handle generation of ERT S-functions and Model Reference simulation targets. Your target does not need to handle these cases.

Make the following changes to your TMF to support the shared utilities directory:

1. Add the following make variables and tokens to be expanded when the makefile is generated:

```
SHARED_SRC      = |>SHARED_SRC<|
SHARED_SRC_DIR  = |>SHARED_SRC_DIR<|
SHARED_BIN_DIR  = |>SHARED_BIN_DIR<|
SHARED_LIB      = |>SHARED_LIB<|
```

SHARED_SRC specifies the shared utilities directory location and the source files in it. A typical expansion in a makefile is

```
SHARED_SRC      = ../slprj/ert/_sharedutils/*.c
```

SHARED_LIB specifies the library file built from the shared source files, as in the following expansion.

```
SHARED_LIB      = ../slprj/ert/_sharedutils/rtwshared.lib
```

SHARED_SRC_DIR and SHARED_BIN_DIR allow specification of separate directories for shared source files and the library compiled from the source files. In the current release, all TMFs actually use the same path, as in the following expansions.

```
SHARED_SRC_DIR  = ../slprj/ert/_sharedutils
SHARED_BIN_DIR  = ../slprj/ert/_sharedutils
```

1 Set the SHARED_INCLUDES variable according to whether shared utilities are in use. Then append it to the overall INCLUDES variable.

   SHARED_INCLUDES =

   ifneq ($(SHARED_SRC_DIR),)

   SHARED_INCLUDES = -I$(SHARED_SRC_DIR)

   endif

   INCLUDES = -I. $(MATLAB_INCLUDES) $(ADD_INCLUDES) \

   $(USER_INCLUDES) $(SHARED_INCLUDES)

2 Update the SHARED_SRC variable to list all shared files explicitly.

   ```
   SHARED_SRC := $(wildcard $(SHARED_SRC))
   ```

3 Create a SHARED_OBJS variable based on SHARED_SRC.

   ```
   SHARED_OBJS = $(addsuffix .o, $(basename $(SHARED_SRC)))
   ```

4 Create an `OPTS` (options) variable for compilation of shared utilities.

```
SHARED_OUTPUT_OPTS = -o $@
```

5 Provide a rule to compile the shared utility source files.

$(SHARED_OBJS) : $(SHARED_BIN_DIR)/%.o : $(SHARED_SRC_DIR)/%.c

$(CC) -c $(CFLAGS) $(SHARED_OUTPUT_OPTS) $<

6 Provide a rule to create a library of the shared utilities. The following example is Unix-based.

```
$(SHARED_LIB) : $(SHARED_OBJS)
   @echo "### Creating $@ "
   ar r $@ $(SHARED_OBJS)
   @echo "### Created  $@ "
```

7 Add `SHARED_LIB` to the rule that creates the final executable.

$(PROGRAM) : $(OBJS) $(LIBS) $(SHARED_LIB)

$(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(LIBS) $(SHARED_LIB) $(SYSLIBS)

@echo "### Created executable: $(MODEL)"

8 Remove any explicit reference to `rt_nonfinite.c` from your TMF. For example. change

```
ADD_SRCS = $(RTWLOG) rt_nonfinite.c
```

to

```
ADD_SRCS = $(RTWLOG)
```

Note If your target interfaces to a development environment that is not makefile based, you must make equivalent changes to provide the needed information to your target compilation environment.

Model Reference Compatibility for Custom Targets

This note describes how to adapt your custom target for code generation compatibility with the model reference features introduced in Release 14. Most of the guidelines below concern required modifications to your system target file (STF) and template makefile (TMF).

General Considerations

- A model reference compatible target must be derived from the ERT or GRT targets.
- When generating code from a model that references another model, both the top-level model and the referenced models must be configured for the same code generation target.
- Note that the **External mode** option is not supported in model reference Real-Time Workshop target builds. If the user has selected this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in "Supporting the Shared Utilities Directory in the Build Process" on page 5-50.

System Target File Modifications

Your STF must implement a `SelectCallback` function (see "SelectCallback Function for System Target Files" on page 5-50). Your `SelectCallback` function must declare model reference compatibility by setting the `ModelReferenceCompliant` flag.

The callback is executed if the function is installed in the `SelectCallback` field of the `rtwgensettings` structure in your STF. The following code installs the `SelectCallback` function:

```
rtwgensettings.SelectCallback =
['custom_open_callback_handler(hDlg, hSrc)'];
```

Your callback should set the `ModelReferenceCompliant` flag as follows.

```
slConfigUISetVal(hDlg, hSrc, 'ModelReferenceCompliant',
'on');
```

See Chapter 11, "Real-Time Workshop 4.0 Release Notes" for details on the callback API, including `slConfigUISetVal`.

Template Makefile Modifications

In addition to the TMF modifications described in "Supporting the Shared Utilities Directory in the Build Process" on page 5-50, you must modify your TMF variables and rules. See "Template Makefile Modifications" in the Real-Time Workshop documentation for instructions.

Macro Required in Template Make File for Tornado Target

Tornado 2.2.1 installs some standard header files in an include directory under the target compiler target directory. For example, if you are targeting the Motorola 68k processor for VxWorks with the GCC 2.96 compiler, Tornado installs the header files at the following location:

```
WIND_BASE/host/WIND_HOST_TYPE/lib/gcc-lib/m68k-wrs-vxworks
/gcc-2.96/include
```

To use Tornado 2.2.1 or higher with the Tornado (VxWorks) Real-Time Target, `tornado.tlc`, you must enable a macro in template makefile `tornado.tmf`. To enable the macro

1 Open *`matlabroot`*`/rtw/c/tornado/tornado.tmf`.

2 Search for `TORNADO_TARGET_COMPILER_INCLUDES`.

3 Uncomment the macro `TORNADO_TARGET_COMPILER_INCLUDES` and set it to the include directory that contains the Tornado standard header files.

   Given the path shown above, you would set the macro as follows:

   ```
   TORNADO_TARGET_COMPILER_INCLUDES =
   $(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/m68k-wrs-v
   xworks/gcc-2.96/include
   ```

   Although this example shows the macro definition wrapped, you should include it on a single line.

If you are using a version of Tornado lower than 2.2.1, leave the macro commented out.

Custom Storage Classes Can No Longer Be Used with GRT Targets

In prior releases, it was possible to use Custom Storage Classes with the Generic Real-Time Target if a Real-Time Workshop Embedded Coder license was available. In Release 14, you can no longer use Custom Storage Classes when you generate code for GRT-based targets. If you have licensed Real-Time Workshop Embedded Coder, you should instead use ERT Target, and enable the **GRT compatible call interface** option (found on the **Real-Time Workshop/Interface** tab). Doing this will generate GRT-compatible code using the full code generation capabilities of Real-Time Workshop Embedded Coder, including Custom Storage Classes.

For information on how GRT and ERT targets now compare, see "GRT and ERT Target Unification" on page 5-34. See "Code Generation Options and Optimizations" in the Real-Time Workshop Embedded Coder documentation for detailed information on the GRT compatible call interface option.

Accessing the Number of Sample Times from TLC for Custom Targets

In previous release, you could directly access an undocumented TLC variable, `NumSampleTimes`, which held the number of periodic (synchronous) sample times. In the current release the variable that holds the number of periodic sample times is called `NumSynchronousSampleTimes`. In addition, there are two new variables, `NumAsynchronousSampleTimes` and `NumVariableSampleTimes`. The total number of sample times in a model is given by:

```
NumSampleTimes = NumSynchronousSampleTimes +
NumAsynchronousSampleTimes + NumVariableSampleTimes
```

Do not use `NumSampleTimes`. Instead, call TLC library functions, as follows:

- `LibNumDiscreteSampleTimes()` to access `NumSynchronousSampleTimes`

- `LibNumAsynchronousSampleTimes()` to access `NumAsynchronousSampleTimes`

TLC TLCFILES Built-in Now Returns the Full Path to Model File Rather Than the Relative Path

A change in TLC invocation now specifies a full path to model files rather than a relative path, creates backwards incompatibility in some custom targets.

When migrating Release 13 targets to Release 14, custom target use of the TLC function `TLCFILES` to determine context, such as the path to the model file, may be affected by this change.

ISSLPRMREF TLC Built-in Provides Support for Parameter Sharing with Simulink

To support parameter sharing with Simulink, a new built-in function (`ISSLPRMREF`) has been added to the Target Language Compiler. It returns a Boolean value indicating whether its argument is a reference to a Simulink parameter or not. Using this function can save memory and time during code generation. Here is an example:

```
%if !ISSLPRMREF(param.Value)
  %assign param.Value = CAST("Real", param.Value)
%endif
```

Additional Argument for TLC GENERATE_FORMATTED_VALUE Built-in

The `GENERATE_FORMATTED_VALUE` built-in has a new optional third argument. The syntax for the function is now

```
GENERATE_FORMATTED_VALUE(expr, string, expand)
```

The third argument is a Boolean, which when `TRUE`, causes `expr` to be expanded into raw text before being output. `expand = TRUE` uses much more memory than the default (`FALSE`); set `expand = TRUE` only if the parameter text needs to be processed for some reason before being written to disk.

Known Software and Documentation Problems

Real-Time Workshop Documentation Status

The Real-Time Workshop Getting Started Guide has been fully updated for Version 6.1, and includes a new tutorial on generating code for referenced models. The Real-Time Workshop User's Guide is updated, and includes most of the information on new features described in this chapter. The Real-Time Workshop Target Language Compiler Reference Guide has also been updated; note that it no longer includes an appendix describing all the records that might be encountered in a `model.rtw` file. Some Simulink and Real-Time Workshop Embedded Coder documentation is also relevant to Real-Time Workshop users. This chapter of release notes references sections of those documents.

Refer to the following sections in the "New Features" part of this chapter for overviews of changes and enhancements to Real-Time Workshop and details on how to use them. The new features are categorized as follows:

- "User Interface and Configuration Enhancements" on page 5-2
- "Model Referencing (Model Block) Enhancements" on page 5-9
- "Signal, Parameter Handling and Interfacing Enhancements" on page 5-10
- "External Mode Enhancements" on page 5-18
- "Code Customization Enhancements" on page 5-23
- "Timing-Related Enhancements" on page 5-29
- "GRT and ERT Target Unification" on page 5-34

Also see "Upgrading from an Earlier Release" on page 5-46 for more details on compatibility issues between this and previous versions, particularly with respect to target customizations. You can find related details in the "Real-Time Workshop Embedded Coder 4.0 Release Notes".

DSP Support Documentation Error

The Real-Time Workshop User's Guide Version 5 section "DSP Processor Support" on p. 14-107 contained obsolete information, regarding how to specify word sizes.

DSP targets may use registers with sizes other than 32 bits and vary in their saturation and overflow behavior. In Version 5 (Release 13), these characteristics were specified by target-specific hook files, which were provided for all Version 5 targets supplied by The MathWorks. The `%assign DSP32=1` command to the system target makefile and the `-DDSP32=1` command to the template makefile that formerly handled DSP targets were deprecated in Version 5 and no longer have any effect. However, the documentation did not reflect that fact.

In the current version of Real-Time Workshop, hardware word sizes and other characteristics are specified on a per-processor basis using the **Hardware** configuration dialog. For more information, see "Hook Files Describing Hardware Characteristics Are Deprecated" on page 5-26.

No Code Generation Support for 64-bit Integer Values

Since Release 13, MATLAB has supported both signed (`INT64`) and unsigned (`UINT64`) integers. There is, however, no corresponding support in Real-Time Workshop for such values, meaning that they cannot be read from the Workspace or declared in generated code, including downcasts.

