Real-Time Workshop Release Notes

The "Real-Time Workshop 6.0 Release Notes" on page 1-1 describe the changes introduced in the latest version of Real-Time Workshop. The following topics are discussed in these Release Notes:

The Real-Time Workshop Release Notes also provide information about recent versions of the product, in case you are upgrading from a version that was released prior to Release 13 with Service Pack 1.

Printing the Release Notes

If you would like to print the Release Notes, you can link to a PDF version.

Contents

1 Real-Time Workshop 6.0 Release Notes

2 Real-Time Workshop 5.1 Release Notes

3 Real-Time Workshop 5.0.1 Release Notes

4 Real-Time Workshop 5.0 Release Notes

5 Real-Time Workshop 4.1 Release Notes

6 Real-Time Workshop 4.0 Release Notes

1

Real-Time Workshop 6.0 Release Notes

New Features

This section introduces the new features and enhancements added in the Real-Time Workshop 6.0 since Real-Time Workshop 5.0.1. If you are upgrading from a release earlier than Version 5.0.1, then you should also see "New Features" on page 3-2 of the Real-Time Workshop 5.0.1 Release Notes.

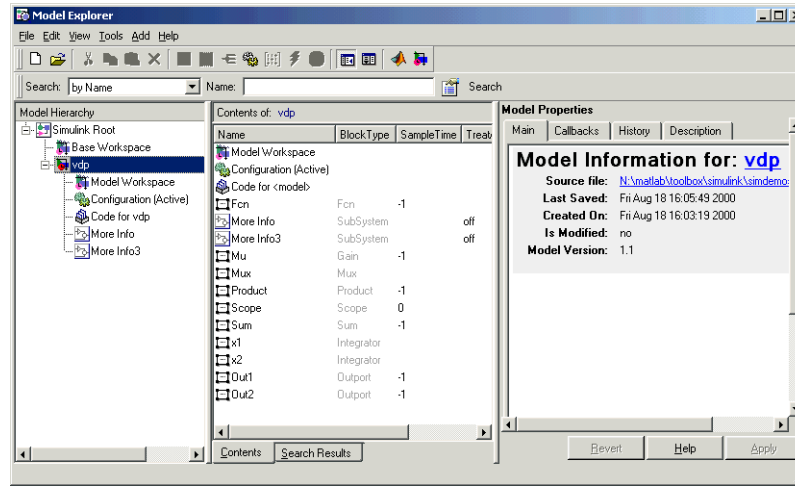
The new features are organized into the following categories:

Note The Real-Time Workshop User's Guide Documentation has not been fully updated for Release 14 Prerelease 2. However, most of the information you need in order to use Real-Time Workshop either can be found in the documentation set or is provided in this release note chapter.

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, as shown below:



The Model Explorer features three resizable, scrolling panes:

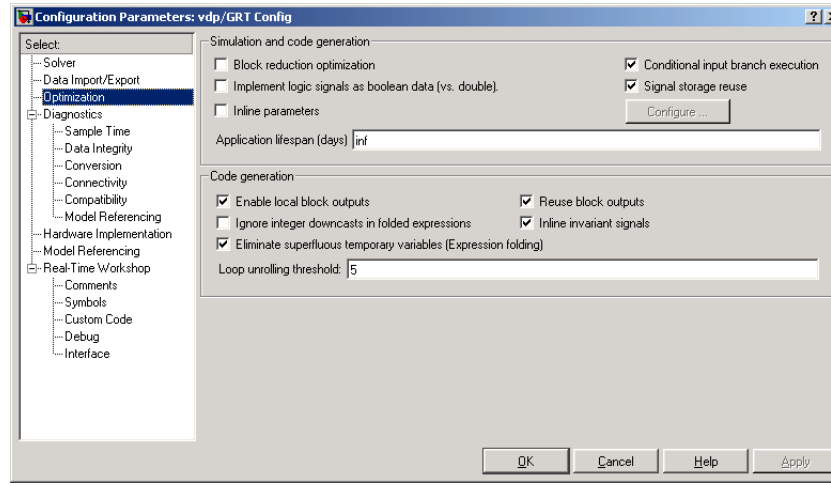
- **Model Hierarchy** pane
- **Contents** pane
- **Dialog** pane

See the Simulink Release Notes for a general description of the Model Explorer, which provides access to all blocks, subsystems, parameters, signals, workspace variables, and user-defined objects within a model.

