Real-Time Workshop®
Release Notes

Contents

Summary by Version

This table provides quick access to what's new in each version. For clarification, see "About Release Notes" on page 2.

Version (Release)	New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Latest Version V6.6.1 (R2007a+)	No	No	Bug Reports Includes fixes	Printable Release Notes: PDF No changes to documentation
V6.6 (R2007a)	Yes Details	No	Bug Reports Includes fixes	Current product documentation
V6.5 (R2006b)	Yes Details	Yes Summary	Bug Reports Includes fixes	No
V6.4.1 (R2006a+)	No	No	Bug Reports Includes fixes	No
V6.4 (R2006a)	Yes Details	Yes Summary	Bug Reports Includes fixes	No

Version (Release)	New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
V6.3 (R14SP3)	Yes Details	Yes Summary	Bug Reports Includes fixes	No
V6.2.1 (R14SP2+)	No	No	Bug Reports Includes fixes	No
V6.2 (R14SP2)	Yes Details	No	Bug Reports Includes fixes	No
V6.1 (R14SP1)	Yes Details	No	Fixed bugs	No
V6.0 (R14)	Yes Details	Yes Summary	Fixed bugs	No
V5.2 (R13SP2)	No	No	Fixed bugs	No
V5.1.1 (R13SP1+)	Yes Details	Yes Summary	Fixed bugs	No
V5.1 (R13SP1)	No	No	Fixed bugs	No
V5.0.1 (R13+)	Yes Details	No	Fixed bugs	No
V5.0 (R13)	Yes Details	Yes Summary	Fixed bugs	No
V4.1 (R12.1)	Yes Details	Yes Summary	Fixed bugs	No
V4.0 (R12)	Yes Details	Yes Summary	No bug fixes	No

About Release Notes

Use release notes when upgrading to a newer version to learn about new features and changes, and the potential impact on your existing files and practices. Release notes are also beneficial if you use or support multiple versions.

If you are not upgrading from the most recent previous version, review release notes for all interim versions, not just for the version you are installing. For example, when upgrading from V1.0 to V1.2, review the New Features and Changes, Version Compatibility Considerations, and Bug Reports for V1.1 and V1.2.

New Features and Changes

These include

- New functionality
- Changes to existing functionality
- Changes to system requirements (complete system requirements for the current version are at the MathWorks Web site)
- Any version compatibility considerations associated with each new feature or change

Version Compatibility Considerations

When a new feature or change introduces a known incompatibility between versions, its description includes a **Compatibility Considerations** subsection that details the impact. For a list of all new features and changes that have compatibility impact, see the "Compatibility Summary for Real-Time Workshop" on page 170.

Compatibility issues that become known after the product has been released are added to Bug Reports at the MathWorks Web site. Because bug fixes can sometimes result in incompatibilities, also review fixed bugs in Bug Reports for any compatibility impact.

Fixed Bugs and Known Problems

MathWorks Bug Reports is a user-searchable database of known problems, workarounds, and fixes. The MathWorks updates the Bug Reports database as new problems and resolutions become known, so check it as needed for the latest information.

Access Bug Reports at the MathWorks Web site using your MathWorks Account. If you are not logged in to your MathWorks Account when you link

to Bug Reports, you are prompted to log in or create an account. You then can view bug fixes and known problems for R14SP2 and more recent releases.

The Bug Reports database was introduced for R14SP2 and does not include information for prior releases. You can access a list of bug fixes made in prior versions via the links in the summary table.

Related Documentation at Web Site

Printable Release Notes (PDF). You can print release notes from the PDF version, located at the MathWorks Web site. The PDF version does not support links to other documents or to the Web site, such as to Bug Reports. Use the browser-based version of release notes for access to all information.

Product Documentation. At the MathWorks Web site, you can access complete product documentation for the current version and some previous versions, as noted in the summary table.

Version 6.6.1 (R2007a+) Real-Time Workshop

This table summarizes what's new in V6.6.1 (R2007a+):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
No	No	Bug Reports Includes fixes	Printable Release Notes: PDF No changes to documentation: Current product documentation (V6.6)

Version 6.6 (R2007a) Real-Time Workshop

This table summarizes what's new in Version 6.6 (R2007a):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	Bug Reports Includes fixes	Current product documentation

New features and changes introduced in this version are

Static File Dependencies Reduced For Improved Integration and Builds

Real-Time Workshop provides additional source files and functions for use in building your code in the *matlabroot*/rtw/c/libsrc directory. During code generation, these files are added to the build process. This release removes over 200 source files from the libsrc directory. Instead, these functions and files are generated only when needed. This reduces the number of additional source files required to compile and build the code, which improves compile time and can simplify code integration and verification.

Support for Simulink Legacy Code Tool Enhancements

Real-Time Workshop supports the following Legacy Code Tool enhancements:

- New fields in the Legacy Code Tool data structure: `InitializeConditionsFcnSpec` and `SampleTime`. `InitializeConditionsFcnSpec` defines a function specification for a reentrant function that the S-function calls to initialize and reset states. `SampleTime` allows you to specify whether sample time is inherited from the source block, represented as a tunable parameter, or fixed.
- Support for state (persistent memory) arguments in registered function specifications.
- Support for complex numbers specified for input, output, and parameter arguments in function specifications. This support is limited to use with Simulink built-in data types.
- Support for multidimensional arrays specified for input and output arguments in function specifications. Previously, multidimensional array support applied to parameters only.
- Ability to apply the `size` function to any dimension of function input data—input, output, parameter, or state. The data type of the `size` function's return value can be any type except complex, bus, or fixed-point.
- A new Legacy Code Tool option, `'backward_compatibility'`, which you can specify with the `legacy_code` function. This option enables Legacy Code Tool syntax, as made available from MATLAB Central in releases before R2006b, for a given MATLAB session.
- The following new demos:

```
rtwdemo_lct_sampletime
rtwdemo_lct_work
rtwdemo_lct_cplxgain
rtwdemo_lct_ndarray
```

For more information, see

- "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" in the Writing S-Functions documentation
- "Using the Legacy Code Tool to Automate the Generation of Files for Fully Inlined S-Functions" in the Real-Time Workshop documentation
- `legacy_code` function reference page

New Target Language Compiler Tutorial

The Target Language Compiler (TLC) has been updated and includes a new tutorial that provides new users with an introduction to TLC syntax. See "Target Language Compiler Tutorials" for more information.

Code Generation for Multidimensional Signals

Real-Time Workshop now supports code generation for multidimensional signals. For more information, see "Multidimensional Signals" in the Simulink release notes.

Enhanced Checking and Reporting for Identifier Conflicts

Real-Time Workshop supports the following identifier conflict enhancements:

- Performance improvement: For large models, Real-Time Workshop reviews 100 times more identifiers while delivering the same performance as in R2006a.
- MISRA compliance: All model reference identifiers and all reused subsystem identifiers comply with MISRA guidelines.
- Conflict detection: A more unified conflict detecting and reporting mechanism checks more identifiers, providing more information when conflicts occur.

Enhanced Support for Tunable Parameters in Expressions

Expressions that index into tunable parameters, such as `P(1)+P(2)/P(i)`, retain their tunability in generated code, including simulation code that is generated for a referenced model. Both the indexed parameter and the index itself can be tuned.

Parameter expressions of the form P(i) retain their tunability if all of the following are true:

- The index `i` is a constant or variable of `double` datatype
- `P` is a 1D array, or a 2D array with one row or one column, of `double` datatype
- `P` does not resolve to a mask parameter, but to a variable in the model or the base workspace

New Loss of Tunability Diagnostic

Previously, any loss of tunability generated a warning. In R2007a, you can use the **Loss of Tunability** diagnostic to control whether loss of tunability is ignored or generates a warning or error. See “Detect loss of tunability” for details.

Support for Microsoft Visual C++ Express Edition

Real-Time Workshop now supports using Microsoft Visual C++ Express Edition for compilation.

Enhanced Code Efficiency, Including Merge Block Optimizations

The Merge block adds support for dead code elimination of its inputs. For any Merge block input port that is connected to an unconditionally executed block, dead code elimination removes the calculation of unused values.

Reporting of Unconnected Signal Generators

During code generation, the Signal & Scope Manager reports unconnected signal generators.

Real-Time Workshop Profiling Works with Referenced Models

You can now use Real-Time Workshop Profiling with referenced models, which was not possible in previous releases.

New Makefile Command Controls Location and Naming of Model Reference Libraries

Except on the Macintosh platform, you can use the makefile command `USE_MDL_LIBPATHS` to change the default location and naming that Real-Time Workshop uses for model reference libraries. See "Controlling the Location and Naming of Model Reference Libraries" for details.

New TMF Token MODELREF_LINK_RSPFILE Supports Linking Response Files for Model Reference

R2007a adds the template makefile (TMF) token `MODELREF_LINK_RSPFILE` for use with model reference. The `MODELREF_LINK_RSPFILE` token for the top-level model expands to the name of a response file that the top-level model links against. This token is valid only for build environments that support linker response files.

`MODELREF_LINK_RSPFILE` offers a preferred alternative to the token `MODELREF_LINK_LIBS`, which expands to a list of referenced model libraries that the top-level model links against, provided that the linker supports response files.

For more information, see "Making Custom Targets Compatible with Model Reference" in the Real-Time Workshop documentation.

New and Enhanced Demos

The following demos have been added:

Demo...	Shows How You Can...
`rtwdemo_lct_sampletime`	Specify whether sample time is inherited from the source block, represented as a tunable parameter, or fixed
`rtwdemo_lct_work`	Use state arguments in registered function specifications
`rtwdemo_lct_cplxgain`	Specify complex numbers for input, output, and parameter arguments in function specifications.
`rtwdemo_lct_ndarray`	Specify multi-dimensional arrays for input and output arguments in function specifications.

Version 6.5 (R2006b) Real-Time Workshop

This table summarizes what's new in Version 6.5 (R2006b):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports Includes fixes	No

New features and changes introduced in this version are

Support for Simulink Report Generator

Real-Time Workshop has added support for the Simulink® Report Generator. You can use the Report Generator to document a code generation project in a variety of output formats: Rich Text Format (RTF), Extensible Markup Language (XML), and Hypertext Markup Language (HTML). By using the Real-Time Workshop `codegen.rpt` setup file and the components Code Generation Summary and Import Generated Code, you can generate a report that includes

- Model name and version
- Real-Time Workshop version
- List of generated source and header (include) files
- Optimization and Real-Time Workshop target selection and build process configuration settings
- Mapping of subsystem numbers to subsystem labels for models that include subsystems
- Listings of generated and custom code for the model

To see how the Report Generator fits into the general Real-Time Workshop workflow, see "Real-Time Workshop Workflow". For details on using the Report Generator with Real-Time Workshop, see the tutorial in "Documenting a Code Generation Project". See `rtwReport` for a description of the Real-Time Workshop function for generating a report from the MATLAB® command line or from a script. For descriptions of the new Real-Time Workshop Report Generator components, see Code Generation Summary and Import Generated Code in the MATLAB Report Generator documentation. For details on using Report Generator, see the MATLAB Report Generator documentation.

New Pack-and-Go Utility

Real-Time Workshop V6.5 (R2006b) introduces a new pack-and-go utility that you can use to relocate static and generated code files for a model to another development environment, such as a secure system or an integrated development environment (IDE) that does not include MATLAB and Simulink. This utility uses the tools for customizing post code generation build processing and a `packNGo` function to find and package all files needed

to build an executable image for a model into a compressed file that you can relocate and unpack using a standard zip utility.

For more information on how to use the new utility, see "Relocating Code to Another Development Environment".

Support for New Simulink.SubSystem.getChecksum Command for Determining Why Subsystem Code Is Not Reused

V6.5 (R2006b) Real-Time Workshop supports a new `Simulink.SubSystem.getChecksum` command that you can use to determine why subsystem code is not reused.

For a discussion on the code generation aspects of this feature, see "Determining Why Subsystem Code Is Not Reused" in the Real-Time Workshop documentation. For a description of the command, see `Simulink.SubSystem.getChecksum` in the Simulink documentation.

Merge Block Input Signals Can Have Storage Classes

In previous releases, an input signal connected to a Merge block could not specify any storage class except `Auto`. In R2006b, an input signal connected to a Merge block can specify a non-`Auto` storage class. This class must be the same for every non-`Auto` input or output signal connected to the block.

For information about the Merge block, see the Merge reference page in the Simulink documentation. For details about using Merge block signal storage classes in generated code, see the bullet for the Merge block under Other Optimization Tools and Techniques in the Real-Time Workshop documentation.

Code Formatting Consistency Improvements

Real-Time Workshop is enhanced to apply a more consistent formatting style to the code that it generates, including integrated custom code. The new style rules for code generation formatting pertain to

- Comments
- Blank lines and spaces
- Indentation
- Line breaks
- International characters
- `#define` statements
- Function definitions
- Branching and looping structural statements
- Structure definition and value initialization statements

Compatibility Considerations

No code generation readability enhancement affects the syntax or semantics of the code. Such changes affect only the physical code appearance. Therefore, applications and correctly engineered automated tests should be unaffected by any readability enhancements. However, automated tests that depended on the physical code layout may be affected. You should change such tests to operate only on functional properties of the code.

Support for Simulink Legacy Code Tool

Real-Time Workshop provides code generation support for the new Simulink Legacy Code Tool. This tool speeds the creation of S-functions from legacy C or C++ code. For a list of related demos, see “New and Enhanced Demos” on page 18.