Setting Environment Variable to Run Rapid Simulation Target Executables on Solaris

To run RSim executables outside of MATLAB on the Solaris platform, you need to modify your `LD_LIBRARY_PATH` environment variable to include `bin/sol2` directory where MATLAB is installed. For example, if you have installed MATLAB under `/usr/local/MATLAB` then you need to add `/usr/local/MATLAB/bin/sol2` to your environment variable.

Limitation Affecting Rolling Regions of Discontiguous Signals

This note describes a limitation affecting discontiguous signals that have regions that have a width greater than or equal to the **Loop unrolling threshold**. This parameter is set in the **Optimizations** pane of the **Configuration parameters** dialog.)

Such signal regions are called *rolling* regions.

If a rolling region of a discontiguous signal has storage class `ImportedExternPointer`, all other rolling regions of the signal must also have storage class `ImportedExternPointer`. Otherwise, a code generation error is displayed. If this error occurs, try increasing the **Loop unrolling threshold**.

Code Generation Failure in Nested Directories Under Windows 98

This note describes a limitation affecting both the Simulink Accelerator and Real-Time Workshop, under Windows 98. The problem is due to a limitation of Windows 98.

If the present working directory (pwd) is a folder nested in 7 or more levels, Real-Time Workshop (or Simulink Accelerator) cannot generate code. The workaround is to connect to a higher-level (less deeply nested) directory before initiating the build process.

Turn the New Wrap Lines Option Off

The MATLAB Command Window has a new **Wrap lines** option. Real-Time Workshop frequently displays very long message lines as a build progresses. This can cause some display problems. Therefore, when using Real-Time Workshop, you can turn the **Wrap lines** option off using the **Preferences** setting.

ASAP2 File Generation Changes

The Generating ASAP2 Files chapter in the *Real-Time Workshop Embedded Coder User's Guide* has been moved to an appendix in the Real-Time Workshop User's Guide, and has been updated as explained below.

All procedures have been updated to reflect the fact that the **Simulation Parameters** dialog has been replaced by the **Configuration Parameters** dialog.

The ASAP2 file generation feature is available to Real-Time Workshop users who do not have a Real-Time Workshop Embedded Coder licence. See

"Generating ASAP2 Files" in the Real-Time Workshop documentation for details.

Some changes occurred in the ASAP2 file structure on the MATLAB path.

The following property names for ASAP2 objects `ASAP2.Parameter` and `ASAP2.Signal` have been replaced with standard Simulink object properties.

Property Name...	Changed To...
LONGIG_ASAP2	Description
PhysicalMin_ASAP2	Min
PhysicalMax_ASAP2	Max
Units_ASAP2	DocUnits

See "Generating ASAP2 Files" in the Real-Time Workshop documentation for details.

Note Release 14 supports ASAP2 file generation using built in Simulink data objects. This allows you to use data objects of `Simulink.Signal`, `Simulink.Parameter`, and derived classes to generate ASAP2 files. The ASAP2 Appendix of the Real-Time Workshop User's Guide explains that, to generate ASAP2 files, you should define each signal or parameter as an `ASAP2.Signal` or `ASAP2.Parameter` data object. The appendix refers to the supplied `asap2demo` to illustrate this. Note, however, that even though `ASAP2.Signal` and `ASAP2.Parameter` are supported in Release 14, you should define data objects using the built-in classes `Simulink.Signal` and `Simulink.Parameter` instead of using `ASAP2.Signal` and `ASAP2.Parameter`. For details, see the new demo, `rtwdemo_asap2`. The ASAP2 appendix will be updated in a subsequent release.

Custom Code in Configuration Sets Is Ignored by Certain Targets

Code that you place in the **Custom Code** pane of the **Real-Time Workshop Configuration Parameters** dialog is ignored by the following targets:

- Accelerator
- Real-Time Workshop S-function target
- Model reference simulation target

Custom code can be used with ERT and GRT whether or not the model being built references included models.

Real-Time Workshop 5.1.1 Release Notes

New Features

The following new feature is provided in Version 5.1.1 of Real-Time Workshop.

New -dr Command Line Switch in TLC Detects Cyclic Record Creation

The -dr command line option enables the Target Language Compiler to detect at run time when cyclic records are created and to produce a diagnostic message.

Cyclic records are problematic because they cause memory leaks in TLC. A cyclic record is one which ends up pointing to itself. They only can be constructed manually, as in the following example:

```
%createrecord x { }    %% create an empty record x
%createrecord y { }    %% create an empty record y

%addtorecord x field y %% add a field to x which points to y
%addtorecord y field x %% add a field to y which points to x
```

At this point, a cyclic record exists, i.e. `x.field.field == x`

As this feature significantly slows Target Language Compiler performance, it is off by default.

Major Bug Fixes

Real-Time Workshop 5.1.1 includes important bug fixes made since Version 5.1.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

If you are upgrading from a version earlier than Version 5.1.1, you should also review major bug fixes for all versions between the version currently installed and Version 5.1.1.

Upgrading from an Earlier Release

If you are upgrading from a version earlier than Version 5.1.1, you should also see the Release Notes for all versions between the version currently installed and Version 5.1.1.

Inaccessible Signal Reporting

In previous releases, Simulink and the Real-Time Workshop reported an error whenever the Floating Scope or a user-written S-function tried to access an inaccessible signal during simulation or code generation. In this release, Simulink displays only a warning if you use the `sim` command to start the simulation. Real-Time Workshop generates neither a warning nor an error message.

Real-Time Workshop 5.1 Release Notes

Major Bug Fixes

Real-Time Workshop 5.1 includes important bug fixes made since Version 5.0.1.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

If you are upgrading from a version earlier than Version 5.1, you should also see the Release Notes for all versions between the version currently installed and Version 5.1.1

Real-Time Workshop 5.0.1 Release Notes

New Features

This section introduces the new features and enhancements added in Real-Time Workshop since Version 5.0 (Release 13).

Expanded Hook File Options

This update adds new options for specifying target characteristics via hook files.

During its build process, Real-Time Workshop checks for the existence of *target*_rtw_info_hook.m, where *target* is the base file name of the active system target file. For example, if your system target file is grt.tlc, then the hook file name is grt_rtw_info_hook.m. If the hook file is present (i.e., is on the MATLAB path), the target specific information is extracted via the API found in this file. Otherwise, the host computer is the assumed target.

Three hook file keyword options have been added since release 13:

- TypeEmulationWarnSuppressLevel: Used to suppress warnings about emulation of word sizes. The default value is 0 which gives full warnings. This is the preferred setting when generating code for the production target. Increasing the value gives less warnings. When generating code for a rapid prototyping system, emulation may not be a concern and a suppression level of 2 may be desirable.
- PreprocMaxBitsSint: Specify limitations of the target C preprocessor to do math with signed integers. This is used to prevent errors in the preprocessor phase.

  As an example, suppose the target had 64-bit longs. Porting the generated code to a machine that does not have 64-bit longs can lead to errors in the processing of integer data types. To prevent these errors, a check is included in the generated code.

  ```
  #if ( LONG_MAX != (0x7FFFFFFFFFFFFFFFL) )
  #error Code was generated for compiler with different sized
  longs.
  #endif
  ```

This code requires the preprocessor to compare signed 64-bit integers. Some preprocessors have bugs that cause such comparisons to yield incorrect results. The preprocessor math may only be fully correct for say 32-bit signed integers. To specify, this `PreprocMaxBitsSint` would be set to `32`. Generating the code with this setting causes problematic size checks to be skipped.

```
#if 0
/*
Skip this size verification because of preprocessor
limitation
*/
#if ( LONG_MAX != (0x7FFFFFFFFFFFFFFFL) )
#error Code was generated for compiler with different sized
longs.
#endif
#endif
```

- `PreprocMaxBitsUint`: Specify limitations of the target C preprocessor to do math with unsigned integers. This is just like `PreprocMaxBitsSint` except that it pertains to unsigned integer operations such as

```
#if ( ULONG_MAX != (0xFFFFFFFFFFFFFFFFUL) )
```

If you are not certain about the proper settings for your target, type `rtwtargetsettings` in MATLAB for more details.

Hook Files for Customizing Make Commands

Custom targets may require a target-specific hook file to generate an appropriate make command when a non-default compiler is used. Such M-files should be located on the MATLAB path and be named *`target`*`_wrap_make_cmd_hook.m`, e.g. `MPC555pil_wrap_make_cmd_hook.m` for the MPC555 PIL target. When such a file exists, and returns an appropriate make command, Real-Time Workshop will override its default (e.g., Lcc) batch file wrapping code. For an example make command hook file, see *`matlabroot`*`/toolbox/rtw/rtw/wrap_make_cmd.m`. Note that such hook files are distinct from the target-specific hook files that are used to describe hardware characteristics (see above).

Major Bug Fixes

Real-Time Workshop 5.0.1 includes several important bug fixes made since Version 5.0.

If you are viewing these Release Notes in PDF form, refer to the HTML form of the Release Notes, using either the Help browser or the MathWorks Web site, and use the link provided.

If you are upgrading from a version earlier than Version 5.0.1, you should also see Version 5.0 “Major Bug Fixes” on page 9-26.

9

Real-Time Workshop 5.0 Release Notes

Release Summary

Real-Time Workshop 5.0 includes many new features, numerous improvements in the quality of generated code, as well as enhancements to existing features. This section summarizes new features and enhancements added in the Real-Time Workshop 5.0 since the Real-Time Workshop 4.1 release.

New Features and Enhancements

Code Generation Infrastructure Enhancements

Code Generation Configuration Features

Block-Level Enhancements

Target and Mode Enhancements

TLC, model.rtw, and Library Enhancements

Documentation Enhancements

Major Bug Fixes

If you are upgrading from a version earlier than Version 5.0, you should also see the Release Notes for all versions between the version currently installed and Version 5.0.

Upgrading from an Earlier Release

If you are upgrading from a version earlier than Version 5.0, review the following notes. You should also see the Release Notes for all versions between the version currently installed and Version 5.0.

New Features and Enhancements

This section introduces the new features and enhancements added in the Real-Time Workshop 5.0 since the Real-Time Workshop 4.1. A number of enhancements to Simulink that can impact code generation are also described.

Note For information about closely related products that extend the Real-Time Workshop, see the Release Notes for those products.

Code Generation Infrastructure Enhancements

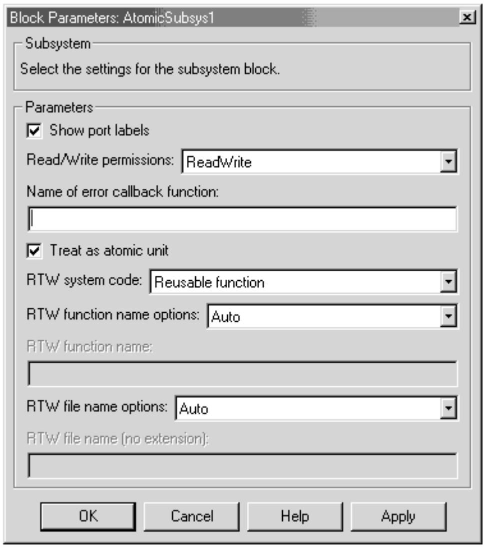
Code for Nonvirtual Subsystems Is Now Reusable

Real-Time Workshop 5.0 alters certain aspects of generated code to implement the capability to reuse code for nonvirtual subsystems. You have the ability to select or override this feature, as well as to specify function and file names from the Real-time Workshop GUI.

In prior releases, each nonvirtual subsystem in a model generated a separate block of code. In some circumstances—for example, when a library block is used multiple times in the same fashion—it is possible to generate a single shared function for the block and call that function multiple times. Consolidating code in this fashion can significantly improve the size and efficiency of generated code.