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

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

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



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

- The Data Import/Export Configuration Component
- The Optimization Configuration Component
- The Diagnostics Configuration Component
- The Hardware Configuration Component
- Real-Time Workshop
  - Real-Time Workshop/General Tab
  - Real-Time Workshop/Debug Tab
  - Real-Time Workshop/Comments Tab

  - Real-Time Workshop/Interface Tab (specific to the current Real-Time Workshop target)
- The Model Referencing Configuration Component

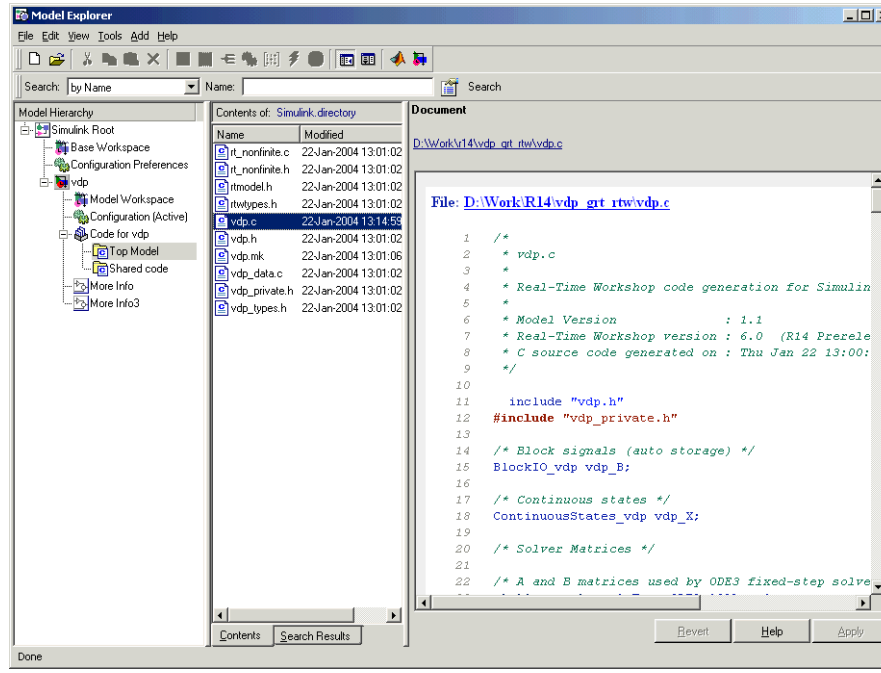
Note The controls visible in configuration component panes can change according to the target you are using and the options you select. This means that the interfaces shown below may not be identical to those that you see in a given context.

For information on configuration parameter options relevant to code generation, see "The Real-Time Workshop User Interface" 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 Hierarchy (left) Pane of Model Explorer contains a node named `Code for` *`model`*. Under that node will be other nodes, typically called `Top Model` and `Shared Code`. Clicking on `Top Model` displays in the Content (middle) Pane a list of source code files in the build directory of each model that is currently open. The figure below shows code for the `vdp` model:



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

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

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

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

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

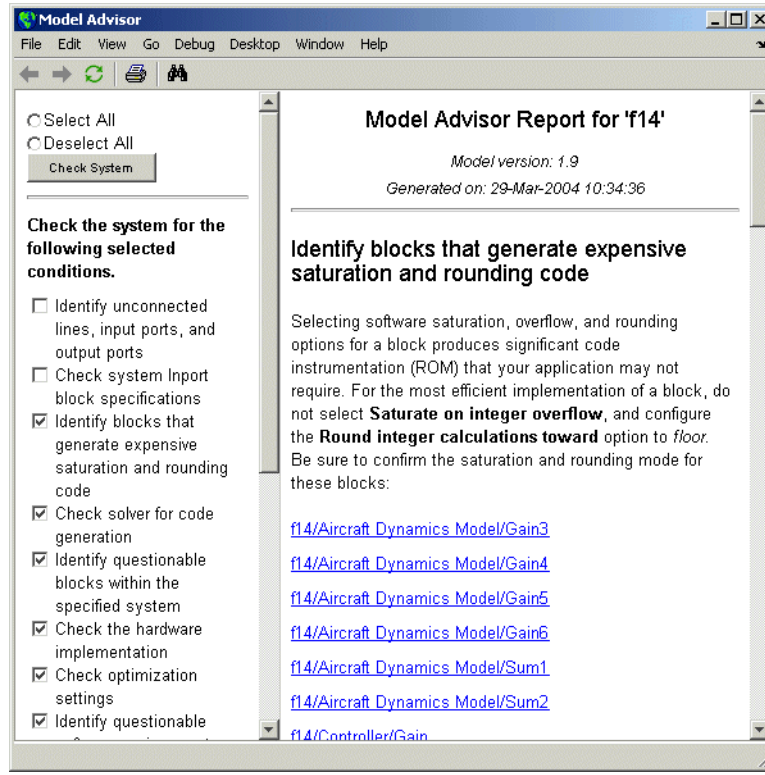
Model Advisor Helps You to Configure and Optimize Any Target