New findIncludeFiles Function

A new `findIncludeFiles` function has been added to the build information API. This function searches for include (header) files in all source and include

paths recorded in a model's build information object and adds the files found, along with their full paths, to the build information object.

For information on how to use the functions in the build information API, see "Customizing Post Code Generation Build Processing".

Show eliminated statements Model Configuration Option Renamed

The **Show eliminated statements** option on the **Real-Time Workshop > Comments** pane of the Configuration Parameters dialog box has been renamed to **Show eliminated blocks**. The corresponding configuration parameter `ShowEliminatedStatement` is not affected by this change.

Change to Default Settings for Multitasking Diagnostic Options

For new models created with Simulink V6.5 (R2006b) or later, the default value for the following multitasking diagnostic options on the Configuration Parameters dialog is set to `error` to avoid generation of code that might corrupt data or produce unpredictable behavior.

- **Diagnostics > Sample Time > Multitask conditionally executed subsystem**
- **Diagnostics > Data Validity > Data Store Memory Block > Multitask data store**

For more information about these multitasking diagnostic options, see in the Simulink documentation.

Compatibility Considerations

For models created with a version of Simulink before V6.5 (R2006b), when the following conditions exist, Simulink displays a dialog box that reports the issue and suggests that you change the diagnostic setting to `error`, as recommended by the Model Advisor.

- The **Tasking mode for periodic sample times** option on the Configuration Parameters **Solver** pane is set to `MultiTasking` or the model includes an asynchronous task.
- The diagnostic setting that corresponds to the multitasking issue is set to a value other than `error`.
- You use Real-Time Workshop to generate code for the model.
- Real-Time Workshop detects a multitasking issue that might corrupt data or produce unpredictable behavior.

The dialog presents you with the following options.

Button	Action
Change	Change the diagnostic setting to `error`.
Ignore	Leave the diagnostic setting as is.
Always ignore	Leave the diagnostic setting as is and do not show the dialog box again.

PreLookup Index Search and Interpolation (n-D) Using PreLookup Block Changes

Real-Time Workshop no longer uses the runtime library functions in `c/libsrc` to generate code for the PreLookup Index Search and Interpolation (n-D) Using PreLookup blocks. Instead, Real-Time Workshop generates the lookup functions for the two blocks dynamically. In addition, if you enable the **Utility function generation** option on the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box, the lookup functions are stored as shared utility functions.

Character Patterns You Should Not Use in Block Names

To avoid possible ambiguity in the comments in generated code, do not use the following character patterns in block names:

- /*
- */

New and Enhanced Demos

New demos are

Demo...	Shows How You Can...
`rtwdemo_codegenrpt`	Use the Simulink Report Generator to automatically document code you generate with Real-Time Workshop.

Collection of custom code demos:
 	
`rtwdemo_lct_bus`
`rtwdemo_lct_cpp`
`rtwdemo_lct_filter`
`rtwdemo_lct_fixpt_params`
`rtwdemo_lct_fixpt_signals`
`rtwdemo_lct_gain`
`rtwdemo_lct_inherit_dims`
`rtwdemo_lct_lut`
`rtwdemo_lct_start_term`
	Call external legacy C and C++ functions (existing or custom code) that are external to a Simulink model. The demos use an unsupported Legacy Code Tool application programming interface (API) to register a legacy function prototype and create necessary files for an S-function that calls the specified legacy function during simulation and in generated code. The Legacy Code Tool allows you to:
	
	• Provide the legacy function specification
	• Generate a C-MEX S-function for simulation
	• Generate a block TLC file and optional `rtwmakecfg.m` file that is used during code generation to call the legacy code

Version 6.4.1 (R2006a+) Real-Time Workshop

This table summarizes what's new in V6.4.1 (R2006a+):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
No	No	Bug Reports at Web site	No

Version 6.4 (R2006a) Real-Time Workshop

This table summarizes what's new in V6.4 (R2006a):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports at Web site	No

New features and changes introduced in this version are

New Build Information Application Program Interface

V6.4 (R2006a) Real-Time Workshop® introduces an application program interface (API) for populating and managing all build information associated with a given model in a single source. This feature

- Provides a mechanism for defining build information for tool chains that do not use make files
- Makes it easier to customize and maintain a model's build information

The API includes methods for adding, managing, and retrieving:

- Compiler flags
- Preprocessor identifier definitions
- Link flags
- Include files and paths
- Source files and paths
- Libraries

The API also includes methods for updating file paths, extensions, and separators.

For information on how to use the API, see the demo `rtwdemo_buildInfo` and "Customizing Post Code Generation Build Processing" in the Real-Time Workshop documentation. For descriptions of the API methods, see "Functions — By Category" and "Functions — Alphabetical List".

New Mechanism for Customizing Post Code Generation Build Processing

Starting with V6.4 (R2006a), you can customize the Real-Time Workshop build process to evaluate a post code generation command after generating and writing the model's code to disk and before generating a makefile. A post code generation command is a user-defined M-file that typically calls functions to get data from or add data to the model's build information object.

This feature is useful for applications that need to control various aspects of the build process after code generation. For example, this is necessary when you develop your own target, or you want to apply an analysis tool to the generated code before continuing with the build process.

To use this feature, you program the command as a script or function and then define the command with the new `PostCodeGenCommand` model configuration parameter.

For more information, see the demo `rtwdemo_buildInfo` and "Customizing Post Code Generation Build Processing" in the Real-Time Workshop documentation.

New Model Configuration Option for Suppressing Makefile Generation

V6.4 (R2006a) adds a new option to the **Real-Time Workshop** pane of the Configuration Parameters dialog box and a corresponding model configuration parameter, `GenerateMakefile`, which you can use to suppress makefile generation during the build process. For example, you might do this to integrate tools into the build process that are not driven by makefiles.

This option controls whether Real-Time Workshop generates a makefile during the build process and is selected by default. If you clear the check box in the graphical user interface or set the parameter to `off`, Real-Time Workshop does not generate a makefile for the model. When you suppress makefile generation, you must specify any post code generation processing, including compilation and linking, as a command you program and define, using the feature described in "New Mechanism for Customizing Post Code Generation Build Processing" on page 22.

For more information, see “Customizing Post Code Generation Build Processing”.

New RSim Target Option for Feeding Inport Blocks with MAT-File Data

The RSim target is enhanced with a new `-i` command line option that allows you to feed an Inport block with input data during simulation from a single MAT-file or you can change the MAT-file from one simulation to the next. The format requirements of the MAT-file data are flexible in that it can be a single time/data matrix, a single structure, or multiple structures.

For details on how to set up a MAT-file for use with an Inport block and specify signal data for an Inport block, see the demo `rtwdemo_rsim_i` and “Creating a MAT-File for an Inport Block” and “Specifying Signal Data File for an Inport Block” in the Real-Time Workshop documentation.

Switch Block Optimization for Wide Control Port Signals

In releases prior to V6.4 (R2006a), Real-Time Workshop optimized code generated for a Switch block such that the code for blocks connected to the data input ports executed conditionally. This optimization was limited to Switch blocks with a control port receiving scalar signals. V6.4 (R2006a) enhances Real-Time Workshop to generate code that performs conditional branch execution whether the Switch block’s control port signal is a scalar value, a vector, or a matrix.

For a description of the Switch block, see Switch in the Simulink® Reference.

Multiport Switch Block Enhanced to Generate Default Switch Case Statement

In V6.4 (R2006a), Real-Time Workshop is enhanced to generate a default switch case statement for the Mulitport Switch block. For a description of this block, see Multiport Switch in the Simulink Reference.

C++ Language Support Enhancements

V6.4 (R2006a) Real-Time Workshop adds support for C++ code generation for Signal Processing Blockset and Video and Image Processing Blockset products.

Limitations

- Microsoft Visual C/C++, GNU C/C++, Watcom C/C++ and Borland® C/C++ compilers have been fully tested with V6.4 (R2006a) Real-Time Workshop and are fully supported on 32-bit Windows and 32/64-bit Linux platforms. However, V6.4 (R2006a) provides Beta C++ support only for the Intel® C/C++ compiler, which has 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 other 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, the language of the code generated for the top model and any referenced models must match. For example, if you generate C++ code for the top model, the generated code for all referenced models must also be C++ code.

- 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 Simulink Signal Object Initialization

V6.4 (R2006a) introduces the ability to initialize Simulink signal objects with user-defined values for simulation and code generation. Data initialization increases application reliability and is a requirement of safety critical applications. Initializing signals for both simulation and code generation can expedite transitions between phases of Model-Based Design.

For details on using this feature, see the demo `rtwdemo_sigobj_iv`, "Using Signal Objects to Initialize Signals and Discrete States" in the Simulink documentation, and "Using Signal Objects to Initialize Signals and Discrete States" in the Real-Time Workshop documentation.

Compatibility Considerations

In general, if a submodel uses workspace variables and the variables change, Real-Time Workshop rebuilds the submodel. This behavior also occurs if the initial value for a signal object that corresponds to a signal initialized from outside the model, such as a global data store or root input port, changes.

To work around this behavior, specify the signal object's initial value as a tunable parameter. For example:

```
S = Simulink.Signal;
S.InitialValue = 'K';
K = Simulink.Parameter;
K.Value = 4;
K.RTWInfo.StorageClass = 'ExportedGlobal';
```

You can then use the tunable parameter to change the signal's initial value without triggering a subsystem build.

Identifiers and Model Reference Applications

As or Version 6.4 (R2006a), to avoid name clashes in models that use model referencing, do one of the following:

- Increase the maximum identifier length setting for top and referenced models until the following warning disappears:

  ```
  "Warning: Insufficient space for computing symbol names in
  model ...",
  ```

  In this case, uniqueness of model names ensures that the names do not clash.

- If you have a Real-Time Workshop Embedded Coder license, you can define a unique symbol naming scheme for each model. For example, you might define `'m1$R$N$M'` for the first model, `'m2$R$N$M'` for the second model, and so forth. The uniqueness of the naming scheme prevents name clashing.

Support for Simulink Parameter Object Data Type Enhancements

V6.4 (R2006a) Real-Time Workshop supports the following Simulink parameter object data type enhancements discussed in "Data Type Property of Parameter Objects Now Settable" and "Range-Checking for Parameter and Signal Object Values" in the Simulink Release Notes.

- Support for fixed-point data types
- Ability to specify the data type attribute independently of the object's value attribute

For a discussion on the code generation aspects of this enhancement, see the demo `rtwdemo_paramdt` and "Generated Code for Parameter Data Types" in the Real-Time Workshop documentation.

Support for New Simplest Rounding Mode for Fixed-Point Simulink Blocks

V6.4 (R2006a) Real-Time Workshop supports the new `Simplest` rounding mode that is available for the **Round integer calculations toward**

parameter of some fixed-point Simulink blocks. This rounding mode attempts to reduce or eliminate the need for extra rounding code in generated code. The `Simplest` rounding mode is currently available for the following blocks:

- Data Type Conversion
- Product
- Lookup Table
- Lookup Table (2-D)
- Lookup Table Dynamic

For more information, see "Rounding" in the Simulink Fixed Point documentation.

Name Change for PrevZC Identifier in Generated Code

In earlier releases, the identifier generated for a data item representing previous zero-crossing signal states (type `PrevZCSigStates_`*`model`*) was named inconsistently. Depending on your target configuration, the identifier could be generated as *`model`*`_PrevZCSigState` or *`model`*`_PrevZC`. In V6.4 (R2006a), the identifier is generated as *`model`*`_PrevZCSigState` across all configurations. For example, the following would appear in generated C code for a model named `mydemo` (for which zero-crossing data is relevant):

```
/* Previous zero-crossings (trigger) states */
PrevZCSigStates_mydemo mydemo_PrevZCSigState;
```

Format Enhancements for model.rtw File

Starting in V6.4 (R2006a), Real-Time Workshop represents data type information in the file *`model`*`.rtw` in a more compact format. This new format omits the fields `ComplexSignal`, `DataTypeIdx` , `Dimensions`, and `Width` from where they occurred in the following records.

Record	Record Type	Fields Removed

`BlockOutputs`	Block output ports	`ComplexSignal`
		`DataTypeIdx`
		`Width`

`Dworks`	Block Dworks	`ComplexSignal` `Width`

`ExternalInputs`	External inputs	`ComplexSignal`
		`DataTypeIdx`
		`Width`

`ExternalOutputs`	External outputs	`Width`

`ModelParameters`	Model parameters	`ComplexSignal`
		`DataTypeIdx`
		`Dimensions`
		`Width`

The following topics discuss

- "New Target Language Compiler Library Functions That Support the New File Format" on page 28
- "Compatibility Considerations" on page 29

New Target Language Compiler Library Functions That Support the New File Format

In support of the new file format, V6.4 (R2006a) adds the following new Target Language Compiler (TLC) library functions for gaining access to the `ComplexSignal`, `DataTypeIdx`, `Dimensions`, and `Width` fields for a given record. You can use the new functions with the new and old file formats.

Function	Description
`LibGetRecordIsComplex(rec)`	Returns the value 1 if the specified record is complex, and 0 otherwise.
`LibGetRecordDataTypeId(rec)`	Returns the data type identifier of the specified record as a an integer.
`LibGetRecordDimensions(rec)`	Returns the dimensions of the specified record as a vector of integers.
`LibGetRecordWidth(rec)`	Returns the width of the specified record as an integer.