To implement code reuse, the Real-Time Workshop must pass in appropriate data elements (as function arguments) for each caller of a reused subsystem. Code generated by Real-Time Workshop 5.0 enables such arguments for functions generated for nonvirtual subsystems.

You enable code reuse through the **Subsystem parameters** dialog when both **Treat as atomic unit** and `Reusable function` from the **RTW system code** pull-down menu are selected, as illustrated below.



Reusable code will also be generated, when feasible, when you set **RTW system code** to `Auto`. Then, if only one instance of the subsystem exists, it will be inlined; otherwise a reusable function will be generated if other characteristics of the model allow this.

Certain conditions may make it impossible to reuse code, causing Real-Time Workshop to revert to another **RTW system code** option even though you specify `Reusable function` or `Auto`. When `Reusable function` is specified and reuse is not possible, the result will be a function without arguments. When `Auto` is specified and reuse is not possible, the result will be to inline the subsystem's code (or in special cases, create a function without arguments). Diagnostics are available in the HTML code generation report (if enabled; see "“Generate HTML Report Option Available for Additional Targets” on page 9-23) to help identify the reasons why reuse is not occurring in particular instances. In addition to providing these exception diagnostics, the HTML report's *Subsystems* section also maps each noninlined subsystem in the model to functions or reused functions in the generated code.

Requirements for Generation of Reusable Code from Stateflow Charts. To generate reusable code from a Stateflow chart, or from a subsystem containing a Stateflow chart, all of the following conditions must be met:

- The chart (or subsystem containing the chart) must be a library block (see "Working with Block Libraries" in the Simulink documentation).
- Data in the chart must not be initialized from workspace. The data property **Initialize from workspace** should be off.
- The chart must not output a function call.

See "Nonvirtual Subsystem Code Generation" in the Real Time Workshop documentation for more details.

Packaging of Generated Code Files Simplified

The packaging of generated code into `.c` and `.h` files has changed. The following table summarizes the structure of source code generated by the Real-Time Workshop. All code modules described are written to the build directory.

Note The file packaging of the Real-Time Workshop Embedded Coder differs slightly (but significantly) from the file packaging described here. See "Data Structures and Code Modules" in the Real-Time Workshop Embedded Coder User's Guide for more information.

Real-Time Workshop File Packaging

File	Description
model.c	Contains entry points for all code implementing the model algorithm (MdlStart, MdlOutputs, MdlUpdate, MdlInitializeSizes, MdlInitializeSampleTimes). Also contains model registration code.
*model*_private.h	Contains local defines and local data that are required by the model and subsystems. This file is included by *subsystem*.c files in the model. You do not need to include *model*_private.h when interfacing handwritten code to a model.
model.h	Defines model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (*model*_rtM) via access macros. *model*.h is included by *subsystem*.c files in the model. If you are interfacing your handwritten code to generated code for one or more models, you should include *model*.h for each model to which you want to interface.
*model*_data.c (conditional)	*model*_data.c is conditionally generated. It contains the declarations for the parameters data structure and the constant block I/O data structure. If these data structures are not used in the model, *model*_data.c is not generated. Note that these structures are declared extern in *model*.h.

Real-Time Workshop File Packaging (Continued)

File	Description
*model*_types.h	Provides forward declarations for the real-time model data structure and the parameters data structure. These may be needed by function declarations of reusable functions. *model*_types.h is included by all *subsystem*.h files in the model.
rtmodel.h	Contains #include directives required by static main program modules such as grt_main.c and grt_malloc_main.c. Since these modules are not created at code generation time, they include rt_model.h to access model-specific data structures and entry points. If you create your own main program module, take care to include rtmodel.h.
model_pt.c (optional)	Provides data structures that enable a running program to access model parameters without use of external mode.
model_bio.c (optional)	Provides data structures that enable your code to access block outputs.

If you have interfaced handwritten code to code generated by previous releases of the Real-Time Workshop, you may need to remove dependencies on header files that are no longer generated. Use #include*model*.h directives, and remove #include directives referencing any of the following:

- *model*_common.h (replaced by *model*_types.h and *model*_private.h)
- *model*_export.h (replaced by *model*.h)
- *model*_prm.h (replaced by *model*_data.c)
- *model*_reg.h (subsumed by *model*.c)

Most Targets Use rtModel Instead of Root SimStruct

The GRT, GRT-Malloc, ERT, and Tornado targets now use the *rtModel* data structure to store information about the root model. In prior releases, this information was stored in the `SimStruct` data structure. Since the `SimStruct` data structure was also used by noninlined S-functions, it contained a number of S-function fields that were not needed to represent root model information. The new `rtModel` is a lightweight data structure that eliminates these unused fields in representing the root model. Fields in the rtModel capture model-wide information pertaining to timing, solvers, logging, model data (such as block I/O, and DWork, parameters), etc. To generate code for the ERT target, the `rtModel` data structure is further pruned to contain only those fields that are relevant to the model under consideration.

Note If you have previously customized GRT, GRT-Malloc, or Tornado targets, you should upgrade each customized target to use the rtModel instead of the SimStruct. You can find guidelines for this upgrade path in "Upgrading Customized GRT and GRT-Malloc Targets to Work with Release 13" on page 9-34.

Hook Files for Communicating Target-specific Word Characteristics

In order to communicate details about target hardware characteristics, such as word lengths and overflow behavior, you now need to supply an M-file named `*target*_rtw_info_hook.m`. Each system target file needs to implement a hook file. For GRT (`grt.tlc`), for example, the file must be named `grt_rtw_info_hook.m`, and needs to be on the MATLAB path. If the hook file is not provided, default values based on the host's characteristics will be used, which may not be appropriate. For an example, see `toolbox/rtw/rtwdemos/example_rtw_info_hook.m`. In addition, note that the TLC directive `%assign DSP = 1` no longer has any effect. You need to provide a hook file instead.

Code Generation Unified for Real-Time Workshop and Stateflow

In earlier releases, code generated from Stateflow charts in a model was written to source code files distinct from the source code files (such as `*model*.c`, `*model*.h`, etc.) generated from the rest of the model.

Now, by default, Stateflow no longer generates any separate files from the Real-Time Workshop. In addition, Stateflow generated code is seamlessly integrated with other generated code. For example, all Stateflow initialization code is now inlined.

You can override the default and instruct the Real-Time Workshop to generate separate functions, within separate code files, for a Stateflow chart. To do this, use the **RTW system code** options in the **Block parameters** dialog of the Stateflow chart (see "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop documentation). You can control both the names of the functions and of the code files generated.

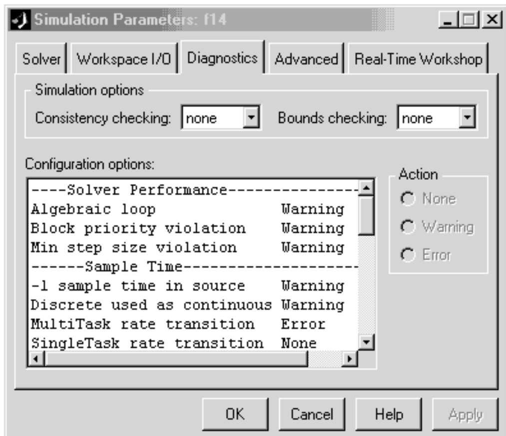
Conditional Input Branch Execution Optimization

This release introduces a new optimization called conditional input branch execution, speeding simulation and execution of code generated from the model. Previously, when simulating models containing Switch or Multiport Switch blocks, Simulink executed all blocks required to compute all inputs to each switch at each time step. In this release, Simulink, by default, executes only the blocks required to compute the control input and the data input selected by the control input at each time step. Likewise, standalone applications generated from the model by Real-Time Workshop execute only the code needed to compute the control input and the selected data input. To explore this feature, look at the `coninputexec` demo.

Code Generation Configuration Features

Diagnostics Pane Items Classified into Logical Groups

To make selecting diagnostics easier, the **Diagnostics** entries on the **Simulation Parameters** dialog have been reorganized according to functionality, and alphabetically within each group, as shown below.

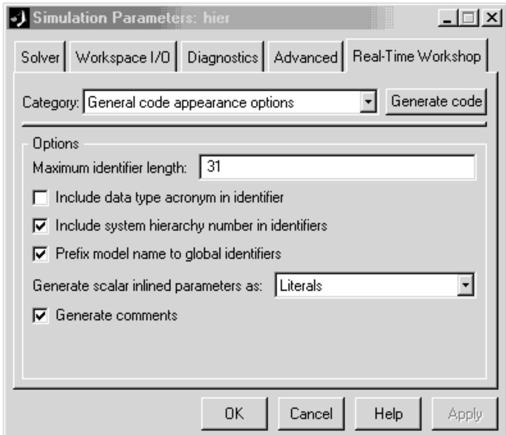


Comments Not Generated for Reduced Blocks When "Show eliminated statements" Is Off

The **Show eliminated statements** option (in the Real-Time Workshop `General code generation options` category) is now off by default. As long as it remains off, Real-Time Workshop no longer generates comments referring to blocks that have been removed from the model via block reduction optimization.

New General Code Appearance Options

A new category has been added to the **Real-Time Workshop** dialog, named `General code appearance options`. This pane adds four new code formatting options to two existing options that formerly occupied other categories. The `General code appearance` dialog is shown below.



The **Maximum identifier length** field allows you to limit the number of characters in function, type definition, and variable names. The default is 31 characters, but Real-Time Workshop imposes no upper limit.

Selecting **Include data type acronym in identifier** enables you to prepend acronyms such as i32 (for long integers) to signal and work vector identifiers to make code more readable. The default is not to include data type acronyms in identifiers.

The **Include system hierarchy number in identifiers** option, when selected, prefixes s#_, where # is a unique integer subsystem index, to identifiers declared in that subsystem. This enhances traceability of code, for example via the hilite_system<`S#'> command. The default is not to include a system hierarchy index in identifiers.

The **Prefix model name to global identifiers** check box is a new option that is ON by default. When this option is on, Real-Time Workshop prefixes subsystem function names with the name of the model (*model_*). The model name is also prefixed to the names of functions and data structures at the model level, when appropriate to the code format. This is useful when you need to compile and link code from two or more models into a single executable, as it avoids potential name clashes.

You can now exercise control over the code style for inlined parameters through a new pull-down menu, **Generate scalar inline parameters as:**`[literals | macros]`. When constant parameters are inlined and declared not tunable, the following code generation options are available:

- Vector parameters were formerly stored as constant parameters in `rtP` vectors. Now they are declared as constant vectors of appropriate type, independent of `rtP`.
- Scalar parameters were formerly inlined as literals. In addition to this approach, users now have the option to have scalar parameters expressed as `#define` macro definitions.

The default is to generate scalar inline parameters as literals.

Note S-functions can mark a run-time parameter as being constant in order to guarantee that it never ends up in the `rtP` data structure. Use `ssSetConstRunTimeParamInfo` in the S-function to register a constant run-time parameter.

Generate comments is an existing global option that was moved from the `General code generation options (cont)` category to this one. As in the prior release, the default for **Generate comments** is `ON`.

Identifier Construction for Generated Code Has Been Simplified

The methods which Real-Time Workshop uses to construct identifiers for variables and functions have been enhanced to make identifiers more understandable and more customizable. As a result of these enhancements

- Changes to sections of the model do not cause identifiers elsewhere to change.
- Reused function input arguments now derive their name from the inport block.
- Subsystem function names can be prefixed by the model name to prevent link errors due to name conflicts.
- Users may specify maximum identifier length (can be > 31 characters).