The Model Advisor (formerly called Model Assistant) is a tool that helps you configure any model to optimally achieve your code generation objectives. Using it, you can quickly configure a model for code generation, and identify aspects of your model that impede production deployment or limit code efficiency. Clicking the icon labeled `Advice on` *`model`* in the **Model Hierarchy** pane launches theModel Advisor From Model Explorer. This node is directly below the **Code for model** node, as the above figure shows. Clicking the Advice node causes the Dialog pane to be labelled 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 checkbox controls.

Another way to invoke Model Advisor is to type

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

specifying the name of model, at the MATLAB prompt. If the model (assumed to be on the MATLAB path) is not currently open, Model Advisor will open it. The following picture illustrates a Model Advisor report:



See “Using the Model Advisor” in the Real-Time Workshop documentation for further information on Model Advisor.

Real-Time Workshop Now Supports Intel Compiler

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

Model Referencing (Model Block) Enhancements

Including Models as Blocks in Simulations and in Generated Code

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

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

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

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

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

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

```
mdlrefdemos
```

The suite contains three separate demos:

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

For further information on generating code for referenced models, including using `mdlref_conversion`, see "Generating Code from Model Blocks" and "Tutorial 6: Generating Code for a Referenced Model" in the Real-Time Workshop documentation.

Signal, Parameter Handling and Interfacing Enhancements

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

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

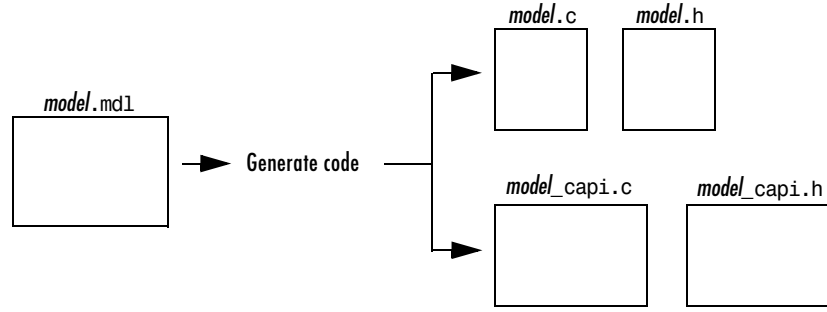
In previous releases, to access model parameters via the C-API, a model-specific parameter mapping file, `model_pt.c` was generated. Similarly, to access the BlockSignals, `model_bio.c` is generated. The new C-API improves the efficiency and capability of the interface while reducing its code size. In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

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

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

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

When you select the C-API feature and generate code, Real-Time Workshop generates two additional files, *model*`_capi.c` and *model*`_capi.h`, where "model" is the name of the model. Real-Time Workshop places the two C-API files in the build directory, based on settings on the **Configuration Parameters** dialog. The *model*`_capi.c` file contains information about global block signals and global parameters defined in the generated code. The *model*`_capi.h` file is an interface header file between the model source code and the generated C-API. You can use the information in these C-API files to create your application. The generated files are illustrated below.