Compatibility Considerations

The Target Language Compiler (TLC) includes library functions for retrieving data from fields of the *`model`*`.rtw` file. If your application retrieves data from *`model`*`.rtw` directly, that is, without using the documented TLC library functions, the application will be incompatible and will produce incorrect results. In such cases, reprogram your application to use the documented TLC library functions to retrieve data from *`model`*`.rtw`.

The following table lists the fields now omitted from *`model`*`.rtw` and the TLC library functions you can use to gain access to the fields for various types of records.

Field	Record Type	TLC Functions
`ComplexSignal`	Block input port	`LibBlockInputSignalIsComplex` `LibGetRecordIsComplex`
	Block output port	`LibBlockOutputSignalIsComplex` `LibGetRecordIsComplex`
	Block parameter	`LibBlockParameterIsComplex` `LibGetRecordIsComplex`

	Block Dwork	`LibBlockDWorkIsComplex`
		`LibGetRecordIsComplex`
		(Both functions return 1 or 0, which map to the old values `'yes'` and `'no'`, respectively.)

Field	Record Type	TLC Functions
DataTypeIdx	Block input port	LibBlockInputSignalDataTypeId LibGetRecordDataTypeId
	Block output port	LibBlockOutputSignalDataTypeId LibGetRecordDataTypeId
	Block parameter	LibBlockParameterDataTypeId LibGetRecordDataTypeId
	Block Dwork	LibBlockDWorkDataTypeId LibGetRecordDataTypeId
Dimensions	Block input port	LibBlockInputSignalDimensions LibGetRecordDimensions
	Block output port	LibBlockOutputSignalDimensions LibGetRecordDimensions
	Block parameter	LibBlockParameterDimensions LibGetRecordDimensions
Width	Block input port	LibBlockInputSignalWidth LibGetRecordWidth
	Block output port	LibBlockOutputSignalWidth LibGetRecordWidth
	Block parameter	LibBlockParameterWidth LibGetRecordWidth
	Block Dwork	LibBlockDWorkWidth LibGetRecordWidth

For descriptions of the new functions `LibGetRecordIsComplex`, `LibGetRecordDataTypeId`, `LibGetRecordDimensions`, and `LibGetRecordWidth`, see "New Target Language Compiler Library Functions That Support the New File Format" on page 28. For descriptions of other functions listed in the preceding table, see "TLC Function Library Reference" in the Real-Time Workshop Target Language Compiler documentation.

Changes to TLC Files in matlabroot/rtw/c/tlc

TLC files in the directory *matlabroot*/rtw/c/tlc have changed.

You should not customize TLC files in this directory even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Compatibility Considerations

Customizations to the files in *matlabroot*/rtw/c/tlc are not compatible across releases. If you have customized TLC files that reside in that directory, you must reapply your customizations when you upgrade.

New and Enhanced Demos

New demos are

Demo...	Shows How You Can...
rtwdemo_buildInfo	Customize post code generation build processing by using the new build information API and new post code generation command
rtwdemo_paramdt	Control the data type of tunable parameters in code that Real-Time Workshop generates
rtwdemo_rsim_i	Use the new -i RSim target option to feed Inport blocks with MAT-file data
rtwdemo_sigobj_iv	Initialize Simulink signal objects with the new Simulink signal object initialization feature

The following demos have been enhanced:

- rtwdemo_rsim_batch_script

Documentation Enhancements

- New reference documentation — Real-Time Workshop Reference
- New tables that summarize dependencies of optimization and interface model configuration parameters
- “Running Rapid Simulations” — reorganized to reflect workflow

Version 6.3 (R14SP3) Real-Time Workshop

This table summarizes what's new in V6.3 (R14SP3):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Bug Reports at Web site	No

New features and changes introduced in this version are

New rtw_precompile_libs Function

V6.3 (R14SP3) Real-Time Workshop 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

V6.3 (R14SP3) 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, see the description of the Inport block.

Support for Variable Transport Delay Enhancements

V6.3 (R14SP3) Real-Time Workshop supports new Simulink enhancements to the Variable Transport Delay block. Prior to V6.3 (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

V6.3 (R14SP3) Real-Time Workshop 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 V6.3 (R14SP3) Real-Time Workshop, 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 matches that of Linux systems.

Prior to V6.3 (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.

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.

Compatibility Considerations

Customizations that you make to built-in Simulink blocks 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 V6.2 (R14SP2) and earlier recommends using the `rtwgen` and `tlc` commands together to create targets and generate code. Instead, you should use the `slbuild` command.

Compatibility Considerations

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 that uses `rtwgen` should change to use `slbuild` instead, 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.

Model Hardware Configuration Parameters Now Honor Device Type Restrictions

Prior to V6.3 (R14SP3), Real-Time Workshop allowed you to use `set_param` to modify model hardware configuration settings such that they did not conform to device type restrictions. In V6.3 (R14SP3), Real-Time Workshop honors device type requirements associated with the following configuration parameters:

```
ProdBitPerLong
ProdBitPerChar
ProdBitPerInt
ProdBitPerShort
ProdIntDivRoundTo
ProdShiftRightIntArith
ProdWordSize
ProdEndianess
TargetBitPerLong
TargetBitPerChar
TargetBitPerInt
TargetBitPerShort
TargetIntDivRoundTo
TargetShiftRightIntArith
TargetWordSize
TargetEndianess
```

If you attempt to reset of one of these parameters, Real-Time Workshop returns an error.

Compatibility Considerations

If you set model parameters programmatically, check for and remove instances of `set_param` that specify the preceding parameters.

rem Function No Longer Supports Tunable Arguments

In V6.3 (R14SP3), the `rem` function no longer supports tunable parameters when used with Real-Time Workshop.

Compatibility Considerations

If you use tunable parameters with the `rem` function, Real-Time Workshop inlines the equivalent numeric value into the generated code in place of the tunable expression.

Block Libraries, RSim Target Executables, and MAT-Files

You can use Real-Time Workshop to rebuild an RSim target executable for a model that you previously created with blocks provided in an earlier release.

Compatibility Considerations

Block libraries for V6.3 (R14SP3) are not compatible with block libraries provided in earlier releases. Consequently, starting with V6.3 (R14SP3) , if you rebuild an RSim target executable as noted above, you cannot use the `-p` option to run the rebuilt executable with a new MAT-file. To use new MAT-files, you need to replace the blocks in the model with blocks provided in the R14SP3 block libraries

Documentation Enhancements

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

- **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

- 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 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 listing Real-Time Workshop and Real-Time Workshop Embedded Coder block support for blocks available in Simulink
- Target Language Compiler documentation

Version 6.2.1 (R14SP2+) Real-Time Workshop

This table summarizes what's new in V6.2.1 (R14SP2+):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
No	No	Bug Reports at Web site	No

Version 6.2 (R14SP2) Real-Time Workshop

This table summarizes what's new in V6.2 (R14SP2):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	Bug Reports at Web site	No

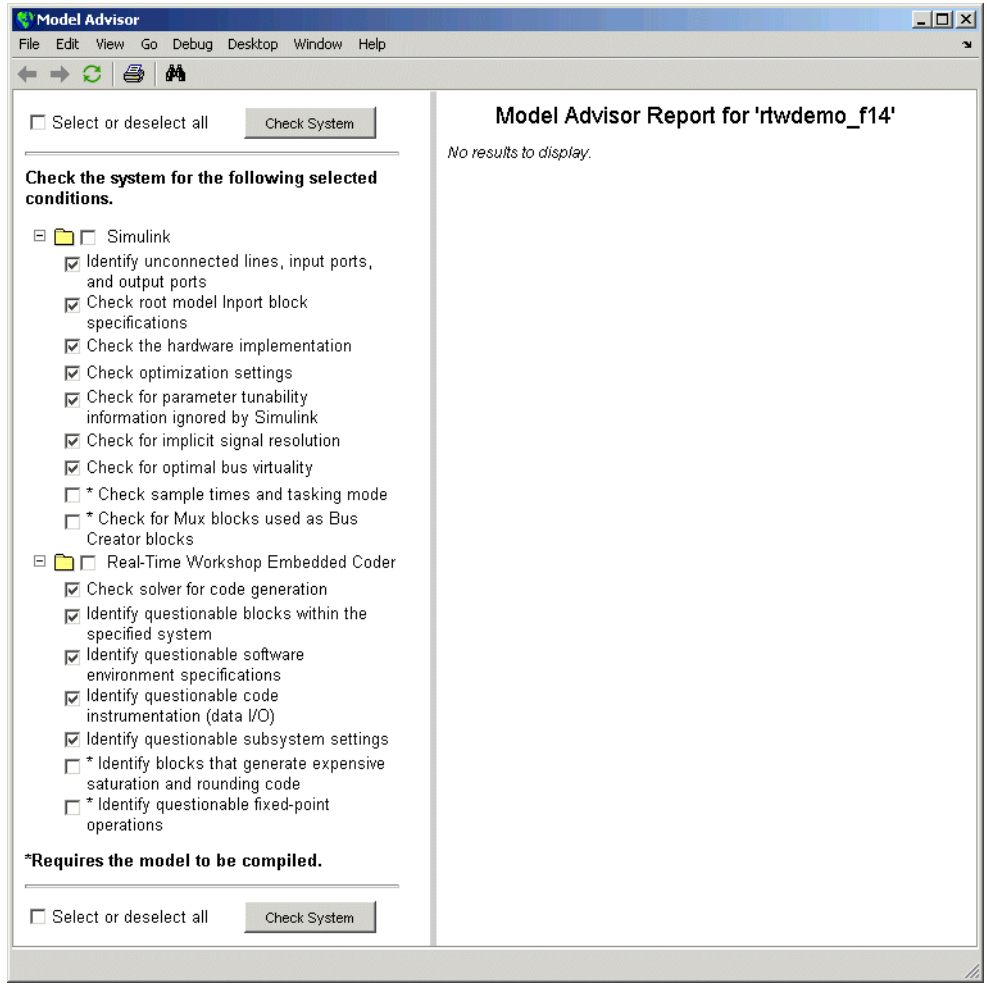
New features and changes introduced in this version are

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 V6.2 (R14SP2) 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:



For more information on the Model Advisor, see “Consulting Model Advisor” in the Simulink documentation.

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 V6.2 (R14SP2), 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 V6.2 (R14SP2), 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 V6.2 (R14SP2), the code generated for this block would copy the value of the block to a temporary variable. V6.2 (R14SP2) 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

V6.2 (R14SP2 Real-Time Workshop 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++.

For information on using this feature, see the following topics in the Real-Time Workshop documentation:

- “Choosing and Configuring a Compiler”
- “Language”
- “Integrating C and C++ Code”

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

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

V6.2 (R14SP2) 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, **Remove code from floating-point to integer conversions that wraps out-of-range values**, 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.

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

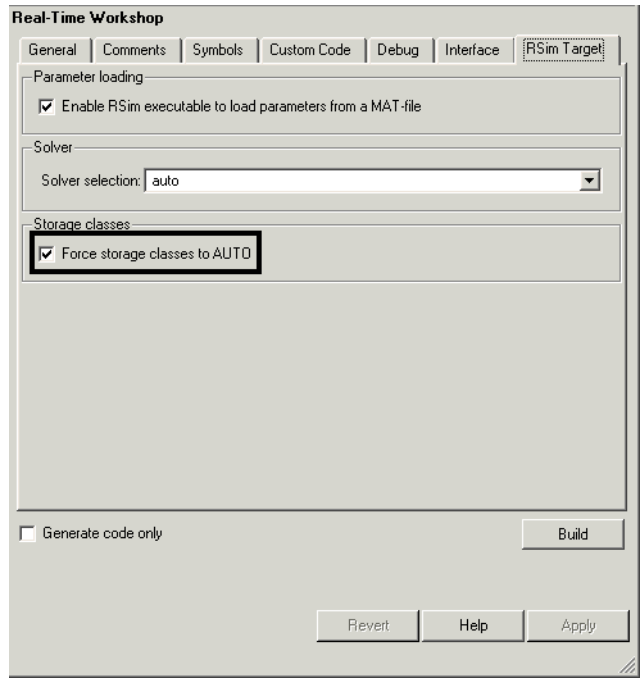
For more information, see "Remove Code from Floating-Point to Integer Conversions That Wraps Out-of-Range Values" in the Real-Time Workshop documentation.

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 in the next figure.



For more information, see “Configuring and Building a Model for Rapid Simulation”.

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 Enhancements

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

- 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

Version 6.1 (R14SP1) Real-Time Workshop

This table summarizes what's new in V6.1 (R14SP1):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	Fixed bugs	No

Changes from the Previous Release

The behavior of the 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.