- A new option exists to include a data type acronym in identifiers.
- Use of _a, _b, ... postfixes to identifiers to prevent name clashes has been dramatically reduced.

See also "New General Code Appearance Options" on page 9-14 for related information.

GUI Control over Behavior of Assertion Blocks in Generated Code

The **Advanced** pane of the **Simulation Parameters** dialog shown above also provides you with a control to specify whether model verification blocks such as Assert, Check Static Gap, and related range check blocks will be enabled, not enabled, or default to their local settings. This **Model Verification block control** popup menu has the same effect on code generated by Real-Time Workshop as it does on simulation behavior, and also may be customized.

For Assertion blocks that are not disabled, the generated code for a model will include one of the following statements

```
utAssert(input_signal);
utAssert(input_signal != 0.0);
utAssert(input_signal != 0);
```

at appropriate locations, depending on the block's input signal type (Boolean, real, or integer, respectively).

By default `utAssert` is a no-op in generated code. For assertions to abort execution you must enable them by including a parameter in the `make_rtw` command. Specify the **Make command** field on the `Target configuration` category pane as follows:

```
make_rtw OPTS='-DDOASSERTS'
```

If you want triggered assertions to not abort execution and instead to print out the assertion statement, use the following `make_rtw` variant:

```
make_rtw OPTS='-DDOASSERTS -DPRINT_ASSERTS'
```

Finally, when running a model in accelerator mode, Simulink will call back to itself to execute assertion blocks instead of using generated code. Thus user-defined callback will still be called when assertions fail.

GUI Control Over TLC %assert Directive Evaluation

Prior versions required specifying the `-da` Target Language Compiler command switch in order for TLC `%assert` directives to be evaluated. Now users can more conveniently trigger `%assert` code by selecting the **Enable TLC Assertions** check box on the **TLC debugging** section of the **Real-Time Workshop** dialog. The default state is for asserts not to be evaluated. You can also control assertion handling from the MATLAB command window. To set or unset assertion handling, use the following command. The default is `Off`.

```
set_param(model, 'TLCAssertion', 'on|off')
```

To see the current setting, use the command

```
get_param(model, 'TLCAssertion')
```

Block-level Enhancements

New Rate Transition Block

In previous releases, Zero-Order Hold and Unit Delay blocks were required to handle problems of data integrity and deterministic data transfer between blocks having different sample rates.

The new Rate Transition block lets you handle sample rate transitions in multirate applications with greater ease and flexibility than the Zero-Order Hold and Unit Delay blocks.

The Rate Transition block handles both types of rate transitions (fast to slow, and slow to fast). When inserted between two blocks of differing sample rates, the Rate Transition block detects the two rates and automatically configures its input and output sample rates for the appropriate type of transition.

The Rate Transition block supports the following modes of operation:

- Protected/Deterministic: By default, the Rate Transition block operates exactly like a Zero-Order Hold (for fast to slow transitions) or a Unit Delay (for slow to fast transitions), and can replace these blocks in existing models without any change in model performance. (There is one exception: in a transition between a continuous block and a discrete block, a Zero-Order Hold must be used.)

In its default mode of operation, the Rate Transition block guarantees the integrity of data transfers and guarantees that data transfers are deterministic.

- Protected/Non-Deterministic: In this mode, data integrity is protected by double-buffering data transferred between rates. The blocks downstream from the Rate Transition block always use the latest available data from the block that drives the Rate Transition block. Maximum latency is less than or equal to 1 sample period of the faster task.

  The drawbacks of this mode are its non-deterministic timing and its use of extra memory buffers. The advantage of this mode is its low latency.

- Unprotected/Non-Deterministic: This mode is the least safe, and is not recommended for mission-critical applications. The latency of this mode is the same as for Protected/Non-Deterministic mode, but memory requirements are reduced since there is no double-buffering.

For more information on the use of the Rate Transition block with the Real-Time Workshop, see "Sample Rate Transitions" in the Real-Time Workshop documentation. For a description of the Rate Transition block, see the description of the Rate Transition block in the Simulink documentation.

S-Function API Extended to Permit Users to Define DWork Properties

The S-Function API has been extended to permit specification of an Real-Time Workshop identifier, storage class, and type qualifier for each `DWork` that an S-Function creates. The extensions consist of the following macros:

- `ssGetDWorkRTWIdentifier(S,idx)`
- `ssSetDWorkRTWIdentifier(S,idx,val)`
- `ssGetDWorkRTWStorageClass(S,idx)`
- `ssSetDWorkRTWStorageClass(S,idx,val)`
- `ssGetDWorkRTWTypeQualifier(S,idx)`

- `ssSetDWorkRTWTypeQualifier(S,idx,val)`

As is the case with data store memory or discrete block states, the Real-Time Workshop identifier may resolve against a Simulink.Signal object. An example has been added to `sfundemos`, in the miscellaneous category.

Lookup Table Blocks Use New Run-Time Library for Smaller Code

Lookup Table (2-D), Lookup Table (3-D), PreLook-Up Using Index Search, and Interpolation using PreLook-Up blocks now generate C code that targets one of the many new specific, optimized lookup table operations in the Real-Time Workshop run-time library. This results in dramatically smaller code size. The library lookup functions themselves incorporate more enhancements to the actual lookup algorithms for speed improvements for most option settings, especially for linear interpolations.

Relay Block Now Supports Frame-based Processing

Relay blocks can now handle frame-based input signals. Each row in a frame-based input signal is a separate set of samples in frames and each column represents a different signal channel. The block parameters should be scalars or row vectors whose length is equal to the number of signal channels. The block does not allow continuous frame-based input signals.

Transport Delay and Variable Transport Delay Improvements

Code generation for models containing the Transport Delay and Variable Transport Delay is now more space-efficient.

Storage Classes for Data Store Memory Blocks

You can now control how Data Store Memory blocks in your model are stored and represented in the generated code, by assigning storage classes and type qualifiers. You do this in almost exactly the same way you assign storage classes and type qualifiers for block states. You can also associate a Data Store Memory block with a signal object, and control code generation for the block through the signal object.

See "Storage Classes for Data Store Memory Blocks" in the Real-Time Workshop documentation for more information.

Target and Mode Enhancements

Rapid Simulation Target Now Supports Variable-Step Solvers

Executables generated for the Rapid Simulation (rsim) target are now able to use any Simulink solver, including variable-step solvers. To use this feature, the target system must be able to check out a Simulink license when running the generated rsim executable. You can maintain backwards compatibility (i.e., fixed-step solvers only, with no need to check out a Simulink license) by selecting `Use RTW fixed step solver` from the **Solver Selection** popup menu on the `Rapid Simulation code generation options` dialog. The default solver option is `Auto`, which will use the Simulink solver module only when the model requires it.

External Mode Support for Rapid Simulation Target

The Rapid Simulation target now includes full support for all features of Simulink external mode. External mode lets you use your Simulink block diagram as a front end for a target program that runs on external hardware or in a separate process on your host computer, and allows you to tune parameters and view or log signals as the target program executes.

External Mode Support for ERT

The Real-Time Workshop Embedded Coder now includes full support for all features of Simulink external mode. External mode lets you use your Simulink block diagram as a front end for a target program that runs on external hardware or in a separate process on your host computer, and allows you to tune parameters and view or log signals as the target program executes.

External Mode Supports Uploading Signals of All Storage Classes

Signals from all storage classes, including custom, can now be uploaded in external mode, as long as signals or parameters have addresses defined. For example, data stored as bit fields or #defines cannot be uploaded, but few other restrictions exist.

Expanded Support for Borland C Compilers

Real-Time Workshop supports version 5.6 of the Borland C compiler.

In addition, Release 13 reinstates support for Borland Version 5.2 "out-of-the-box" for all targets, except when importing Real-Time Workshop-generated S-functions. In such instances, you will need to designate the build directory where the S-function may be found via the `make_rtw` parameter `USER_INCLUDES`. For example, suppose you had generated S-function target code for model `modelA.mdl` in build directory `D:\modelA_sfcn_rtw` and were using that S-function in model `modelB.mdl`. In `modelB.mdl`, the **Make command** field of your `Target configuration` category should define `USER_INCLUDES` as follows:

```
make_rtw "USER_INCLUDES=-ID:\modelA_sfcn_rtw"
```

TLC, model.rtw, and Library Enhancements

New Simulink Data Object Properties Mapped to model.rtw Files

Simulink data objects include several new string properties that can be exploited for customizing code generation. These properties are:

```
Simulink.Data.Description
Simulink.Data.DocUnits
RTWInfo.Alias
```

In this release the Simulink engine does not make use of these properties nor does the Target Language Compiler. The properties are included in the *`model`*`.rtw` file and are reserved for future use. `RTWInfo.Alias` defines the identifier to be used in place of the parent data object (parameter, signal, or state) in the code. The engine checks that the alias is uniquely used by only that object.

SPRINTF Built-in Function Added to TLC

A C-like `sprintf` formatting function has been added which returns a TLC string encoded with data from a variable number of arguments.

`$assign str = SPRINTF(format,var,...)` formats the data in variable `var` (and in any additional variable arguments) under control of the specified `format` string, and returns a string variable containing the values. Operates like C library `sprintf()`, except that output is the return value rather than contained in an argument to `sprintf`.

LCC Now Links Libraries in Directory sys/lcc/lib

The template makefiles have been updated to include linking against `sys/lcc/lib`.

The BlockInstanceData Function has been Deprecated

S-function TLC files should no longer use the BlockInstanceData method. All data used by a block should be declared using data type work vectors (DWork).

New %filescope Directive Added

A new directive, `%filescope`, can be used to limit the scopes of variables to the files they are defined in. All variables defined after the appearance of `%filescope` in a file will have this property; otherwise, they will default to global variables.

Global Variables Accessible Using :: Operator

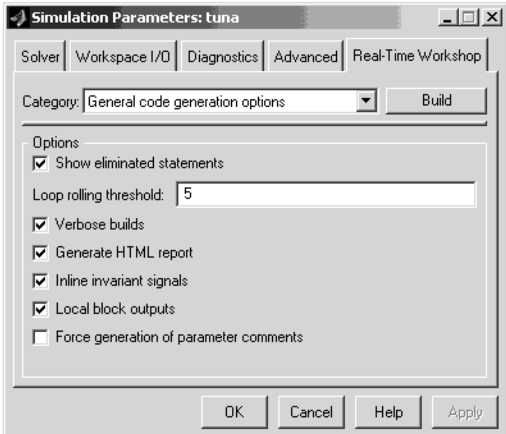
Use of the `::` operator to access global variables is now allowed in TLC files. Variables defined on the command line and records read from *`model`*.rtw files will remain global variables. Nested include files cannot access variables local to the file that included them.

Documentation Enhancements

Generate HTML Report Option Available for Additional Targets

In earlier releases, the **Generate HTML report** option was available only for the Real-Time Workshop Embedded Coder. In the current release, the report is available for all targets (except the S-Function target and the Rapid Simulation target).

The **Generate HTML report** option is now located in the **General code generation options** category of the Real-Time Workshop page of the **Simulation Parameters** dialog, as shown in the picture below.



The option is on by default. Note that an abbreviated report is generated if you do not have Real-Time Workshop Embedded Coder installed.

Expression Folding API Documentation Available

The expression folding API has been documented, and is now promoted for customer use, particularly for user-written, inlined S-functions. In addition, expanded capabilities are available that support the TLC user control variable (`ucv`) in `%roll` directives, and enable expression folding for blocks such as Selector. See “Writing S-Functions That Support Expression Folding” in the Real-Time Workshop documentation.