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

CAPI Files	New C-API Files	Old C-API Files

Data structure interface	Unified interface for signals and parameters:
	`/rtw/c/src/rtw_capi.h`
		Signals Interface:
		`/rtw/c/src/bio_sig.h`
		Parameters Interface:
		`/rtw/c/src/pt_info.h`

RTModel C API Interface	`/rtw/c/src/rtw_modelmap.h`	`/rtw/c/src/mdl_info.h`
TLC files	`/rtw/c/tlc/mw/capi.tlc`	`/rtw/c/tlc/mw/biosig.tlc` `/rtw/c/tlc/mw/ptinfo.tlc`

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

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

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

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

1 In the open model, select **Configuration Parameters** on the **Simulation** menu.

2 Click **Interface** under **Real-Time Workshop** on the left pane.

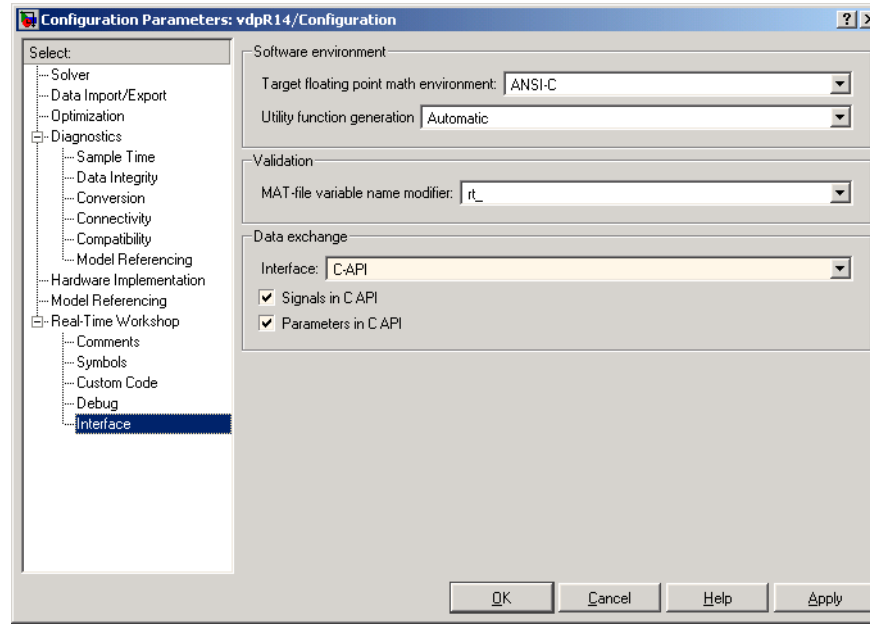
3 Select `C-API` in the **Interface** field. The **Signals in C API** and **Parameters in C API** check boxes appear, as shown below.

4 If you want to generate C-API for global block outputs, select the **Signals in C API** check box. If you want to generate C-API for global block and model parameters, select the **Parameters in C API** check box. If you select both check boxes, the default, both signals and parameters will appear in the C-API.

5 Click the **Apply** button.

6 Click **Real-Time Workshop** in the left pane. The **Generate code** button appears in the right pane.

7 Click **Generate Code**.



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

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

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

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

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

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

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

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

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

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

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

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

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

Declaring Wide Signals, States, and Parameters as ImportedExternPointer

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

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

The legacy code could define and initialize data as follows:

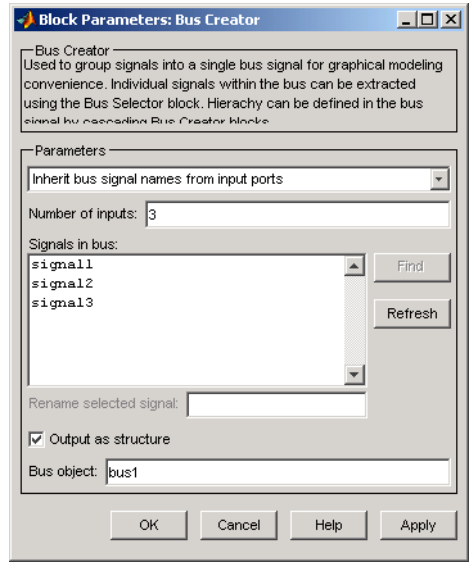
```
double x1_data;
double *x1 = &x1_data;
double x2_data[10];
double *x2 = x2_data;
```

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

Bus Creator Blocks Now Can Emit Structures

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

the MATLAB prompt. An example Bus Creator dialog for a block that outputs a three-element structure is shown below.



For additional details on working with Bus and other Simulink data objects, see the Simulink reference pages.

External Mode Enhancements

External Mode Changes May Impact Customized Makefiles and Static Main files

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

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

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

was changed to

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

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

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

Floating Scopes Now Work in External Mode

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

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

The behavior of wired Scope blocks is unchanged.