Version 6.0 (R14) Real-Time Workshop

This table summarizes what's new in V6.0 (R14):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Fixed bugs	No

New features and changes introduced in this version are organized by these topics:

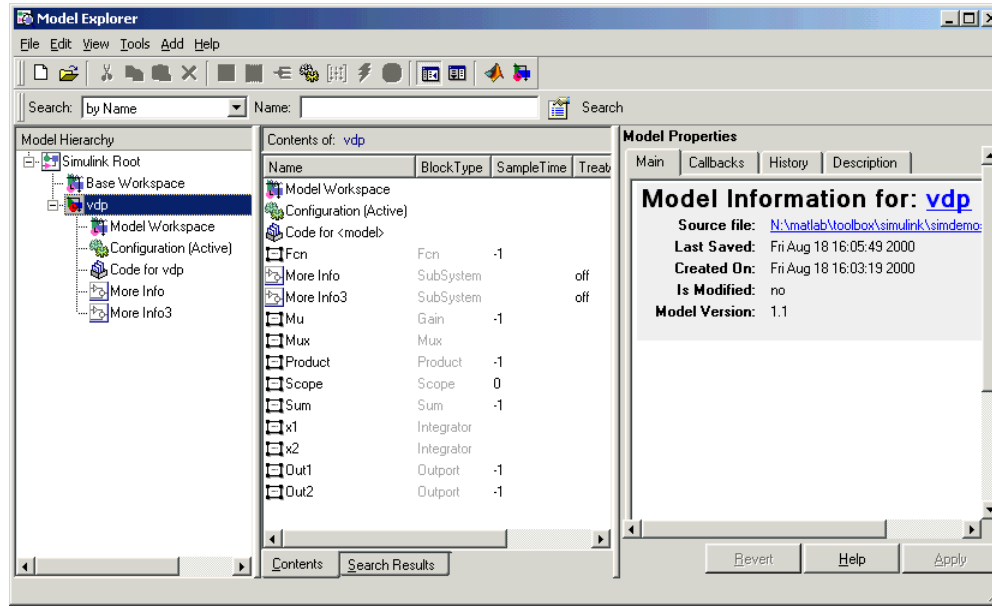
Tornado Support for VxWorks Target

V6.0 (R14) Real-Time Workshop supports Tornado Version 2.*x*, which targets VxWorks 5.*x*.

User Interface and Configuration Enhancements

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:



The Model Explorer features three resizable, scrolling panes:

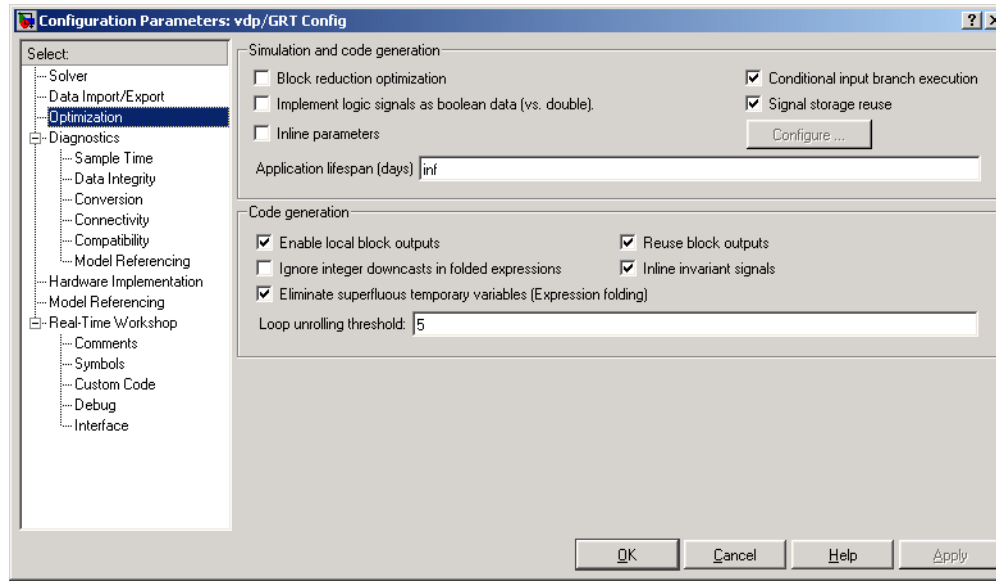
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane

For more information on the Model Explorer, see “The Model Explorer” in 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 in the next figure.



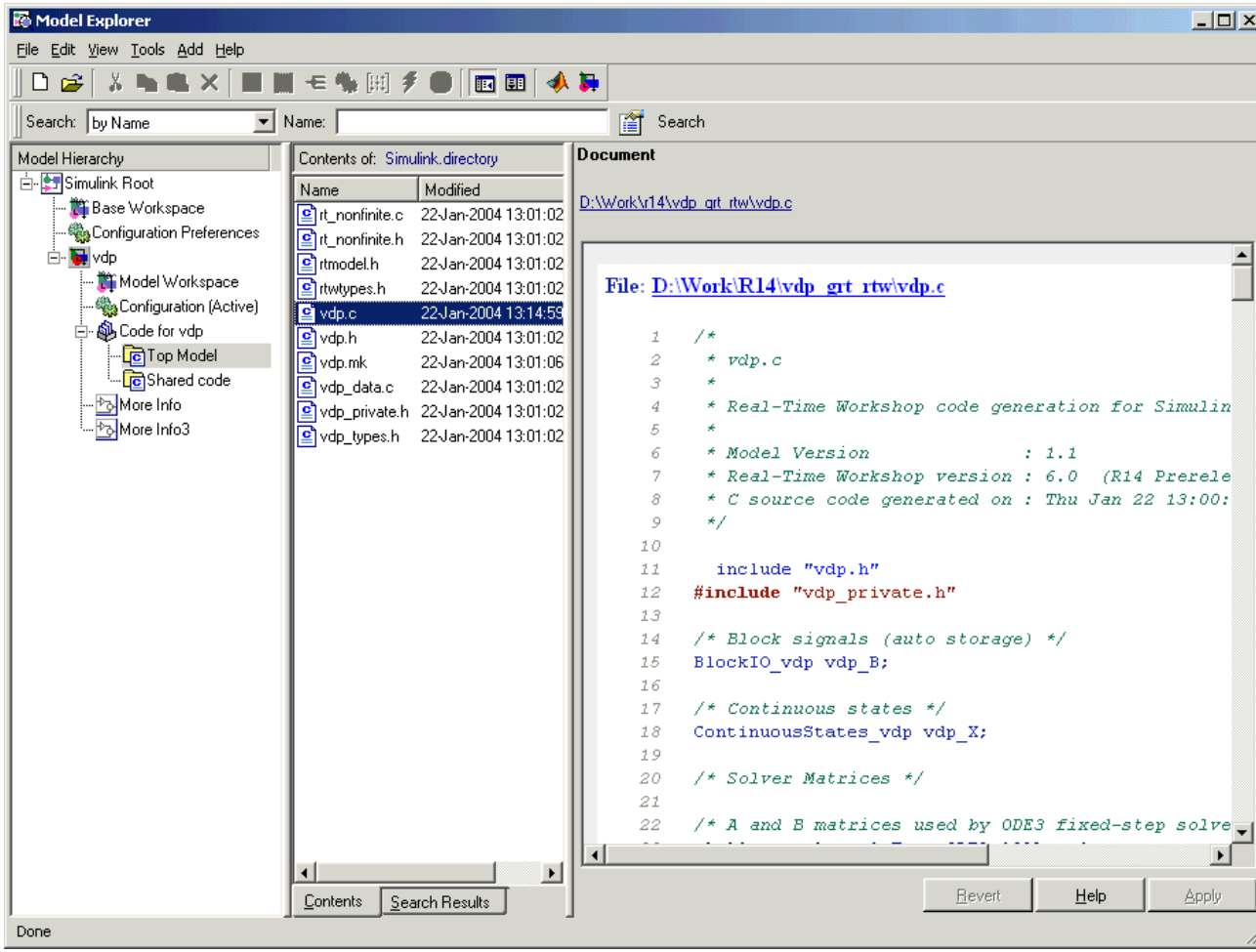
For details on configuration parameters for code generation, see “Choosing and Configuring Your Target”,“Adjusting Simulation Configuration Parameters for Code Generation”, and “Configuring Real-Time Workshop Code Generation Parameters” in the Real-Time Workshop Documentation.

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 **Model Hierarchy** pane of Model Explorer contains a node named `Code for` *`model`*. Under that node are other nodes, typically called `Top Model` and `Shared Code`.

When you click `Top Model`, the **Contents of** pane lists source code files in the build directory of each model that is currently open. The next figure 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 **Contents of** pane, click it once.

The views in the dialog pane are read-only. The code listings in that pane 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 **Model 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 Targets

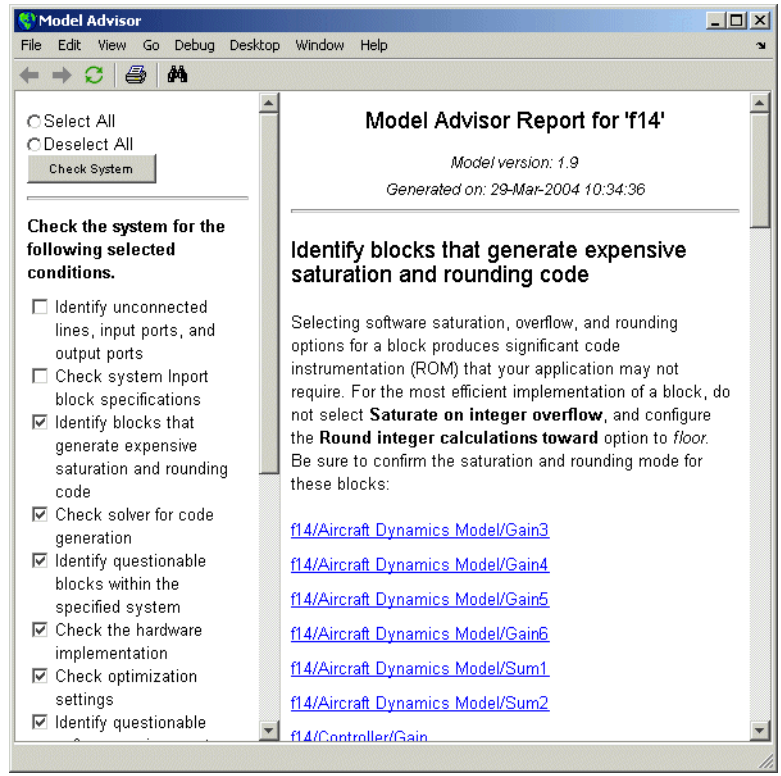
The Model Advisor (formerly called Model Assistant) is a tool that helps you configure any model to optimally achieve code generation objectives. Using it, you can quickly configure a model for code generation, and identify aspects of your model that impede production deployment or limit code efficiency. Clicking the icon labeled `Advice on` *`model`* in the **Model Hierarchy** pane launches the Model Advisor. This node is directly below the `Code for model` node, as the preceding 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 the following command in the MATLAB Command Window, specifying the name of model.

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

If the model (assumed to be on the MATLAB path) is not currently open, the Model Advisor opens it.

The following figure shows 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). The Intel compiler requires Microsoft Visual C/C++ Version 6.0 or higher to be installed.

Support for New Simulink Model Referencing (Model Block) Feature

The new Model block in 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 you open a top model, any models to which it refers are not loaded into memory until they are needed or you open them.

Note To take advantage of incremental model loading, models called from Model blocks must be saved at least once with Simulink V6.0 (R14). 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 (for example, GRT) you have specified (sometimes generically referred to as Real-Time Workshop targets). Real-Time Workshop also stores the generated code 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** on the **Model Referencing** pane of the Configuration Parameters dialog.

You can learn more about how Model blocks work and generate code by running the following 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.

Compatibility Considerations for Custom Targets

If you want to adapt a custom target for code generation compatibility with the model reference features, you need to modify the target's 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 "Shared Utilities Directory and the Build Process" on page 95.

System Target File Modifications. Your STF must implement a `SelectCallback` function (see "New SelectCallback Function for System Target Files" on page 95). 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');
slConfigUISetEnabled(hDlg, hSrc, 'ModelReferenceCompliant', false);
```

Template Makefile Modifications. In addition to the TMF modifications described in "Shared Utilities Directory and the Build Process" on page 95, you must modify your TMF variables and rules. See "Template Makefile Modifications" in the Real-Time Workshop documentation for instructions.

Signal, Parameter Handling, and Interfacing Enhancements

New C-API for Accessing Model Block Outputs and Parameter 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 the C-API, you can build target applications that log signals, monitor signals and tune parameters while the generated code executes.

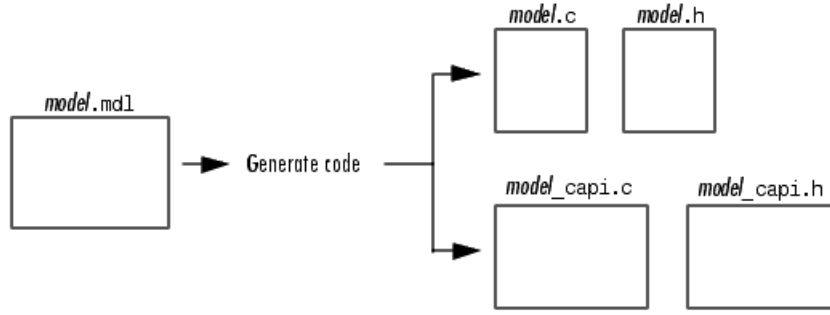
In previous releases, to access model parameters via the C-API, a model-specific parameter mapping file, `model_pt.c` was generated. Similarly, to access the `BlockSignals`, `model_bio.c` is generated. The new C-API improves the efficiency and capability of the interface while reducing its code size.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

- Referenced models
- Fixed-point data
- 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 provides 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 in the Configuration Parameters dialog box. 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 next figure illustrates generated files.



For details on how to use the C-API, see "Data Exchange APIs" in the Real-Time Workshop documentation.

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

C-API 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.

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

If a signal exiting an output port of a subsystem has a storage class other than `auto`, Real-Time Workshop internally propagates the label on that signal backwards into the subsystem so that the code generated for the subsystem uses that signal label, which is defined outside the subsystem.

Compatibility Considerations. 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.

Compatibility Considerations. In V6.0 (R14), whenever a signal state or parameter is wide, you must define the variable as a pointer to an array. In previous versions, an array of pointers was assumed. Here are the changes:

Width	Previous Versions	V6.0 (R14)
scalar	`double *x1`	`double *x1`
wide	`double *x2[]`	`double *x2`

The legacy code could define and initialize data as follows:

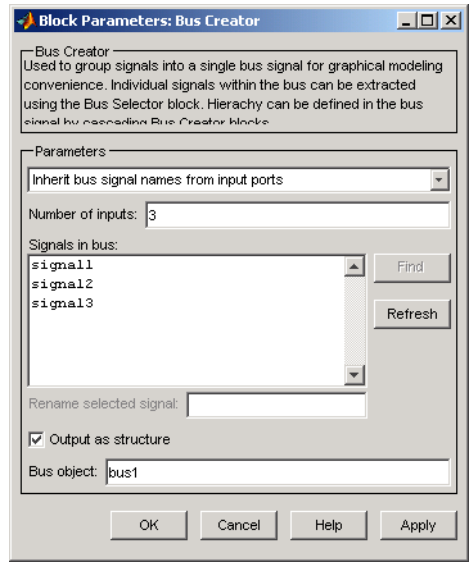
```
double x1_data;
double *x1 = &x1_data;
```