Real-Time Workshop Documentation

The “Real-Time Workshop User’s Guide” has been significantly updated and reorganized for Version 5.0. Information pertaining to data structures and subsystems has been updated and made more accessible, and new features and GUI changes have been documented. In addition, a new printed and online introductory volume exists, “Getting Started with Real-Time Workshop”. This document explains basic Real-Time Workshop concepts, organizes tutorial material for easier access, and cross-references more detailed explanations in the User’s Guide.

Target Language Compiler Documentation

The Target Language Compiler Reference Guide has been significantly updated and reorganized for Version 5.0. A revised collection of tutorial examples provides new users with a more grounded introduction to TLC syntax. Documentation on the TLC Function Library and contents of `model.rtw` files has also been updated.

Major Bug Fixes

Real-Time Workshop 5.0 includes several bug fixes made since Version 4.1. This section describes the particularly important Version 5.0 bug fixes.

If you are upgrading from a release earlier than Version 5.0, then you should also see the Version 4.1 “Bug Fixes” on page 10-11 and Chapter 11, “Real-Time Workshop 4.0 Release Notes”.

ImportedExtern and ImportedExternPointer Storage Class Data No Longer Initialized

Real-Time Workshop now reverts to its previous behavior of not initializing data whose storage class is `ImportedExtern` or `ImportedExternPointer`. Such initializations are the external code's responsibility.

External Mode Properly Handles Systems with no Uploadable Blocks

Connecting to systems with no uploadable blocks in external mode used to fail and cause Simulink to act as though a simulation was running when none was. The only way to kill the model was to kill MATLAB. Connecting to these systems now will display a warning in the MATLAB command window and then run normally.

Nondefault Ports Now Usable for External Mode on Tornado Platform

In the prior release a bug prevented the use of any but the default port to connect to a Tornado (VxWorks) target via external mode. The problem has been fixed and that configuration now works as documented.

Initialize Block Outputs Even If No Block Output Has Storage Class Auto

Previously, block outputs were initialized only if at least one block output had storage class `auto`. Now even if there are no `auto` Block I/O entries, exported globals and custom signals will be initialized.

Code Is Generated Without Errors for Single Precision Data Type Block Outputs

In cases where a reused block outputs entry is the first single-precision data type block output in the full list of block outputs in the model, Real-Time Workshop now operates without reporting errors. See the Simulink Release Notes for related single-precision block enhancements.

Duplicate #include Statements No Longer Generated

Real-Time Workshop now creates a unique list of C header files before emitting `#include` statements in the *`model`*`.h` file (formerly placed in *`model`*`_common.h`). For backwards compatibility, the old text buffering method for includes is still available for use, but can cause multiple includes in the generated code. You should update your custom code formats to use the `(S)LibAddToCommonIncludes()` functions instead of `LibCacheIncludes()`, which has been deprecated.

Custom Storage Classes Ignored When Unlicensed for Embedded Coder

If a user loads a model that uses custom storage classes, and the user is not licensed for Embedded Coder, the custom storage class is ignored (storage class reverts to auto) and a warning is produced. Previously, this situation would have generated an error.

Erroneous Sample Time Warning Messages No Longer Issued

Erroneous warnings regarding sample times not being in the sample time table for models that contain a variable sample time block and a fixed step solver are no longer issued during model compilation.

Discrete Integrator Block with Rolled Reset No Longer Errors Out

Simulink Accelerator / Real-Time Workshop used to error out if they had a Discrete Integrator block configured in 'ForwardEuler', non-level external reset, and the reset signal was a 'rolled' signal (having a width greater than 5). This has been fixed.

Rate Limiter Block Code Generation Limitation Removed

Simulink Accelerator will now generate code for variable-step solver models that contain a rate limiter block inside an atomic subsystem.

Multiport Switch with Expression Folding Limitation Removed

Simulink Accelerator and Real-Time Workshop no longer generate a Fatal Error for Multiport Switch when expression folding is enabled.

Pulse Generator Code Generation Failures Rectified

Several problems with code generation for the pulse generator block have been eliminated:

- If the block type is PulseGenerator instead of Discrete PulseGenerator, code can now be generated.
- The scalar expansion for the delay variable is now correct.
- The start function for the Time-based mode in a variable-step solver now can generate code.

Note: The first two problems also affected the Simulink Accelerator.

Stateflow I/O with ImportedExternPointer Storage Class Now Handled Correctly

Stateflow input pointers for signals of ImportedExternPointer storage class are now correctly initialized, and no longer error out for charts producing output signals that are nonscalar and of ImportedExternPointer storage class.

Parameters for S-Function Target Lookup Blocks May Now Be Made Tunable

The S-Function target code will now compile for models having lookup and Lookup Table (2-D) blocks when parameters for those blocks are tunable.

PreLookup Index Search Block Now Handles Discontiguous Wide Input

The PreLookup Index Search block formerly only generated code for signals from the first roll region of discontiguous wide inputs, such as from a Max block. This has been fixed.

SimViewingDevice Subsystem No Longer Fails to Generate Code

Code generation no longer aborts for atomic subsystems configured with `SimViewingDevice=on`.

Accelerator Now Works with GCC Compiler on UNIX

The previous version of the Accelerator did not work when the user selected the `gcc` compiler with `mex -setup`. The Accelerator now supports using the `gcc` compiler on UNIX systems.

Expression Folding Behavior for Action Subsystems Stabilized

When a model contains an action subsystem (e.g., a for loop or while iterator subsystem) and expression folding is enabled, invalid or inefficient code sometimes was generated for the model. This problem has been fixed.

Dirty Flag No Longer Set During Code Generation

In previous releases a model would be marked as *dirty* during the code generation process and the status would be restored when the process was finished. With this release the model's dirty status does not change during code generation.

Subsystem Filenames Now Completely Checked for Illegal Characters

In previous releases it was possible to specify a subsystem filename that contained illegal (non-alphanumeric) characters, if the name was long enough and the invalid characters were toward the end of the string. In this release this bug has been fixed, and the entire character string is now validated.

Sine Wave and Pulse Generator Blocks No Longer Needlessly Use Absolute Time

Previously, code generated for the Sine Wave and Pulse Generator blocks accessed absolute time when the blocks were configured as sample based. This access is not necessary and its overhead has been removed from the generated code.

Generated Code for Action Subsystems Now Correctly Guards Execution of Fixed in Minor Time Step Blocks

All blocks contained in an action subsystem must have the same rate unless some are continuous and some are fixed in minor step (a.k.a. *zoh continuous*). If there are both continuous and fixed in minor step blocks then the generated code needs to guard the code for the fixed in minor time step blocks to protect it from being executed in minor time steps.

These guards were not being generated causing some models to have wrong answers and consistency failures. This problem has been fixed and the guards are now generated.

Note This is also a fix for the Simulink Accelerator.

Report Error when Code Generation Requested for Models with Algebraic Loops

Real-Time Workshop does not support models containing algebraic loops. Version 4.1 contained a bug that enabled some models having algebraic loops to generate code which could compute incorrect answers. The models affected were those containing no algebraic loops in their root level but having algebraic loops in one or more subsystems. This bug has been fixed, and now building these models will always cause an error to be reported.

Platform Limitations for HP and IBM

Note The Release 12.0 platform limitation for Real-Time Workshop for the HP and IBM platforms still apply to Release 13. That limitation is described below.

On the HP and IBM platforms, the Real-Time Workshop opens the Release 11 **Tunable Parameters** dialog in place of the **Model Parameter Configuration** dialog. Although they differ in appearance, both dialogs present the same information and support the same functionality.

Upgrading from an Earlier Release

This section discusses issues involved in upgrading from Real-Time Workshop 4.1 to Version 5.0.

If you are upgrading from a version earlier than Version 4.1, you should also see the Release Notes for all versions between the version currently installed and Version 4.1.

Replacing Obsolete Header File #includes

Generated code is packaged into fewer files in this release (see "Packaging of Generated Code Files Simplified" on page 9-9). If you have interfaced code to code generated by previous releases of Real-Time Workshop, you may need to remove dependencies on header files that are no longer generated (such as `*model*_common.h`, `*model*_export.h`, `*model*_prm.h`, and `*model*_reg.h`) and add `#include *model*.h` directives.

Custom Code Blocks Moved from Simulink Library

The Custom Code blocks have been removed in Real-Time Workshop version 5.0 (R13). These blocks are now located in a new library, named `custcode.mdl` (type `custcode` to access them). Because custom code blocks are linked to this new library, backward compatibility is assured.

Updating Custom TLC Code

In this release, a number of changes have been made to *model*.rtw files. If your applications depend on parsing *model*.rtw files using customized TLC scripts, read "model.rtw Changes Between Real-Time Workshop 5.0 and 4.1" in Appendix A of the Target Language Compiler documentation, which describes the structure and contents of compiled models.

Upgrading Customized GRT and GRT-Malloc Targets to Work with Release 13

Substantial changes have been made to the GRT and GRT-Malloc targets in Release 13 to improve the efficiency of generated code. If you have customized either type of target, you should make changes to your modified files to ensure that your target works properly with Release 13 (Real-Time Workshop Version 5.0).

You should begin with the versions of the target files included in this release, and introduce all of your existing customizations to them. If you are unable to follow this upgrade path, then you would need to perform all of the steps outlined in sections A and B below.

A. Changes Resulting from the Replacement of SimStruct with the rtModel

Prior to Release 13 of Real-Time Workshop, the GRT and GRT-Malloc targets used the `SimStruct` data structure to capture and store model-wide information. Since the SimStruct was also used by noninlined S-functions, it suffered from the drawback that some of its fields remained unused when it was used to capture root (model-wide) information. To avoid this drawback, Version 5.0 introduces a special data structure called the `rtModel` to capture root model data.

As a result, `grt_main.c` and `grt_malloc_main.c` need to be updated to accommodate `rtModel`. Following are the changes that you need to make to these files to use the `rtModel` instead of the `SimStruct`:

- Include `rtmodel.h` instead of `simstruc.h` at the top.
- Since the `rtModel` data structure has a type that includes the model name, you need to include the following lines at the top of the file:

```
#define EXPAND_CONCAT(name1,name2) name1 ## name2
#define CONCAT(name1,name2) EXPAND_CONCAT(name1,name2)
#define RT_MODEL CONCAT(MODEL,_rtModel)
```

- Change the `extern` declaration for the function that creates and initializes the `SimStruct` to be:

  ```
  extern RT_MODEL *MODEL(void);
  ```

- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 13 version of `grt_main.c` (or `grt_malloc_main.c`).
- Change all function prototypes to have the argument `'RT_MODEL'` instead of the argument `'SimStruct'`.
- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`, `rt_UpdateDiscreteEvents`, `rt_UpdateDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. You need to change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass into them.

  See `grt_main.c` (or `grt_malloc_main.c`) for the list of arguments that need to be passed into each function.
- You need to modify the all macros that refer to the `SimStruct` to now refer to the `rtModel`. Examples of these modifications include changing
  - `ssGetErrorStatus` to `rtmGetErrorStatus`
  - `ssGetSampleTime` to `rtmGetSampleTime`
  - `ssGetSampleHitPtr` to `rtmGetSampleHitPtr`
  - `ssGetStopRequested` to `rtmGetStopRequested`
  - `ssGetTFinal` to `rtmGetTFinal`
  - `ssGetT` to `rtmGetT`

In addition to the changes to the main C files, you need to change the target TLC file and the template make files.