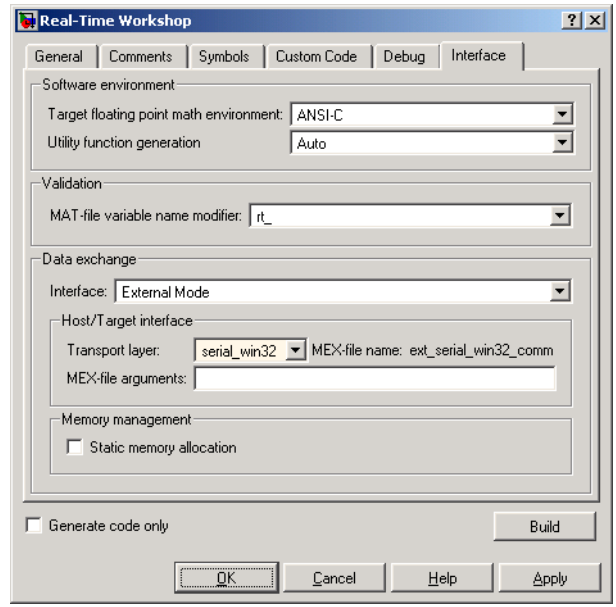
Serial Transport Mechanism for External Mode on Windows

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

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

To use serial data communications, you need to first instruct Real-Time Workshop to generate support code for external mode. Do this by selecting the **Interface** pane (which is sometimes labelled to specify the

current target) of the **Real-Time Workshop Configuration Parameters** dialog. First choose `External mode` from the **Interface** drop-down menu in the Data exchange section of the dialog. Next, in the Host/target interface subsection that then appears, select `serial_win32` for the **Transport layer**, as illustrated below:



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

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

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

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

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

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

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

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

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

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

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

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

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

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

- The functions `ExtTargetDataPending()` and `ExtTargetMsgPending()` have been merged into `ExtTargetPktPending()`.

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

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

New Static Memory Allocation Option for External Mode Code Generation

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

Code Customization Enhancements

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

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

Now, Real-Time Workshop adds an include path to the generaterd makefiles whenever it finds a file named `<s-function-name>.h` in the

same directory that the S-function MEX-file is in. This directory must be on the MATLAB path.

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

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

Custom Code Block Library Enhancements

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

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

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

Preventing User Source Code from Being Deleted from Build Directories

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

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

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

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

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

Generating Code with the Target Language Compiler Without Rebuilding

If you are developing custom TLC scripts for code generation, you can now decrease development time by speeding the edit-generate-inspect cycle when generating code for models that are not changing between iterations. You can bypass rebuilding the model (Ctrl-B if all you are doing is editing TLC files used in the latter part of the code generation process.

To use this new feature, select the **Retain .rtw file** option under the **Real-time Workshop/Debug** tab in the Model Explorer. The next time you build, the *model*.rtw file will be saved in your build directory, along with two other files, named runtlccmd.m and tlccmd.mat. From that point on, you can invoke the Target Language Compiler independently and with the proper parameters by executing runtlccmd.m. The MAT-file is used to store the parameters used by the M-file in issuing the TLC command. You can rebuild the model as required, and this time-saving option will remain available as long as you continue to retain your *model*.rtw file each time you build.

Designating Target-Specific Math Functions

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

Real-Time Workshop provides three different TFLs:

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

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

Selecting ANSI-C provides the ANSI-C set of library functions. For example, selecting ANSI-C would result in generated code that calls sin() whether the input argument is double precision or single precision. However, if ISO-C is selected, the call would instead be to the function sinf(), which is single-precision. If your compiler supports the ISO-C math extensions, selecting the ISO-C library can result in more efficient code.

Hook Files Describing Hardware Characteristics Are Deprecated

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

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

For additional details, see "Hardware Implementation Options" in the Real-Time Workshop documentation.

Timing-Related Enhancements

Application Lifespan Option Optimizes Timer Data Storage

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

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

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

- If your model contains fixed-point blocks that require absolute time, this option affects both simulation and generated code.

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

Enabling the Rapid Simulation Target to Time Out

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

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

```
vdp -L 20
```

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

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

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

On Unix platforms the message will be

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

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

New Asynchronous Block Library

The new VxWorks block library (`vxlib1`) allows you to model and generate code for asynchronous event handling, including servicing of

hardware generated interrupts, maintenance of timers, asynchronous read and write operations, and spawning of asynchronous tasks under a real-time operating system (RTOS).