```
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 past releases, you could not assign a storage class to the output of a Bus Creator block. If you select the block's new parameter **Output as structure**, the output of the block can be assigned a storage class. This enables bus signals to occupy contiguous memory. When you select this parameter, you must specify a Simulink Bus object. You can make and modify bus objects (class `Simulink.Bus`) using the Bus Editor. Type `buseditor` in the MATLAB Command Window. An example Bus Creator dialog for a block that outputs a three-element structure is shown in the next figure.



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

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 box.
- A utility function, `disableautosignalresolution`, is available for converting existing models (that depended on implicit signal label resolution) to the new, more efficient approach.

CustomStorageClass and StorageClass Properties Initialized Differently

V6.0 (R14), Real-Time Workshop merges functionality of custom storage classes into the standard `Simulink.Parameter` and `Simulink.Signal` classes.

Compatibility Consideration. 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 V5.0 (R13).

In V5.0 (R13), the properties were initialized as

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

Starting in V6.0 (R14), the properties are initialized as

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

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 is not affected).

Compatibility Considerations. If you have customized any of these static files or their makefiles, 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 use Floating Scope blocks in external mode. A new section in the **External Mode** pane, **Floating scope**, contains the following new options:

- **Enable data uploading**

  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**

  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. Serial transport is implemented only for Windows 32-bit architectures.

For details on how to use the serial transport mechanism for external mode, see "Using the Serial Implementation".

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).

Compatibility Considerations. Most users will not notice this change. If, however, you have created your own custom transport layer for external mode, you must 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/`):

- Rename the function `ExtWaitForStartMsgFromHost()` to `ExtWaitForStartPktFromHost()`.
- Replace the functions `ExtSetHostData()` and `ExtSetHostMsg()` with `ExtSetHostPkt()`.
- Rename the function `ExtGetHostMsg()` to `ExtGetHostPkt()`.

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

- Replace the functions `ExtTargetDataPending()` and `ExtTargetMsgPending()` with `ExtTargetPktPending()`.
- Replace the functions `ExtGetTargetData()` and `ExtGetTargetMsg()` with `ExtGetTargetPkt()`.
- Rename the function `ExtSetTargetMsg()` to `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

You can now generate code for external mode such that it uses only static memory allocation ("malloc-free" code). The **Static memory allocation** check box 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 issues an out-of-memory error when it tries to allocate more memory than you are

allowed. In such cases, increase the value of **Static memory buffer size** and regenerate the code. 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 displays 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, you can use this console log to adjust the value specified for **Static memory buffer size**.

For more information on this new option, see "External Mode Interface Options" in the Real-Time Workshop documentation.

Code Customization Enhancements

Source Code for User S-Functions 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 as the S-function MEX-file. This directory must be on the MATLAB path.

Similarly, Real-Time Workshop adds 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 eliminates 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. You can use the blocks in subsystems as in top-level models (with minor exceptions). Custom Code blocks enable you to add your own code fragments to specific functions in the source code and header files generated by Real-Time Workshop. You can include the user code 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. 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 example**.)

Preventing User Source Code from Being Deleted from Build Directories

In V5.0 (R13), the behavior of Real-Time Workshop regarding handling of user source files in the build directory changed. Previously, any `.c` or `.h` files that you placed in the build directory were not deleted when you rebuilt targets. Now all foreign source files are deleted by default, but you can preserve them 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.

Compatibility Considerations. 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 C runtime library support that is 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

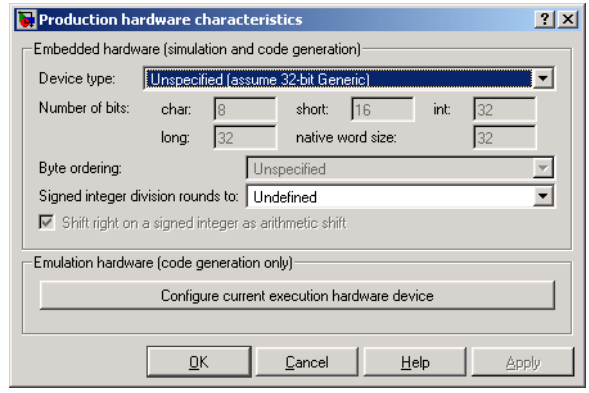
You choose among them by setting the **Target floating point math environment** option on the **Real-Time Workshop/Interface** pane of the Configuration Parameters dialog box. This enables you to specify different runtime libraries for different configuration sets within a given model.

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 you select `ISO-C`, the generated code calls 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 No Longer Supported

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 no longer supported.

Compatibility Considerations. When you open a preexisting model that has not been saved using the current version of Simulink, and select the **Hardware Implementation** pane of 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 V5.0 (R13) Real-Time Workshop is built for the first time in V6.0 (R14) without

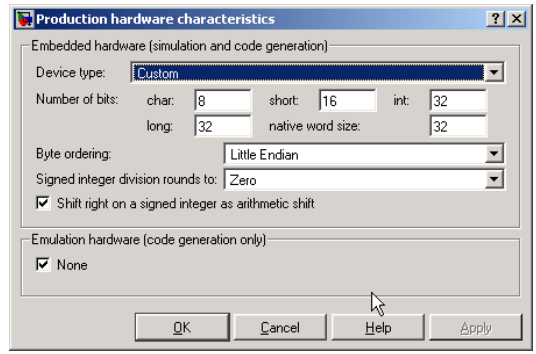
you having first specified characteristics of the **Current code generation execution hardware device** on the **Hardware Implementation** pane. If you build a model in this under-specified state, Real-Time Workshop scans the current directory, then the MATLAB path, for an existing hook file with the name *target*_rtw_info_hook.m. If the file is found, its instructions override the defaults in that section. You can subsequently specify any characteristic freely. If at any point prior to building the target code you specify **Current code generation execution hardware device**, Real-Time Workshop ignores hook files , as hardware characteristics are now configured.

When you open a preexisting (before V6.0) model, the **Hardware Implementation**pane displays a **Configure current execution hardware device** button. This button disappears after you press it once. When code is generated (**Ctrl+B**) for the target the model specifies,

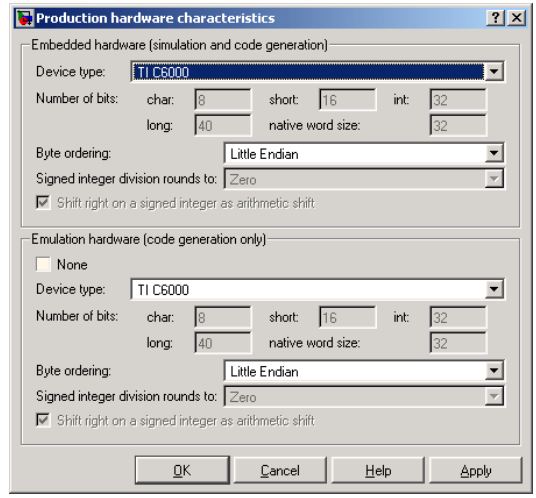
- If the target has a hook file, and the **Configure current execution hardware device** button has not yet been pressed,
  - The hook file is executed and configures the fields specifying current code generation execution hardware device.
  - A warning is issued to the user that the hook file was used.
  - 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 default.

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 preceding 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 in the next figure. In this figure, 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 box 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 ensures that timers will not overflow within the lifespan you specify. If you set **Application lifespan (days)** 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, `-L` *n*. 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.

For more information and an example, see “Setting a Clock Time Limit for a Rapid Simulation” in the Real-Time Workshop documentation.

New Asynchronous Block Library

A 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.

For descriptions of the VxWorks library blocks and information on gaining access to the library, see“Asynchronous Support” in the Real-Time Workshop documentation.

Compatibility Considerations. The older Interrupt Templates library (`vxlib`) is obsolete. It is provided only to allow models created prior to Real-Time Workshop V6.0 (R14) to continue to operate. If you have models that use `vxlib` blocks, The MathWorks recommends that you change them to use `vxlib1` blocks.

Automatic Slow-to-Fast and Fast-to-Slow Transition Detection for Rate Transition Block

The Rate Transition block has been updated to automatically detect whether transitions must be slow-to-fast or fast-to-slow, and act appropriately. Accordingly, the Block Parameters dialog box for the block has been modified to include only the following four options:

- **Ensure data integrity during transfer**
- **Ensure deterministic data transfer**
- **Outport sample time**
- **Initial condition**

For more information, see “Sample Rate Transitions” in the Real-Time Workshop documentation.

Compatibility Consideration. Simulink automatically updates all Rate Transition blocks in a model with this enhancement when you save the model in V6 (R14).

Automatic Insertion of Rate Transition Blocks

When you set up a model to use a fixed-step solver for multitasking, Simulink now automatically inserts Rate Transition blocks between periodic tasks that run at different rates and transfer data. This feature does not apply to transitions to or from non-periodic (asynchronous) tasks. You can control whether Simulink inserts Rate Transition blocks automatically with the **Automatically handle data transfers between tasks** check box on the **Solver** pane of the Configuration Parameters dialog box.

Simulink configures the blocks that it inserts automatically to ensure both data integrity and deterministic data transfer. As mentioned above, this feature applies to multitasking models only. Rate Transition blocks that Simulink inserts automatically do not appear on the model's block diagram. Nevertheless, they are implemented as semaphores or double buffers, depending on the constraints being observed, and affect simulation and code generation.

For more details, see "Automatic Rate Transition" in the Real-Time Workshop documentation.

Enhanced Absolute and Elapsed Time Computation

Certain blocks require the value of either *absolute* time (that is, the time from the start of program execution to the present time) or *elapsed* time (for example, the time elapsed between two trigger events). 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.
- S-function and TLC APIs 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

An important goal for both Real-Time Workshop and Real-Time Workshop Embedded Coder in V6.0 (R14) has been *target unification*. Target unification includes enhancements to the underlying technology and features 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 81).
- 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.

The following topics provide a high-level overview and comparison of feature enhancements and compatibility issues that result from target unification in Real-Time Workshop V6.0 (R14) and Real-Time Workshop Embedded Coder V4.0 (R14).

- "Code Format Unification" on page 81
- "Compatibility Considerations for GRT-Based Targets" on page 82
- "Real-Time Workshop and Real-Time Workshop Embedded Coder Feature Set Comparison" on page 85
- "Symbol Formatting Options Replaced" on page 88

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 (for example, `RealTime` or `RealTimeMalloc`).

In V6.0 (R14, 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`). 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 "Data Structures, Code Modules, and Program Execution" of the Real-Time Workshop Embedded Coder documentation. The calling interface generated by the GRT target is described in "Program Architecture" 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 Considerations for GRT-Based Targets" on page 82 for a description of target conversion issues.

Compatibility Considerations 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 83. 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 85. This lets your target support all ERT features 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 reimplement your custom target as a completely ERT-based target, including use of an ERT generated main program. This approach lets your target support all ERT features, 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 target. See "Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation for 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 V5.0 (R13). The conversion requires changes to your system target file, template makefile, and main program module.

To use `rtModel` instead of `SimStruct`, make the following changes to the system target file and template makefile:

- 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.

Make the following changes to your main program module (that is, your customized version of `grt_main.c`):

- 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 V6.0 (R14) 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 you select this option, 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 should take to achieve ERT compatibility depends on the features 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.

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.
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.

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
Module Packaging Features (MPF)	Not supported.	Extensive code customization features. See the Real-time Workshop Embedded Coder documentation.
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.
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.

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
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 Real-Time Workshop model configuration options, all related to formatting generated symbols, have been removed from the Configuration Parameters dialog box and replaced with 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, and so

on), followed by a unique *name mangling* string that is generated (if required) to resolve potential conflicts with other generated symbols.

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, Real-Time Workshop generates a name mangling string. The string has the minimum number of characters required to avoid the collision. Real-Time Workshop then inserts the other symbol components. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other components, they are truncated.