- In your template make file, you need to define the symbol USE_RTMODEL. See one of the GRT or GRT-Malloc template make files for an example.
- In your target TLC file, you need to add the following global variable assignment:

```
%assign GenRTModel = TLC_TRUE
```

B. Changes Resulting from Moving the Logging Code to the Real-Time Workshop Library:

In Release 13, all the support functions used for logging data have been moved from rtwlog.c to the Real-Time Workshop library. As a result, you need to make the following changes to ensure compatibility with the new logging functions:

- Remove rtwlog.c from all of your template make files.
- In your target's main C file (which was derived from grt_main.c or grt_malloc_main.c), include rt_logging.h instead of rtwlog.h.
- In your target's main C file (which was derived from grt_main.c or grt_malloc_main.c), you need to change the calls to the logging related functions because the prototypes of these functions have changed. See grt_main.c (or grt_malloc_main.c) for the list of arguments that needs to be passed into each function.

The BlockInstanceData Function has been Deprecated

S-function TLC files should no longer use the BlockInstanceData method. All data used by a block should be declared using data type work vectors (DWork).

10

Real-Time Workshop 4.1 Release Notes

Release Summary

Real-Time Workshop 4.1 includes significant new and enhanced features and many improvements in the quality of generated code, including:

- Expression folding, which increases code efficiency and decreases code usage
- External mode support for inlined parameters
- Block states can now be interfaced to externally written code, in a manner similar to signals
- New debugger for Target Language Compiler (TLC) programs
- Support for new Simulink blocks, including control flow constructs such as do-while, for, and if
- Numerous bug fixes

New Features

This section introduces the new features and enhancements added in the Real-Time Workshop 4.1 since the Real-Time Workshop 4.0.

For information about Real-Time Workshop features that are incorporated from recent releases, see "Release Summary" on page 10-2.

Note For information about closely related products that extend the Real-Time Workshop, see the Release Notes for those products.

Block Reduction Option On by Default

The **Block reduction** option (on the **Advanced** page of the **Simulation Parameters** dialog) is now turned on by default. In prior releases, this option was off by default.

Block reduction collapses certain groups of blocks into a single, more efficient block, or removes them entirely. This results in faster model execution during simulation and in generated code.

See "Block Reduction Optimization" in the Real-Time Workshop documentation for more information.

Buffer Reuse Code Generation Option

The **Buffer reuse** option is now available via the **General Code Generation Options (cont.)** category of the Real-Time Workshop page. When the **Buffer reuse** option is selected, signal storage is reused whenever possible.

In previous releases, this option was available only through MATLAB `set_param` and `get_param` commands, such as:

```
set_param(gcs,'bufferreuse','on')
```

The `set_param` and `get_param` commands are still supported.

See the Real-Time Workshop documentation for more information.

Build Directory Validation

The build process now disallows building programs in the MATLAB directory tree. If you attempt to generate code in the MATLAB directory tree, an error message will be displayed prompting you to change to a working directory that is not in the MATLAB directory tree. On a PC, you can continue to build in the directory `matlabroot/Work`.

The build process also prevents building programs when *`matlabroot`* has a dollar sign (`$`) in its MATLAB directory name.

Build Subsystem Enhancements

The **Build Subsystem** feature, introduced in Real-Time Workshop 4.0, lets you generate code and build an executable from any nonvirtual subsystem within a model. In Real-Time Workshop 4.1, the Build Subsystem feature has been enhanced as follows:

- The **Build Subsystem** window now displays additional information about block parameters referenced by the subsystem.
- From the **Build Subsystem** window, you can now inline any parameter or set the storage class of any parameter.

See “Generating Code and Executables from Subsystems” in the Real-Time Workshop Documentation for more information.

C API for Parameter Tuning Documented

Real-Time Workshop provides data structures and a C API that enable a running program to access model parameters without use of external mode.

To access model parameters via the C API, you generate a model-specific parameter mapping file, `model_pt.c`. This file contains parameter mapping arrays containing information required for parameter tuning.

See “C-API for Parameter Tuning and Signal Monitoring” in the Real-Time Workshop documentation for information on how to generate and use the parameter mapping file.

Code Readability Improvements

Improvements to the readability of generated code include:

- Elimination of redundant parentheses.
- Long C statements in the generated code are now split across multiple lines.
- Block comments are more informative.

Control Flow Blocks Support

Simulink 4.1 implements a number of blocks that support logic constructs such as if-else and switch, and looping constructs such as do-while, for, and while. The Real-Time Workshop 4.1 supports code generation from these blocks.

For more information on the control flow blocks, see “Modeling with Control Flow Blocks” in the Simulink documentation.

Expression Folding

Expression folding is a code optimization technique that minimizes the computation of intermediate results at block outputs, and the storage of such results in temporary buffers or variables. Wherever possible, the Real-Time Workshop collapses, or "folds," block computations into single expressions, instead of generating separate code statements and storage declarations for each block in the model.

Expression folding dramatically improves the efficiency of generated code, frequently achieving results that compare favorably to hand-optimized code. In many cases, model computations fold into a single highly optimized line of code.

Most Simulink blocks support expression folding.

For more information, see “Expression Folding” in the Real-Time Workshop documentation.

External Mode Enhancements

Inline Parameters Support

The Real-Time Workshop now lets you use the **Inline parameters** code generation option when building an external mode target program. When you inline parameters, you can use the **Model Parameter Configuration** dialog to remove individual parameters from inlining and declare them to be tunable. This allows you to improve overall efficiency by inlining most parameters, while at the same time retaining the flexibility of run-time tuning for selected parameters that are important to your application. In addition, the **Model Parameter Configuration** dialog offers you options for controlling how parameters are represented in the generated code.

Each time Simulink connects to a target program that was generated with **Inline parameters** on, the target program uploads the current value of its tunable parameters (if any) to the host. These values are assigned to the corresponding MATLAB workspace variables. This procedure ensures that the host and target are synchronized with respect to parameter values.

All targets that support external mode (i.e., `grt`, `grt_malloc`, and Tornado) now allow inline parameters.

See "External Mode Communications Overview" in the Real-Time Workshop documentation for more information.

Status Bar Display

When Simulink is connected to a running external mode target program, the simulation time and other status bar information is now displayed and updated just as it would be in normal mode.

Generate Comments Option

This option lets you control whether or not comments are written in the generated code. See "Comments Options" in the Real-Time Workshop documentation for more information.

Include System Hierarchy in Identifiers

When this option is on, the Real-Time Workshop inserts system identification tags in the generated code (in addition to tags included in comments). The tags help you to identify the nesting level, within your source model, of the block that generated a given line of code.

See “How Symbols Are Formatted in Generated Code” in the Real-Time Workshop documentation for more information.

Rapid Simulation Target Supports Inline Parameters

The Rapid Simulation Target now works with **Inline parameters** on. Note that when **Inline parameters** is on, the storage class for all parameters and signals is silently forced to `auto`.

S-Function Target Enhancements

The S-Function Target **Generate S-function** feature, introduced in Real-Time Workshop 4.0, lets you generate an S-function from a subsystem. This feature has been enhanced as follows:

- The **Generate S-function** window now displays additional information about block parameters referenced by the generating subsystem.
- If you have installed and licensed the Real-Time Workshop Embedded Coder, the **Generate S-function** window lets you invoke the Embedded Coder to generate an S-function wrapper.

See “Automated S-Function Generation” in the Real-Time Workshop documentation for details.

Storage Classes for Block States

For certain block types, the Real-Time Workshop lets you control how block states in your model are stored and represented in the generated code. Using the **State Properties** dialog, you can:

- Control whether or not states declared in generated code are interfaceable (visible) to externally written code. You can also specify that signals are to be stored in locations declared by externally written code.

- Assign symbolic names to block states in generated code.

For more information, see “Block States: Storing and Interfacing” in the Real-Time Workshop documentation.

Support for tilde (~) in Filenames on UNIX Platforms

All filename fields in Simulink now support the mapping of the tilde (~) character in filenames. For example, in a To File block you can specify `<code>~/outdir/file.mat</code>`. On most systems, this will expand to `/home/$USER/outdir/file.mat`. The Real-Time Workshop uses the expanded names.

Target Language Compiler 4.1

This section summarizes changes that have been made to the Target Language Compiler in this release. See also “TLC Compatibility Issues” on page 10-16.

New TLC Debugger

The TLC debugger helps you identify programming errors in your TLC code. The debugger lets you set breakpoints in your TLC code, execute TLC code line-by-line, examine and change variables, and perform many other useful operations.

The TLC debugger operates during code generation, incurring almost no overhead in the code generation process. You can invoke the debugger:

- By selecting options in the **TLC debugging options** category of the Real-Time Workshop page
- By including `%breakpoint` statements in your TLC file.
- By using the MATLAB `tlc` command, as in

```
tlc -dc <options>
```

- By using the -dc build option in the **System target file** field of the **Real-Time Workshop** page.

For more information, see “Debugging TLC Files” in the Target Language Compiler documentation.

model.rtw File Format Changes

The format of the *model*.rtw file has changed. See “TLC Compatibility Issues” on page 10-16.

Cleanup of Block I/O Connection Handling in TLC

The handling of signal connections in `rtw/c/tlc/blkiolib.tlc` and `rtw/ada/tlc/blkiolib.tlc` was reworked. See the description of `LibBlockInputSignal` in the Target Language Compiler documentation.

Literal String Support

If a string constant is preceded by an `L` format specifier (as in `L"string"`), Target Language Compiler performs no escape character processing on that string. This is useful for specifying PC-style paths without using double backslash characters.

```
%addincludepath L"C:\mytlc"
```

The following examples are equivalent.

- `L"d:\this\is\a\path"`
- `"d:\\this\\is\\a\\path"`

New TLC Library Functions

The following functions have been added to the TLC Function Library:

- `LibBlockInputSignalConnected`
- `LibBlockInputSignalLocalSampleTimeIndex`
- `LibBlockInputSignalOffsetTime`
- `LibBlockInputSignalSampleTime`
- `LibBlockInputSignalSampleTimeIndex`
- `LibBlockOutputSignalOffsetTime`

- `LibBlockOutputSignalSampleTime`
- `LibBlockOutputSignalSampleTimeIndex`
- `LibBlockMatrixParameterBaseAddr`
- `LibBlockParameterBaseAddr`
- `LibBlockNonSampledZC`

See the Target Language Compiler documentation for information on these functions.

TLC Bug Fixes

- Fixed a bug where local variables of calling functions were sometimes incorrectly visible to called functions.
- The `ISINF`, `ISNAN`, and `ISFINITE` functions now work for complex values.
- The `%filescope` directive now works as documented.
- Zero indexing on complex numbers is now supported.

  In prior releases, the Target Language Compiler allowed 0 indexing for integer and real values, but not for complex values. This restriction has been removed in the Target Language Compiler 4.1, as shown in the following example.

  ```
  %assign a = 1.0 + 3.0i
  %assign b = a[0] %% zero index now allowed
  ```

- Fixed a crash that occurred if `ROLL_ITERATIONS` was called outside of a `%roll` construct. `ROLL_ITERATIONS` returns `NULL` if called outside of a `%roll` construct.
- TLC now allows use of any path separator character independent of operating system. You can use either `\` or `/` as a path separator character on Unix or Windows).
- Fixed a bug in the compare for equality operation. `0.0` now compares equal to `-0.0`.

Bug Fixes

The Real-Time Workshop 4.1 includes the bug fixes described in this section. See also “TLC Bug Fixes” on page 10-10 for bug fixes specific to the Target Language Compiler.

Block Reduction Crash Fixed

A problem that crashed MATLAB due to a segmentation fault during the block reduction process has been fixed. This problem occurred only if the **Block Reduction** option was on, and if a Scope block was connected to a block that was removed due to block reduction.

Build Subsystem Gives Better Error Message for Function Call Subsystems

The **Build Subsystem** feature does not currently support triggered function-call subsystems. The Real-Time Workshop now gives a more informative error message when a **Build Subsystem** is attempted with a triggered function-call subsystem.

Check Consistency of Parameter Storage Class and Type Qualifier

The Real-Time Workshop now checks for consistency of parameter storage class and type qualifier when a parameter is specified by both the **Model Parameter Configuration** dialog and a referenced Simulink data object.

Code Optimization for Unsigned Saturation and DeadZone Blocks

When the lower limit of a Saturation or DeadZone block is a zero and is nontunable, and the data type is unsigned, the comparison against the lower limit is omitted from the code. Similarly, if the upper or lower limit of the Saturation block is nontunable and nonfinite, the comparison against the infinite limit is omitted.

Correct Code Generation of Fixed-Point Blockset Blocks in DSP Blockset Models

A code generation bug involving some DSP Blockset blocks (see list below) was fixed. When these blocks were driven by a block from the Fixed-Point Blockset, generated code would write outside array memory bounds. The following DSP Blockset blocks generated incorrect code:

- Delay Line
- Frame Status Conversion
- Matrix Multiply
- Multiport Selector
- Pad
- Submatrix
- Window Function
- Zero Pad

Correct Compilation with Green Hills and DDI Compilers

Compilation errors for files associated with matrix inversion in the *matlabroot*`/rtw/c/libsrc` directory were fixed. These errors occurred with the Green Hills and DDI compilers.

Fixed Build Error with Models Having Names Identical to Windows NT Commands

This fix prevents an error that occurred when building models having names identical to Windows NT internal commands. Examples would be models named `verify` or `path`. Such model names are now allowed.

Fixed Error Copying Custom Code Blocks

An error in the Custom Code block `Copyfcn` callback was fixed. The problem caused an error when copying a custom code block within a model.

Fixed Error in commonmaplib.tlc

A typo in rev 1.17 of `commonmap.tlc` was fixed. This typo caused an error during code generation, when using the `grt_malloc` target with **External mode** selected.

Fixed Name Clashes with Run-Time Library Functions

The Real-Time Workshop now uses the macros `rt_min` and `rt_max` to avoid clashing with run-time library `min` and `max` functions.

Improved Handling of Sample Times

The sample time handling for the S-function and ERT targets has been improved to use the compiled sample time instead of the user specified sample time on the input port blocks.

Look-Up Table (n-D) Code Generation Bug Fix

The Real-Time Workshop now generates correct code for Look-Up Table (n-D) blocks having 5 or more dimensions with different dimension sizes.

Parenthesize Negative Numerics in Fcn Block Expressions

Fcn block expressions in the generated code failed to compile in the case of a unary operator preceding a workspace variable with a negative value, such as the expression

```
-v*u
```

Such expressions are now enclosed in parentheses, as in

```
(-v) * u
```

Removed Unnecessary Warnings and Declarations from Generated Code

Several unnecessary warnings and declarations in the generated code have been removed. These include:

- In functions where the `tid` argument is not referenced, the declaration

  ```
  (void)tid
  ```

  is no longer generated. (The `tid` argument is required because the function API is predefined.)
- Warnings involving `const` casts were suppressed in some of the `rtw/c/libsrc` modules.

Retain .rtw File Option Now Works in Accelerator Mode

In previous releases, the **Retain .rtw file** option (on the TLC Debugging Options page of the **Simulation Parameters** dialog) was ignored if Simulink was in Accelerator mode. Now, you can retain the *`model`*`.rtw` file during a build, regardless of the simulation mode.

S-Function Target Memory Allocation Bug Fix

A segmentation fault during generation of S-functions was removed by fixing the memory management of the port data structure.

Upgrading from an Earlier Release

This section discusses issues involved in upgrading from the Real-Time Workshop Version 4.0 to the Real-Time Workshop Version 4.1.

For information about upgrading from a release earlier than 4.0, see “Upgrading from an Earlier Release” on page 10-15 in the Real-Time Workshop 4.0 Release Notes.

RTWInfo Property Changes

If you use the Simulink Data Object classes `Simulink.Signal` or `Simulink.Parameter`, or have implemented classes derived from these, note the following information concerning the `RTWInfo` properties.

In Release 12.0, the `RTWInfo` class had a `TypeQualifier` property, corresponding to the **RTW storage type qualifier** field of signal ports and parameters.

Real-Time Workshop 4.1 now supports creation of custom storage classes, removing the need for the `TypeQualifier` property. You should use custom storage classes when type qualification is needed.

By default, the `TypeQualifier` property of `RTWInfo` objects is no longer visible in the Simulink Data Explorer. Also, the `TypeQualifier` property is no longer written to `ObjectProperties` records in the *`model`*`.rtw` file. For backward compatibility, the `TypeQualifier` property remains active. The property can be set and retrieved through a direct reference. For example,

```
Kp.RTWInfo.TypeQualifier = 'const'
```

or

```
tq = Kp.RTWInfo.TypeQualifier
```

You can make the `TypeQualifier` property visible in the Simulink Data Explorer for the duration of a MATLAB session. To do this, execute the following command prior to opening the Simulink Data Explorer),

```
feature('RTWInfoTypeQualifier',1)
```

The above command also directs the Real-Time Workshop to include the `TypeQualifier` property in `ObjectProperties` records in the *`model`*`.rtw` file.

For more information see “Simulink Data Objects and Code Generation” in the Real-Time Workshop documentation.

S-Function Target MEX-Files Must Be Rebuilt

S-function MEX-files generated by the S-function target under Release 11 are not compatible with Release 12. The incompatibilities are due to new features, such as parameter pooling, introduced in Release 12.0.

If you have built S-function MEX-files with the S-function target under Release 11, you must rebuild them. See “The S-Function Target” in the Real-Time Workshop documentation for more information.

TLC Compatibility Issues

model.rtw File Format Changes

The format of the *`model`*`.rtw` file has changed. For more information, see the Target Language Compiler documentation.

Reordering of BlockTypeSetup and BlockInstanceSetup Calls

During the initialization phase of code generation, the Target Language Compiler makes a pass over all blocks in the model and executes several functions, including:

- Each block’s `BlockTypeSetup` function the first time that block type is encountered.
- Each block’s `BlockInstanceSetup` function. `BlockInstanceSetup` is called for all instances of a given block type in the model.

The order in which these calls are made is significant, because the `BlockInstanceSetup` function may depend upon global variables that are initialized by the `BlockTypeSetup` function.

In Release 12.1, the `BlockTypeSetup` function is called before the `BlockInstanceSetup`. This corrects a problem in previous releases, where `BlockInstanceSetup` was erroneously called first. You may need to change your S-functions or block implementations if they depend upon the previous behavior.

Obsolete Code Generation Variables

The code generation variables `FunctionInlineType` and `PragmaInlineString` are now obsolete. These variables controlled the generation of inlined functions. In the current release, you can generate inlined functions from subsystems, as described in "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop User's Guide.

Real-Time Workshop 4.0 Release Notes

Release Summary

Release 4.0 of the Real-Time Workshop is a major upgrade, incorporating significant new and enhanced features and many improvements in the quality of generated code. These include:

- Significantly faster Target Language Compiler (TLC) code generation process
- TLC Profiler report for debugging TLC programs
- New efficiencies in generated code include improved signal storage reuse, constant block elimination, and parameter pooling.
- New Real-Time Workshop Embedded Coder add-on product replaces and significantly enhances the Embedded Real-Time (ERT) target.
- User interface improvements, including a redesigned Real-Time Workshop page and Model Parameter Configuration (tunable parameters) dialog
- Support for additional Simulink blocks, including Look-Up table blocks with very efficient generated code
- S-Function Target support for variable-step solvers and parameter tuning
- Support for matrix operations for most Simulink blocks
- Support for frame-based processing for DSP blocks
- External mode support for many additional block types for signal uploading
- Automatic generation of S-function wrappers for embedded code, allowing for validation of generated code in Simulink
- Support for generation of code and executables from subsystems
- Support for Simulink data objects in code generation
- Support for generation of ASAP2 data definition files

New Features

This section introduces the new features and enhancements added in the Real-Time Workshop 4.0 since the Real-Time Workshop 3.0.1.

Real-Time Workshop Embedded Coder

The Real-Time Workshop Embedded Coder is a new add-on product that replaces and enhances the Embedded Real-Time (ERT) target.

The Real-Time Workshop Embedded Coder is 100% compatible with the ERT target. In addition to supporting all previous functions of the ERT target, the Real-Time Workshop Embedded Coder includes many enhancements.

See the Real-Time Workshop Embedded Coder documentation for details.

Simulink Data Object Support

The Real-Time Workshop supports the new Simulink data objects feature. Simulink provides the built-in `Simulink.Parameter` and `Simulink.Signal` classes for use with the Real-Time Workshop. Using these classes, you can create parameter and signal objects and assign storage classes and storage type qualifiers to the objects. These properties control how the generated code represents signals and parameters. The `Simulink.Parameter` and `Simulink.Signal` classes can be extended to include user-defined properties.

See "Simulink Data Objects and Code Generation" in the Real-Time Workshop documentation for complete details.

ASAP2 Support

ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). This standard is used for data measurement, calibration, and diagnostic systems.

The Real-Time Workshop now lets you export an ASAP2 file containing information about your model during the code generation process. See "Generating ASAP2 Files" in the Real-Time Workshop documentation.

Enhanced Real-Time Workshop Page

The Real-Time Workshop page of the **Simulation Parameters** dialog has been reorganized and made easier to use. See “Configuring Real-Time Workshop Code Generation Parameters” in the Real-Time Workshop documentation for complete details.

Other User Interface Enhancements

The **Tools** menu of the Simulink window now contains a **Real-Time Workshop** submenu with shortcuts to frequently used features. See the Real-Time Workshop documentation for details.

You can now select a target configuration from the System Target File Browser by double-clicking on the desired entry in the target list. The previous selection method—selecting an entry and clicking **OK** — is still supported.

Advanced Options Page

An Advanced options page has been added to the **Simulation Parameters** dialog. The Advanced page contains new code generation options, as well as options formerly located in the Diagnostics and Real-Time Workshop pages.

Model Parameter Configuration Dialog

The **Model Parameter Configuration** dialog replaces the **Tunable Parameters** dialog. The **Model Parameter Configuration** dialog enables you to declare individual parameters to be tunable and to control the generated storage declarations for each parameter. See “Parameters: Storage, Interfacing, and Tuning” in the Real-Time Workshop documentation for details.

Tunable Expressions Supported

A tunable expression is an expression that contains one or more tunable parameters. Tunable expressions are now supported during simulation and in generated code.

Tunable expressions are allowed in masked subsystems. You can use tunable parameter names or tunable expressions in a masked subsystem dialog. When referenced in lower-level subsystems, such parameters remain tunable.

See “Tunable Expressions” in the Real-Time Workshop documentation for a detailed description of the use of tunable parameters in expressions.

S-Function Target Enhancements

S-function target enhancements include:

- The S-function target now supports variable-step solvers.
- The S-function target now supports tunable parameters.
- The new **Generate S-function** feature lets you automatically generate an S-function from a subsystem.

The S-function target is now documented in “The S-Function Target” in the Real-Time Workshop documentation.

External Mode Enhancements

Several new features have been added to external mode:

- The default operation of the **External Signal & Triggering** dialog has been changed to make monitoring the target program simpler. See “External Signal Uploading and Triggering” in the Real-Time Workshop documentation for details.
- Signal Viewing Subsystems have been implemented to let you encapsulate processing and viewing of signals received from the target system. Signal Viewing Subsystems run only on the host, generating no code in the target system. This is useful in situations where you want to process or condition signals before viewing or logging them, but you do not want to perform these tasks on the target system. See “Signal Viewing Subsystems” in the Real-Time Workshop documentation for details.
- Previously, only Scope blocks could be used in external mode to receive and view signals uploaded from the target program. The following now support external mode:
  - Dials & Gauges Blockset
  - Display blocks
  - To Workspace blocks

  - Signal Viewing Subsystems
  - S-functions

  See “External Mode Compatible Blocks and Subsystems” in the Real-Time Workshop documentation for details.