Compatibility Considerations. To avoid truncation that can result from the new default symbol formatting specification, 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.

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.

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.

Underscores No Longer Replace Spaces in Identifiers for Multi-Word Block Names

Prior to V6.0 (R14), Real-Time Workshop replaced each space in a multi-word block name with an underscore (_). For example, Actuator Model would be `Actuator_Model`. Starting in V6.0, the spaces in such names are removed rather than replaced. For example, the identifier for Actuator Model is generated as `ActuatorModel`.

Global Data Structure Identifiers for Targets Now Incorporate Model Name

Global data structures, such as `rtB`, `rtP` and `rtY` have new identifiers in ERT and GRT generated code. For GRT, these names now include the model name followed by `_B`, `_P`, `_Y`, and so on. (ERT targets provide you with flexible naming options as explained in "Symbol Formatting Options Replaced" on page 88). The construction of identifiers was changed to prevent name clashes when code for models containing Model blocks is generated and linked.

Compatibility Considerations

If you are interfacing external code to any Simulink global data, you might need to use the GRT compatible calling interface for ERT-based targets (see "Generating GRT Wrapper Code from the ERT Target" on page 85 for more information). The GRT interface enables you to access global data using the old-style identifiers 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.

Support for Simulink Configuration Set Feature

Support for New Simulink getActiveConfigSet Function

A new function, `getActiveConfigSet`, provides safe access to option settings stored in the active configuration set. The function returns an object through which you can access properties of the model's active configuration set. The

following example shows how to call `getActiveConfigSet` to turn the ERT option **Single output/update function** off.

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

Compatibility Considerations. 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.

New switchTarget Function

In V6.0 (R14) Simulink models store model-wide parameters and target-specific data in *configuration sets*. 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).

Real-Time Workshop has added a new function, `switchTarget`, to support configuration sets and enable you to automate target selection from scripts.

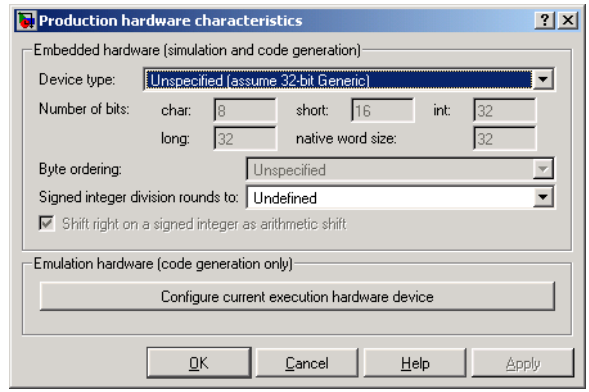
Arguments that you pass to the function include a handle to the model's active configuration set and a string that specifies a system target file.

For more information, see "Selecting a System Target File Programmatically" in the Real-Time Workshop Documentation.

Hardware Configuration Parameters

Compatibility Considerations

When you open a preexisting model that has not been saved using V6.0 (R14) 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 cleared, Real Time Workshop generates fully portable but less efficient code to implement right arithmetic shifts.

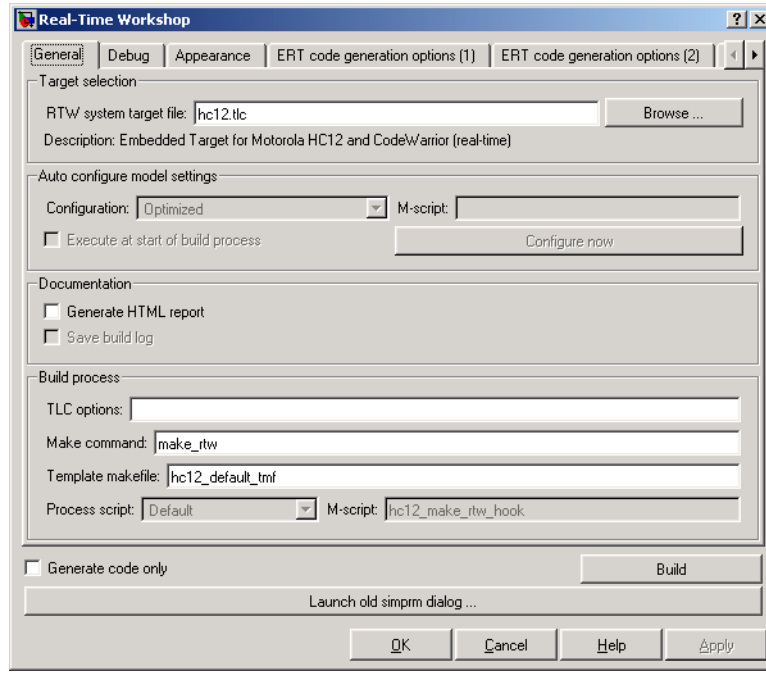
Enhancements and Changes that Affect Custom Targets

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, you should take advantage of the Model Explorer to present target options to end users.

Compatibility Considerations. To take advantage of the Model Explorer for presenting target options, you must modify your custom system target file. If you do not want to make the changes, a mechanism for using the old-style **Simulation Parameters** dialog is available for backwards compatibility.

The following figure shows an example of what users would see if you do not upgrade and the Embedded Target for Motorola® HC12 target is selected.



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 Release Notes* for details.

New SelectCallback Function for System Target Files

The V6.0 (R14) API for system target file callbacks provides a new `SelectCallback` function for use in system target files. This function is associated with the target rather than with any of its individual options. If you implement a `SelectCallback` function for the target, it is triggered once, when the user selects the target via the System Target File Browser.

For details on using the selectCallback function, see "SelectCallback Function for System Target Files" in the Real-Time Workshop documentation.

Compatibility Considerations. 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 "Compatibility Considerations for Custom Targets" on page 60 in the Real-Time Workshop documentation for an example.

Shared Utilities Directory and 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.

Compatibility Considerations

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 "Compatibility Considerations for Custom Targets" on page 60 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
```

2 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)
```

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

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

4 Create a SHARED_OBJS variable based on SHARED_SRC.

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

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

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

6 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) $<
```

7 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  $@ "
```

8 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)"
```

9 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.

Tornado Target Requires Macro in Template Make File

Tornado 2.2.1 installs 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
```

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

Compatibility Considerations

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` as follows:

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.

Custom Storage Classes Can No Longer Be Used with GRT Targets

In prior releases, it was possible to use custom storage classes with the GRT target if a Real-Time Workshop Embedded Coder license was available. In V6.0 (R14), you can no longer use custom storage classes when you generate code for GRT-based targets.

For information on how GRT and ERT targets now compare, see “Global Data Structure Identifiers for Targets Now Incorporate Model Name” on page 90. See “Code Generation Options and Optimizations” in the Real-Time Workshop Embedded Coder documentation for detailed information on the **GRT compatible call interface** option.

Compatibility Considerations

If you have licensed Real-Time Workshop Embedded Coder and want to build a model that uses custom storage classes with the GRT target, you should instead use ERT Target, and enable the **GRT compatible call interface** option. This option appears on the**Real-Time Workshop>Interface**pane

of the Configuration Parameters dialog box. When you use this option, Real-Time Workshop Embedded Coder generates GRT-compatible code that can include custom storage classes.

Target Language Compiler Enhancements and Changes

ISSLPRMREF TLC Built-In Supports 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
```

New Argument for TLC GENERATE_FORMATTED_VALUE Built-In Function

The `GENERATE_FORMATTED_VALUE` built-in function 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.

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
```

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

- `LibNumDiscreteSampleTimes()` to access `NumSynchronousSampleTimes`
- `LibNumAsynchronousSampleTimes()` to access `NumAsynchronousSampleTimes`

TLCFILES Built-In Now Returns Full Path to Model File Rather Than Relative Path

A change in TLC invocation now specifies a full path to model files rather than a relative path.

Compatibility Considerations. This change creates backwards incompatibility in some custom targets.

When migrating V5.0 (R13) custom targets to V6.0 (R14) , check for and adjust usage of the TLC function `TLCFILES` to determine context, such as the path to the model file, as necessary.

Documentation Enhancements

- *Getting Started with Real-Time Workshop* has been fully updated and includes a new tutorial on generating code for referenced models.
- *Real-Time Workshop User's Guide* is updated, and includes most of the information on new features described in this chapter.
- *Real-Time Workshop Target Language Compiler* has been updated. This document no longer includes an appendix describing all the records that might be encountered in a `model.rtw` file.

Version 5.2 (R13SP2) Real-Time Workshop

This table summarizes what's new in V5.2 (R13SP2):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
No	No	Fixed bugs	No

Version 5.1.1 (R13SP1+) Real-Time Workshop

This table summarizes what's new in V5.1.1 (R13SP1+):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Fixed bugs	No

New features and changes introduced in this version are

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 can be constructed only 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 — `x.field.field == x`.

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

Error Resulting from Inaccessible Signal Reporting No Longer Reported

Compatibility Considerations

In previous releases, Simulink and the Real-Time Workshop reported an error whenever a 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.

Version 5.1 (R13SP1) Real-Time Workshop

This table summarizes what's new in V5.1 (R13SP1):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
No	No	Fixed bugs	No

Version 5.0.1 (R13+) Real-Time Workshop

This table summarizes what's new in V5.0.1 (R13+):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	Fixed bugs	No

New features and changes introduced in this version are

- "Expanded Hook File Options" on page 107
- "Hook Files for Customizing Make Commands" on page 109

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 (that is, 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 V5.0 (R13):

- `TypeEmulationWarnSuppressLevel`

  Suppresses 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`:

  Specifies limitations of the target C preprocessor to do math with signed integers. Use this option 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`

  Specifies 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, you can get more details by typing `rtwtargetsettings` in the MATLAB Command Window.

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` (for example, `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 overrides its default (for example, Lcc) batch file wrapping code. For an example make command hook file, see *`matlabroot`*`/toolbox/rtw/rtw/wrap_make_cmd.m`. Such hook files are distinct from the target-specific hook files used to describe hardware characteristics (see “Expanded Hook File Options” on page 107).

Version 5.0 (R13) Real-Time Workshop

This table summarizes what's new in V5.0 (R13):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Fixed bugs	No

New features and changes introduced in this version are organized by these topics:

Compiler Support Enhancements

Expanded Support for Borland C Compilers

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

In addition, V5.0 (R13) reinstates support for Borland Version 5.2 "out-of-the-box" for all targets, except when importing S-functions that Real-Time Workshop generates. In such instances, 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"
```

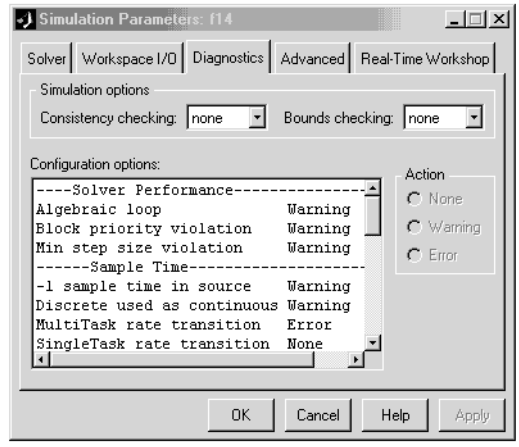
Lcc Now Links Libraries in Directory sys/lcc/lib

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

Model Configuration Features and Enhancements

Diagnostics Pane Items Classified into Logical Groups

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



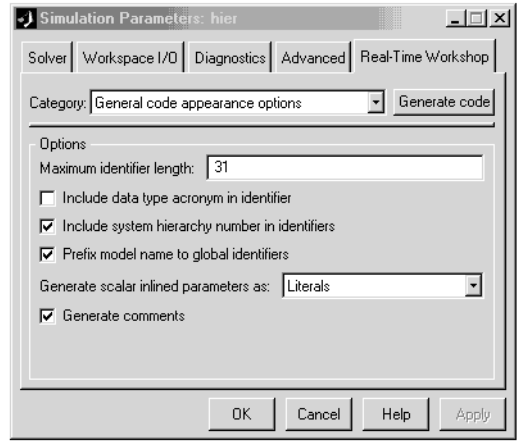
Comments Not Generated for Reduced Blocks When "Show eliminated statements" Is Off

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.

Compatibility Considerations. If you want Real-Time Workshop to generate comments for blocks that are removed due for block reduction optimization, select the **Show eliminated statements** option.

New General Code Appearance Options

A new section has been added to the **Real-Time Workshop** pane of the Simulation Parameters dialog box, named `General code appearance options`. The new section groups four new code formatting options to two existing options. The `General code appearance` appear as shown in the next figure.



The four new options are

Option	Description
Maximum identifier length	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.

Option	Description
Include data type acronym in identifier	Prepends 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.
Include system hierarchy number in identifiers	Adds 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.
Prefix model name to global identifiers	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. This option is on by default.

Option	Description
Generate scalar inline parameters as:	Controls the code style for inlined parameters. You can set this option to `literals` or `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 to guarantee that it never ends up in the `rtP` data structure. Use `ssSetConstRunTimeParamInfo` in the S-function to register a constant runtime parameter.
Generate comments	An existing global option moved from the `General code generation options (cont)` category to this one. As in the prior release, by default **Generate comments** is on.

Identifier Construction for Generated Code Has Been Simplified

The methods 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.
- You can specify a 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.

GUI Control over Behavior of Assertion Blocks in Generated Code

The **Advanced** pane of the Simulation Parameters dialog box provides a new **Model Verification block control** popup menu you can use 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 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 includes one of the following statements at appropriate locations, depending on the block's input signal type (Boolean, real, or integer, respectively).

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

By default `utAssert` is a non-option 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 calls back to itself to execute assertion blocks instead of using generated code. Thus a user-defined callback is still called when assertions fail.

GUI Control Over TLC %assert Directive Evaluation

Prior versions required you to specify the `-da` Target Language Compiler command switch for TLC `%assert` directives to be evaluated. Now you 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 option is off by default.

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

To see the current setting, use the command

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

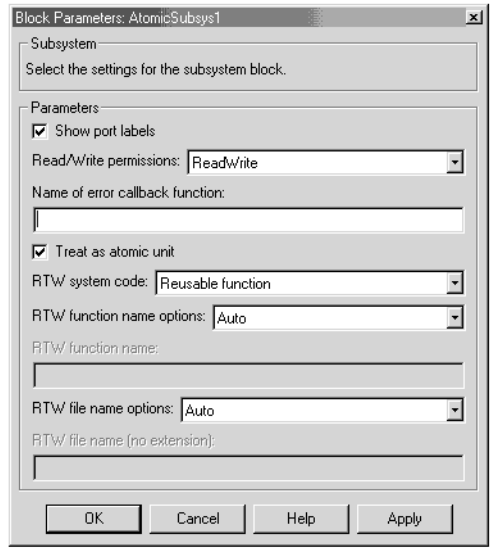
Code Generation Infrastructure Enhancements

Code for Nonvirtual Subsystems Is Now Reusable

Real-Time Workshop V5.0 (R13) introduces the ability to reuse code generated for nonvirtual subsystems. In prior releases, Real-Time Workshop generated a separate block of code for each nonvirtual subsystem. In some circumstances — for example, when you use a library block multiple times in the same fashion — it is now 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, 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 V5.0 (R13) 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 in the next figure.



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 you specify `Reusable function` and reuse is not possible, the result is a function without arguments. When you specify `Auto` and reuse is not possible, Real-Time Workshop inlines the subsystem's code (or in special cases, creates 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 130) 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 Generating 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.

Compatibility Considerations. Real-Time Workshop V5.0 (R13) alters aspects of generated code to support code reuse for nonvirtual subsystems. As explained above, you have the ability to select or override this feature, as well as to specify function and file names from the graphical user interface.

Packaging of Generated Code Files Simplified

The packaging of generated code into `.c` and `.h` files has been simplified. 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 "Code Modules" in the Real-Time Workshop Embedded Coder User's Guide for more information.

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.
*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.

File	Description
`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.

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

Old Filename	New Filenames
`model``_common.h`	*`model`*`_types.h` and *`model`*`_private.h`
`model``_export.h`	*`model`*`.h`
`model``_prm.h`	*`model`*`_data.c`
`model``_reg.h`	*`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.

Compatibility Considerations. In prior releases, the information about the root models was stored in the data structure `SimStruct`. 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` structure is lightweight and eliminates the 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), and so on. 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.

If you have previously customized GRT, GRT-Malloc, or Tornado targets, upgrade each customized target to use the `rtModel` instead of `SimStruct`.

To upgrade a target 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'`.
- Change the names of the following functions such that they use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass into them.

  ```
  rt_GetNextSampleHit
  rt_UpdateDiscreteTaskSampleHits
  ```

```
rt_UpdateContinuousStates
rt_UpdateDiscreteEvents
rt_UpdateDiscreteTaskTime
rt_InitTimingEngine
```

See `grt_main.c` (or `grt_malloc_main.c`) for the list of arguments that need to be passed into each function.

- Modify 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, change the target TLC file and the template make files.

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

```
%assign GenRTModel = TLC_TRUE
```

Hook Files Required for Communicating Target-specific Word Characteristics

You must now supply a target hook file (M-file) to specify target hardware characteristics, such as word sizes and overflow behavior.

Compatibility Considerations. To communicate details about target hardware characteristics, you must now 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, you must name the file grt_rtw_info_hook.m, and the file needs to be on the MATLAB path. If the hook file is not provided, Real-Time Workshop uses default values based on the host's characteristics, 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

Real-Time Workshop now generates code for models that include Stateflow charts in a single set of output files.

Compatibility Considerations. In earlier releases, Real-Time Workshop wrote code generated from Stateflow charts to source code files distinct from the source code files (such as *model*.c, *model*.h, etc.) generated from the rest of a model.

Now, by default, Stateflow no longer generates any separate files from 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 the names of the functions and the code files generated.

Conditional Input Branch Execution Optimization

This release introduces an optimization called conditional input branch execution, which speeds simulation and execution of code generated from the model.

Compatibility Considerations. 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, see the demo `rtwdemo_condinput` .

Block 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.

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 detailed description of the new block, see Rate Transition in the Simulink reference 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 code that targets one of the many new specific, optimized lookup table operations in the Real-Time Workshop runtime 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 require less space.

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.

Rapid Simulation Target Enhancement

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.

For details, see “Licensing Protocols for Simulink Solvers in RSim Executables”.

Compatibility Considerations

You can maintain backwards compatibility (that is, 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 Enhancements

- Support for Rapid Simulation (RSim) target

  The RSim 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.

- Support for ERT target

  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.

- Support for 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.

Simulink Data Object Enhancements

Simulink data objects include several new string properties that you can exploit for customizing code generation. These properties are

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

In this release, the Simulink engine and Target Language Compiler do not use these properties. 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.

model.rtw Changes

In this release, a number of changes have been made to *model*.rtw.

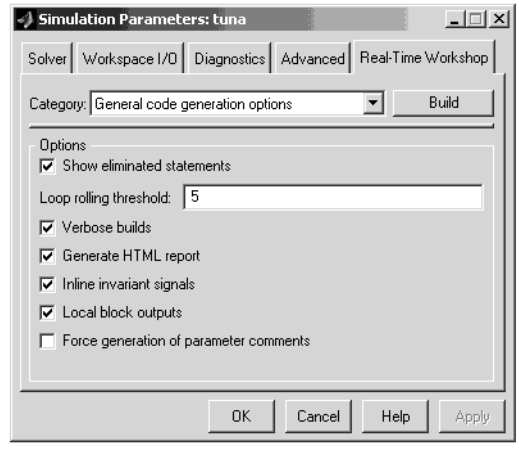
Compatibility Considerations

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.

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 Rapid Simulation target).

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



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

Efficiency of Code Generated for GRT and GRT-Malloc Targets Improved

Substantial changes have been made to the GRT and GRT-Malloc targets to improve the efficiency of generated code.

Compatibility Considerations

If you have customized either type of target, you should make changes to your modified files to ensure that your target works properly with V5.0 (R13) Real-Time Workshop.

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 perform all steps outlined in "Most Targets Use rtModel Instead of Root SimStruct" on page 122 and "Logging Code Moved to the Real-Time Workshop Library" on page 131.

Logging Code Moved to the Real-Time Workshop Library

All the support functions used for logging data have been moved from `rtwlog.c` to the Real-Time Workshop library.

Compatibility Considerations

If you have customized a GRT or GRT-Malloc Target, 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.

Custom Code Blocks Moved from Simulink Library

The Custom Code blocks have been moved to a new library, named `custcode.mdl` (type `custcode` to access them).

Compatibility Considerations

Because custom code blocks are linked to this new library, backward compatibility is assured.

Target Language Compiler Changes

- SPRINTF built-in function added

  A C-like `sprintf` formatting function has been added to the Target Language Compiler, 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. The function operates like C library `sprintf()`, except that output is the return value rather than contained in an argument to `sprintf`.

- `BlockInstanceData` function no longer available

- `%filescope` directive added

  A new directive, `%filescope`, is now available for limiting scopes of variables to the files in which they are defined. All variables defined after the appearance of `%filescope` in a file have this property; otherwise, they default to global variables.

- Global variables `::` operator available

  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 remain global variables. Nested include files cannot access variables local to the file that included them.

Compatibility Considerations

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

Documentation Enhancements

- The expression folding API is documented and available for you to use, particularly for writing 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 for details.
- The *Real-Time Workshop User's Guide* has been significantly updated and reorganized.
- Information pertaining to data structures and subsystems has been updated and made more accessible.
- New features and GUI changes have been documented
- A new *Getting Started with Real-Time Workshop* is available. 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.
- The Target Language Compiler documentation has been significantly updated and reorganized. 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.

Fixed Bugs

- "ImportedExtern and ImportedExternPointer Storage Class Data No Longer Initialized" on page 135
- "External Mode Properly Handles Systems with no Uploadable Blocks" on page 135

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 initialization is the external code's responsibility.

External Mode Properly Handles Systems with no Uploadable Blocks

Connecting to systems with no blocks that can be uploaded in external mode used to fail and cause Simulink to act as though a simulation was running when none was. The only way to exit the model was to exit 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 are 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 and 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 now generates 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.

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 (that is, 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 ("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.

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. V4.1 (R12.1) 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.

Limitations for HP and IBM Platforms

The V4.0 (R12) platform limitation for Real-Time Workshop for the HP and IBM platforms still applies to V5.0 (R13). On the HP and IBM platforms, the Real-Time Workshop opens the V3.0 (R11) **Tunable Parameters** dialog box in place of the **Model Parameter Configuration** dialog box. Although they differ in appearance, both dialogs present the same information and support the same functionality.

Version 4.1 (R12.1) Real-Time Workshop

This table summarizes what's new in V4.1 (R12.1):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	Fixed bugs	No

New features and changes introduced in this version are

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** pane of the Simulation Parameters dialog box is now turned on 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" in the Real-Time Workshop documentation for more information.

Compatibility Considerations

In previous releases, the **Block reduction** option on the **Advanced** pane of the Simulation Parameters dialog box was off by default. This option is now on by default.

Buffer Reuse Code Generation Option

A **Buffer reuse** option has been added to the **Real-Time Workshop** pane of the Simulation Parameters dialog box. When you select this option, Real-Time Workshop reuses signal storage whenever possible.

See "Reuse Block Outputs" in the Real-Time Workshop documentation for more information.

Compatibility Considerations

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

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

The ability to set and get this option with the `set_param` and `get_param` commands is still supported.

For a description of `bufferreuse`, see .

Build Directory Validation

The build process now disallows building programs in the MATLAB directory tree.

Compatibility Consideration

Prior to this release, Real-Time Workshop allowed you to build programs in the MATLAB directory tree. As of V4.1 (Release 12.1), this is no longer allowed. If you attempt to generate code in the MATLAB directory tree, Real-Time Workshop displays an error message, 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 V4.0 (R12), lets you generate code and build an executable from any nonvirtual subsystem within a model. In Real-Time Workshop V4.1 (R12.1), 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 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 that contain information required for parameter tuning.

See "C-API for Interfacing with Signals and Parameters" 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.

Support for Control Flow Blocks

Simulink V4.1 (R12.1) 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. Real-Time Workshop V4.1 (R12.1) introduces code generation support for these blocks.

For more information on the control flow blocks, see "Modeling Control Flow Logic" 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

- Support for inline parameters

  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 box 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 runtime tuning for selected parameters that are important to your application. In addition, the **Model Parameter Configuration** dialog box 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 (that is, `grt`, `grt_malloc`, and Tornado) now allow inline parameters.

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

- New 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

A new **Comments** option has been added to the **Real-Time Workshop** pane of the Simulation Parameters dialog box. 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 Option

A new **Include system hierarchy in identifiers** option has been added to the **Real-Time Workshop** pane of the Simulation Parameters dialog box. When you select this option , 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 Support for Inline Parameters

The Rapid Simulation (RSim) 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 V4.0 (R12), 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, 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 State Storage 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 expands to `/home/$USER/outdir/file.mat`. The Real-Time Workshop uses the expanded names.

Target Language Compiler Enhancements

This section summarizes Target Language Compiler enhancements.

- New TLC debugger added

  The TLC debugger helps you identify programming errors in your TLC code. You can 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** pane.
  - 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** pane.

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

- *`model`*`.rtw` file format changed

  The format of the *`model`*`.rtw` file has changed.

- Block I/O connection handling cleaned up

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

- Support for literal string added

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

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

  The following examples are equivalent.

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

- Library functions added

  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.

Compatibility Considerations

- *model*`.rtw` file format has changed. For more information, see the Target Language Compiler documentation.
- `BlockTypeSetup` and `BlockInstanceSetup` calls have been reordered. 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 V4.1 (R12) , 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.

RTWInfo Property Changed

Changes have been made to the RTWInfo property of `Simulink.Signal` and `Simulink.Parameter` data objects.

Compatibility Considerations

In V4.0 (R12), the `RTWInfo` class had a `TypeQualifier` property, corresponding to the **RTW storage type qualifier** field of signal ports and parameters.

Real-Time Workshop V4.1 (R12.1) 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. You can set and retrieve the property 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.

Fixed Bugs

Real-Time Workshop V4.1 (R12) includes the following bug fixes.

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. 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

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

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

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
```

Unnecessary Warnings and Declarations Removed 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.

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`.

Version 4.0 (R12) Real-Time Workshop

This table summarizes what's new in V4.0 (R12):

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes—Details labeled as **Compatibility Considerations**, below. See also Summary.	No bug fixes	No

New features and changes introduced in this version are

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.

See the Real-Time Workshop Embedded Coder documentation for details.

Compatibility Considerations

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.