- In Release 12, the external mode communications application program interface (API) is documented. If you want to implement external mode communications via your own low-level protocol, see “Creating an External Mode Communication Channel” in the Real-Time Workshop documentation.

Build Directory

The Real-Time Workshop now creates a *build directory* within your working directory. The build directory stores generated source code and other files created during the build process. The build directory name, `model_target_rtw`, derives from the name of the source model and the chosen target.

See “Directories Used During the Build Process” in the Real-Time Workshop documentation for details.

Note If you have created custom targets for the Real-Time Workshop under Release 11, you must update your custom system target files and template makefiles to create and utilize the build directory. See “Updating Release 11 Custom Targets” on page 11-12.

Code Optimization Features

This section describes new or modified code generation options that are designed to help you optimize your generated code. The options described are located on the Advanced page of the **Simulation Parameters** dialog.

- **Block reduction**: When the **Block reduction** option is selected, Simulink collapses certain groups of blocks into a single, more efficient block, or removes them entirely. This results in faster model execution during simulation and in generated code.

- **Parameter pooling**: When multiple block parameters refer to storage locations that are separately defined but structurally identical, you can use this option to save memory.
- **Signal storage reuse**: This option replaces the (Enable/Disable) **Optimized block I/O storage** option of previous releases. **Signal storage reuse** is functionally identical to the older feature. Turning **Signal storage reuse** on is equivalent to enabling **Optimized block I/O storage**.

See "Optimizing a Model for Code Generation" in the Real-Time Workshop documentation for more information on code optimization.

Subsystem Based Code Generation

The Real-Time Workshop now generates code and builds an executable from any subsystem within a model. The build process uses the code generation and build parameters of the root model. See "Generating Code and Executables from Subsystems" in the Real-Time Workshop documentation for details.

Nonvirtual Subsystem Code Generation

The Real-Time Workshop now lets you generate code modules at the subsystem level. This feature applies only to nonvirtual subsystems. With nonvirtual subsystem code generation, you control how many files are generated, as well as the file and function names. To set options for nonvirtual subsystem code generation, you use the subsystem's **Block Parameters** dialog.

Nonvirtual subsystem code generation is a more general and flexible method of controlling the number and size of generated files than the **Function management** code generation options (**File splitting** and **Function splitting**) used in previous releases. The **Function management** code generation options have been replaced by nonvirtual subsystem code generation.

See "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop documentation for details.

Filename Extensions for Generated Files

In previous releases, some generated files were given special filename extensions, such as `.prm` or `.reg`. All the Real-Time Workshop generated code and header files now use standard filename extensions (`.c` and `.h`). The file naming conventions for the following generated files have changed:

- Model registration file (formerly `model.reg`) is now named `model_reg.h`.
- Model parameter file (formerly `model.prm`) is now named `model_prm.h`.
- `BlockIOSignals` structure file (formerly `model.bio`) is now named `model_bio.c`.
- `ParameterTuning` file (formerly `model.pt`) is now named `model_pt.c`.
- External mode data type transition file (formerly `model.dt`) is now named `model_dt.c`.

hilite_system and Code Tracing

The Real-Time Workshop writes system/block identification tags in the generated code. The tags are designed to help you identify the block, in your source model, that generated a given line of code. In previous releases, the `locate_system` command was used to trace a tag back to the generating block.

The new `hilite_system` command replaces `locate_system`, for the purposes of tracing the Real-Time Workshop identification tags. You should use the `hilite_system` command to trace a tag back to the generating block. For more information on identification tags and code tracing, see "Tracing Generated Code Back to Your Simulink Model".

Generation of Parameter Comments

The **Force generation of parameter comments** option in the **General code generation options** category of the Real-Time Workshop page controls the generation of comments in the model parameter structure (`rtP`) declaration in `model_prm.h`. This lets you reduce the size of the generated file for models with a large number of parameters.

Borland 5.4 Compiler Support

The Real-Time Workshop now supports Version 5.4 of the Borland C/C++ compiler.

Enhanced Makefile Include Path Rules

Two new rules and macros have been added to Real-Time Workshop template makefiles. These rules let you add source and include directories to makefiles generated by Real-Time Workshop without having to modify the template makefiles themselves. This feature is useful if you need to include your code when building S-functions.

Target Language Compiler 4.0

TLC File Parsing Before Execution

The Target Language Compiler 4.0 completes parsing of the TLC file just before execution. This aids development because syntax errors are caught the first time the TLC file is run instead of the first time the offending line is reached.

Enhanced Speed

The Target Language Compiler 4.0 features speed improvements throughout the software. In particular, the speed of block parameter generation has been enhanced.

Build Directory

The Target Language Compiler 4.0 creates and uses a `build` directory. The `build` directory is in the current directory and prevents generated code from clashing with other files generated for other targets, and keeps your model directories maintenance to a minimum.

TLC Profiler

An entirely new TLC Profiler has been added to the Target Language Compiler to help you find performance problems in your TLC code.

model.rtw Changes

This release contains a new format and changes to the *model*.rtw file. The size of the *model*.rtw file has been reduced.

Block Parameter Aliases

Aliases have been added for block parameters in the *model*.rtw file.

Improved Text Expansion

This release of the Target Language Compiler contains new, flexible methods for text expansion from within strings.

Column-Major Ordering

Two-dimensional signal and parameter data now use column-major ordering.

Improved Record Handling

The Target Language Compiler 4.0 utilizes new record data handling.

New TLC Language Semantics

Many changes have been made to the language including:

- Improved `EXISTS` behavior (see "TLC Compatibility Issues" on page 11-13)
- New TLC primitives for record handling
- Functions can return records.
- Records can be printed.
- Records can be empty.
- Record aliases are available.
- Records can be expanded with `%<>`.
- Built-in functions cannot be undefined via `%undef`.
- Short circuit evaluation for Boolean operators, `%if-%elseif-%endif`, and `?:` expressions are handled properly
- Conversions of values to and from MATLAB.

- Enhanced conversion rules for `FEVAL`. You can now pass records and structs to `FEVAL`.
- Relational operators can be used with nonfinite values.
- Loop control variables are local to loop bodies.

New Built-In Functions

The following built-in functions have been added to the language.

`FIELDNAMES`, `GENERATE_FORMATTED_VALUE`, `GETFIELD`, `ISALIAS`, `ISEMPTY`, `ISEQUAL`, `ISFIELD`, `REMOVEFIELD`, `SETFIELD`

New Built-In Values

The following built-in values have been added to the language.

`INTMAX`, `INTMIN`, `TLC_FALSE`, `TLC_TRUE`, `UINTMAX`

Added Support for Inlined Code

Support has been added for two-dimensional signals in inlined code.

Upgrading from an Earlier Release

This section discusses issues involved in upgrading from the Real-Time Workshop Version 3.0 to Real-Time Workshop Version 4.0.

Column-Major Matrix Ordering

The Real-Time Workshop now uses column-major ordering for two-dimensional signal and parameter data. In previous releases, the ordering was row-major.

If your hand-written code interfaces to such signals or parameters via `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` declarations, make sure to review any code that relies on row-major ordering, and make appropriate revisions.

Including Generated Files

Filename extensions for certain generated files have changed. If your application code uses `#include` statements to include the Real-Time Workshop generated files (such as `model.prm`), you may need to modify these statements. See "Files Created During Build Process" in the Real-Time Workshop documentation.

Updating Release 11 Custom Targets

If you have created custom targets for the Real-Time Workshop under Release 11, you must update your custom system target files and template makefiles to create and utilize the build directory. See `matlabroot/rtw/c/grt` for examples.

To update a Release 11 target:

1 Add the following to your system target file.

```
/%
BEGIN_RTW_OPTIONS
...
rtwgensettings.BuildDirSuffix = '_grt_rtw';
END_RTW_OPTIONS
%/
```

2 Add ".." to the INCLUDES rule in your template makefile. The following example is from grt_lcc.tmf.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(USER_INCLUDES)
```

The first -I. gets files from the build directory, and the second -I.. gets files (e.g., user written S-functions) from the current working directory.

Conceptually, think of the current directory and the build directory as the same (as it was in Release 11). The current working directory contains items like user written S-functions. The reason ".." must be added to the INCLUDES rule is that make is invoked in the build directory (i.e., the current directory was temporarily moved).

3 Place the generated executable in your current working directory. The following example is from grt_lcc.tmf.

```
PROGRAM = ../$(MODEL).exe
$(PROGRAM) : $(OBJS) $(RTWLIB)
 $(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(RTWLIB) $(LIBS)
```

hilite_system Replaces locate_system

If you use the locate_system command, in MATLAB programs for tracing the Real-Time Workshop system/block identification tags, you should use hilite_system instead.

TLC Compatibility Issues

In bringing Target Language Compiler files from Release 11 to Release 12, the following changes may affect your TLC code base:

- Nested evaluations are no longer supported. Expressions such as

```
%<LibBlockParameterValue(%<myVariable>,"", "", "")>
```

  are no longer supported. You will have to convert these expressions into equivalent non-nested expressions.

- Aliases are no longer automatically created for Parameter blocks while reading in the Real-Time Workshop files.

- You cannot change the contents of a "Default" record after it has been created. In the previous TLC, you could change a "Default" record and see the change in all the records that inherited from that default record.
- The `%codeblock` and `%endcodeblock` constructs are no longer supported.
- `%defines` and macro constructs are no longer supported.
- Use of line continuation characters (`...` and `\`) are not allowed inside of strings. Also, to place a double quote (`"`) character inside a string, you must use `\"`. Previously, the Target Language Compiler allowed you to use `"""` to get a double quote in a string.
- Semantics have been formalized to `%include` files in different contexts (e.g., from generate files, inside of `%with` blocks, etc.) `%include` statements are now treated as if they were read in from the global scope.
- The previous the Target Language Compiler had the ability to split function definitions (and other directives) across include file boundaries (e.g., you could start a `%function` in one file and `%include` a file that had the `%endfunction`). This no longer works.
- Nested functions are no longer allowed. For example,

```
%function foo ()
  %function bar ()
  %endfunction
%endfunction
```

- Built-in functions cannot be undefined via `%undef`. It is possible to undefine built in values, but this practice is not encouraged.
- Recursive records are no longer allowed. For example,

```
Record1   {
  Val    2
  Ref   Record2
}
Record2   {
  Val   3
  Ref   Record1
}
```

- Record declaration syntax has changed. The following code fragments illustrate the differences between declaring a record `recVar` in previous versions of the Target Language Compiler and the current release.
  - Previous versions:

    ```
    %assign recVarAlias = recVar { ...
     field1  value1 ...
     field2  value2 ...
     ...
     fieldN  valueN ...
    }
    ```

  - Current version:

    ```
    %createrecord recVar { ...
     field1  value1 ...
     field2  value2 ...
     ...
     fieldN  valueN ...
    }
    ```

- Semantics of the `EXISTS` function have changed. In the previous release of TLC, `EXISTS(var)` would check if the variable represented by the string value in `var` existed. In the current release of TLC, `EXISTS(var)` checks to see if `var` exists or not.

  To emulate the behavior of `EXISTS` in the previous release, replace

  ```
  EXISTS(var)
  ```

  with

  ```
  EXISTS("%<var>")
  ```