Support for Simulink Data Objects

The Real-Time Workshop supports Simulink data objects. 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. You can extend the `Simulink.Parameter` and `Simulink.Signal` classes to include user-defined properties.

See "Simulink Data Objects and Code Generation" in the Real-Time Workshop documentation for details.

Support for ASAP2 Data Files

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 for details.

Enhanced Real-Time Workshop Configuration Pane

The **Real-Time Workshop** pane of the Simulation Parameters dialog box has been reorganized and made easier to use. See "Configuring Real-Time Workshop Code Generation Parameters" in the Real-Time Workshop documentation for details.

Other User Interface Enhancements

- The **Tools** menu of the Simulink model window contains a new **Real-Time Workshop** submenu with shortcuts to frequently used features.
- You can now select a target configuration from the System Target File Browser by double-clicking on the desired entry in the target list.

See the Real-Time Workshop documentation for details.

Compatibility Considerations

The double-click mechanism for selecting a target configuration from the System Target File Browser does not replace the previous selection method. You can still select a target entry and then click **OK**.

Support for New Simulink Advanced Options Pane

An **Advanced** pane options pane has been added to the Simulation Parameters dialog box. The **Advanced** pane contains

- New code generation options
- Options formerly located in the **Diagnostics** pane
- Options formerly located in the **Real-Time Workshop** pane

Compatibility Considerations

Simulation Parameters dialog box options formerly located in the Diagnostics and Real-Time Workshop panes have been relocated to the new **Advanced** pane.

Model Parameter Configuration Dialog Box

The **Model Parameter Configuration** dialog box extends and replaces the **Tunable Parameters** dialog box. The **Model Parameter Configuration** dialog box enables you to

- Declare individual parameters to be tunable
- Control the generated storage declarations for each parameter

See "Parameter Storage, Interfacing, and Tuning" in the Real-Time Workshop documentation for details.

Compatibility Considerations

You must now use the **Model Parameter Configuration** dialog box instead of the **Tunable Parameters** dialog box to declare tunable parameters.

Support for Tunable Expressions

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 box. When referenced in lower-level subsystems, such parameters remain tunable.

See "Tunable Expressions" in the Real-Time Workshop documentation for a detailed discussion on using tunable parameters in expressions.

S-Function Target Enhancements

S-function target enhancements include:

- Support for variable-step solvers
- Supports for tunable parameters
- New **Generate S-function** menu option on the Simulink Tools menu that 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

New features have been added to external mode:

- 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.
- The external mode communications application program interface (API) is now 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.
- The **External Signal & Triggering** dialog box has been enhanced as indicated in the compatibility considerations listed below.
- As indicated in the compatibility considerations listed below, several additional features now support external mode.

Compatibility Considerations

Previously, you could use only Scope blocks in external mode to receive and view signals uploaded from the target program. The following features now support external mode:

- The default operation of the **External Signal & Triggering** dialog box has been changed to make monitoring a target program simpler. See "External Signal Uploading and Triggering" in the Real-Time Workshop documentation for details.
- Previously, you could use only Scope blocks to receive and view signals uploaded from a target program. The following features 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.

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. Real-Time Workshop derives the build directory name, *`model_target`*`_rtw`, 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.

Compatibility Considerations

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)
```

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** pane of the Simulation Parameters dialog box.

- **Block reduction**: When 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 selected, Simulink optimizes memory usage when multiple block parameters refer to storage locations that are separately defined but structurally identical.
- **Signal storage reuse**: Replaces the (Enable/Disable) **Optimized block I/O storage** option. See the compatibility considerations below for details.

See "Optimizing a Model for Code Generation" in the Real-Time Workshop documentation for more information on code optimization.

Compatibility Considerations

The **Signal storage reuse** option replaces the (Enable/Disable) **Optimized block I/O storage** option of previous releases. **Signal storage reuse** is functionally identical to the older option. Turning **Signal storage reuse** on is equivalent to enabling **Optimized block I/O storage**.

Subsystem 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

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 box.

See “Nonvirtual Subsystem Code Generation” in the Real-Time Workshop documentation for details.

Compatibility Considerations

Nonvirtual subsystem code generation replaces the **Function management** code generation options available in previous releases. Nonvirtual subsystem code generation is a more general and flexible mechanism for controlling the number and size of generated files than the **Function management** code generation options, **File splitting** and **Function splitting**.

Standard Filename Extensions for Generated Files

Real-Time Workshop now generates source code and header files that have standard filename extensions —`.c` and `.h`.

Compatibility Considerations

In previous releases, Real-Time Workshop gave some generated files special filename extensions, such as `.prm` or `.reg`. As of this release, Real-Time Workshop generates source code and header files that have standard filename extensions. The file naming conventions for the following generated files have changed:

File	Old Filename	New Filename
Model registration file	*model*`.reg`	*model*`_reg.h`
Model parameter file	*model*`.prm`	*model*`_prm.h`
`BlockIOSignals` structure file	*model*`.bio`	*model*`_bio.c`
`ParameterTuning` file	*model*`.pt`	*model*`_pt.c`
External mode data type transition file	*model*`.dt`	*model*`_dt.c`

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.

hilite_system and Code Tracing

Real-Time Workshop uses a new command, `hilite_system`, to write 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.

For more information on identification tags and code tracing, see "Tracing Generated Code Back to Your Simulink Model".

Compatibility Considerations

In previous releases, Real-Time Workshop used the `locate_system` command to trace a tag back to the generating block. Real-Time Workshop now uses the new `hilite_system` command to trace identification tags instead of `locate_system`. Starting with this release, use the `hilite_system` command to trace a tag back to the generating block.

Generation of Parameter Comments

The **Force generation of parameter comments** option in the **General code generation options**section of the **Real-Time Workshop** pane of the Simulink Parameters dialog box 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.

Column-Major Matrix Ordering

Real-Time Workshop now uses column-major ordering for two-dimensional signal and parameter data instead of row-major ordering.

Compatibility Considerations

In previous releases, Real-Time Workshop used row-major ordering for two-dimensional signal and parameter data. Real-Time Workshop now uses column-major ordering.

If your hand-written code interfaces to such signals or parameters via `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` declarations, review any code that relies on row-major ordering, and make appropriate revisions.

S-Function Target MEX-Files Must Be Rebuilt

Compatibility Considerations

S-function MEX-files generated by the S-function target under V3.0 (R11) are not compatible with V4.0 (R12). The incompatibilities are due to new features, such as parameter pooling.

If you have built S-function MEX-files with the S-function target under V3.0 (R11), you must rebuild them. See “The S-Function Target” in the Real-Time Workshop documentation for more information.

Target Language Compiler Enhancements

The Target Language Compiler has been enhanced as follows:

- TLC file parsed before execution

  The Target Language Compiler now 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.

- Speed enhanced

  The Target Language Compiler features speed improvements throughout the software. In particular, the speed of block parameter generation has been enhanced.

- Build directory created and used

  The Target Language Compiler now 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.

- Profiler added

  A new profiler has been added to the Target Language Compiler to help you find performance problems in TLC code.

- *`model`*`.rtw` changes

  This release contains a new format and changes to the *`model`*`.rtw` file and the size of the file has been reduced.

- Block parameter aliases added

  Aliases have been added for block parameters in the *`model`*`.rtw` file.

- Text expansion improved

  TLC contains new, flexible methods for text expansion from within strings.

- Column-major ordering used

  Two-dimensional signal and parameter data now use column-major ordering.

- Record handling improved

  TLC now utilizes new record data handling.

- Language semantics changed

  - Improved `EXISTS` behavior.
  - 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.

- Built-in functions added

  The following built-in functions have been added to the language:

  ```
  FIELDNAMES
  GENERATE_FORMATTED_VALUE
  GETFIELD
  ISALIAS
  ISEMPTY
  ISEQUAL
  ISFIELD
  REMOVEFIELD
  SETFIELD
  ```

- Built-in values added

  The following built-in values have been added to the language:

  ```
  INTMAX
  INTMIN
  TLC_FALSE
  TLC_TRUE
  UINTMAX
  ```

- Support for inlined code added

  Support has been added for two-dimensional signals in inlined code.

Compatibility Considerations

If you are upgrading from Release 11 to Release 12, the following changes may affect your TLC code:

- Nested evaluations are no longer supported. Expressions such as the following are no longer supported:

  ```
  %<LibBlockParameterValue(%<myVariable>,"", "", "")>
  ```

  You must 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>")
```

Compatibility Summary for Real-Time Workshop

This table summarizes new features and changes that might cause incompatibilities when you upgrade from an earlier version, or when you use files on multiple versions. Details are provided in the description of the new feature or change.

Version (Release)	New Features and Changes with Version Compatibility Impact
Latest Version V6.6.1 (R2007a+)	None
V6.6 (R2007a)	None
V6.5 (R2006b)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Code Formatting Consistency Improvements" on page 15 • "Change to Default Settings for Multitasking Diagnostic Options" on page 16
V6.4.1 (R2006a+)	None
V6.4 (R2006a)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Format Enhancements for model.rtw File" on page 27 • "Changes to TLC Files in matlabroot/rtw/c/tlc" on page 31
V6.3 (R14SP3)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Customizations to Built-In Blocks" on page 36 • "Use slbuild Instead of rtwgen" on page 36 • "Model Hardware Configuration Parameters Now Honor Device Type Restrictions" on page 37 • "rem Function No Longer Supports Tunable Arguments" on page 38
V6.2.1 (R14SP2+)	None

Version (Release)	New Features and Changes with Version Compatibility Impact
V6.2 (R14SP2)	None
V6.1 (R14SP1)	None
V6.0 (R14)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Support for New Simulink Model Referencing (Model Block) Feature" on page 59 • "New C-API for Accessing Model Block Outputs and Parameter Data" on page 62 • "Back-Propagating Auto, Test-pointed Signal Labels Through Subsystem Output Ports" on page 64 • "Declaring Wide Signals, States, and Parameters as ImportedExternPointer" on page 64 • "External Mode Changes May Impact Customized Makefiles and Static Main files" on page 67 • "Upgrading Custom Transport Layers for External Mode to Single-Channel Architecture" on page 68 • "Preventing User Source Code from Being Deleted from Build Directories" on page 71 • "Hook Files Describing Hardware Characteristics No Longer Supported" on page 73 • "New Asynchronous Block Library" on page 77 • "Symbol Formatting Options Replaced" on page 88 V6.0 (R14) **Compatibility Considerations** continue in the next cell

Version (Release)	New Features and Changes with Version Compatibility Impact
V6.0 (R14) (Continued)	V6.0 (R14) **Compatibility Considerations** continued from previous • "Global Data Structure Identifiers for Targets Now Incorporate Model Name" on page 90 • "Hardware Configuration Parameters" on page 92 • "Defining and Displaying Custom Target Options" on page 93 • "New SelectCallback Function for System Target Files" on page 95 • "Shared Utilities Directory and the Build Process" on page 95 • "Tornado Target Requires Macro in Template Make File" on page 98 • "Custom Storage Classes Can No Longer Be Used with GRT Targets" on page 99 • "Accessing the Number of Sample Times from TLC for Custom Targets" on page 101 • "TLCFILES Built-In Now Returns Full Path to Model File Rather Than Relative Path" on page 101
V5.1.1 (R13SP1+)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Error Resulting from Inaccessible Signal Reporting No Longer Reported" on page 105
V5.1 (R13SP1)	None
V5.0.1 (R13+)	None

Version (Release)	New Features and Changes with Version Compatibility Impact
V5.0 (R13)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Comments Not Generated for Reduced Blocks When "Show eliminated statements" Is Off" on page 112 • "Code for Nonvirtual Subsystems Is Now Reusable" on page 118 • "Packaging of Generated Code Files Simplified" on page 120 • "Most Targets Use rtModel Instead of Root SimStruct" on page 122 • "Hook Files Required for Communicating Target-specific Word Characteristics" on page 124 • "Code Generation Unified for Real-Time Workshop and Stateflow" on page 125 • "Conditional Input Branch Execution Optimization" on page 125 • "model.rtw Changes" on page 130 • "Efficiency of Code Generated for GRT and GRT-Malloc Targets Improved" on page 131 • "Logging Code Moved to the Real-Time Workshop Library" on page 131 • "Custom Code Blocks Moved from Simulink Library" on page 132 • "Target Language Compiler Changes" on page 132

Version (Release)	New Features and Changes with Version Compatibility Impact
V4.1 (R12+)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Block Reduction Option On by Default" on page 141 • "Buffer Reuse Code Generation Option" on page 141 • "Build Directory Validation" on page 142 • "Target Language Compiler Enhancements" on page 146 • "RTWInfo Property Changed" on page 148
V4.0 (R12)	See the **Compatibility Considerations** subheading for each of these new features or changes: • "Real-Time Workshop Embedded Coder" on page 155 • "Other User Interface Enhancements" on page 156 • "Support for New Simulink Advanced Options Pane" on page 156 • "Model Parameter Configuration Dialog Box" on page 157 • "External Mode Enhancements" on page 158 • "Build Directory" on page 159 • "Code Optimization Features" on page 161 • "Nonvirtual Subsystem Code Generation" on page 161 • "Standard Filename Extensions for Generated Files" on page 162 • "hilite_system and Code Tracing" on page 163 • "Column-Major Matrix Ordering" on page 164 • "S-Function Target MEX-Files Must Be Rebuilt" on page 164 • "Target Language Compiler Enhancements" on page 164