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

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

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

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

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

- Async Interrupt block: Generates interrupt-level code. Each output of the Async Interrupt block is associated with a user-specified Vxworks VME interrupt. When an output is connected to the control input of a triggered subsystem such as a function-call subsystem, the generated subsystem code is called from an interrupt service routine (ISR).
- Task Synchronization block: a function-call subsystem that spawns an independent VxWorks task that calls the function-call subsystem connected to its output. The Task Synchronization block is designed to work in conjunction with the Async Interrupt block connected its control input.

- Protected Rate Transition block: The Protected Rate Transition block that is configured to ensure data integrity during data transfers between blocks running at different priorities.
- Unprotected Rate Transition block: The Unprotected Rate Transition block is configured to operate in unprotected / non-deterministic mode during data transfers between blocks running at different priorities.

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

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

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

```
vxlib1
```

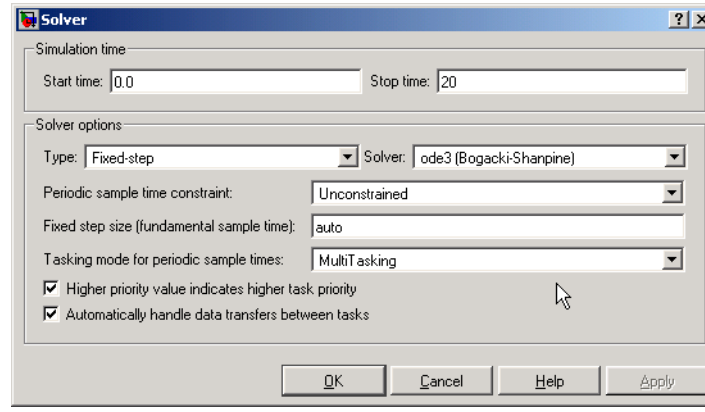
Rate Transition Block Improvements

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

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

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

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



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

Enhanced Absolute and Elapsed Time Computation

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

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

For further information see the "Timing Services" chapter of the Real-Time Workshop documentation.

Improved Singletasking Code Generation

New efficiencies in code generation no longer require code generated for singletasking 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 release 14 has been *target unification*. Target unification includes enhancements to the underlying technology and feature sets of both products, such that:

- Both products use a common backend generated code format. This enhancement, termed *code format unification*, has a number of implications (see "Code Format Unification" below).

- The set of features common to both products is expanded. Some features and efficiencies formerly exclusive to Real-Time Workshop Embedded Coder and the Embedded Real-Time (ERT) target are now generally available via the Generic Real-Time (GRT) target. Conversely, the Real-Time Workshop Embedded Coder now supports some features that were previously available only via the GRT target (for example, support of continuous-time blocks and noninlined S-functions).

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

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

Code Format Unification

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

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

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

characteristics make the `Embedded-C` code format the preferred choice for production code generation.

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

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

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

Since the GRT target now uses the `Embedded-C` code format for back-end 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 (see Table , Comparison of Features Licensed with Real-Time Workshop vs. Real-Time Workshop Embedded Coder, on page 1-35).

Code format unification simplifies the conversion of GRT-based custom targets to ERT-based targets. See "Compatibility Issues for GRT-Based Targets" on page 1-31 for a description of target conversion issues.

Compatibility Issues for GRT-Based Targets

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

There are several approaches to ERT compatibility:

- If your installation is not licensed for Real-Time Workshop Embedded Coder, you can convert a GRT-based target as described in "Converting Your Target to Use rtModel" on page 1-32. 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 1-34. This lets your target support the full ERT feature set, without changing your GRT-based runtime interface. This approach requires that your installation be licensed for Real-Time Workshop Embedded Coder.
- If your installation is licensed for Real-Time Workshop Embedded Coder, you can re-implement your custom target as a completely ERT-based target, including use of an ERT generated main program. This approach lets your target support the full ERT feature set, without the overhead caused by wrapper calls.

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

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

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

- Integer absolute and elapsed timing services
- Independent timers for asynchronous tasks

- Generation of improved C-API code for signal and parameter monitoring

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

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

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

```
%assign GenRTModel = TLC_TRUE
```

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

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

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

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

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

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

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

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

- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 14 version of `grt_main.c`.
- Change all function prototypes to have the argument `'RT_MODEL'` instead of the argument `'SimStruct'`.

- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`, `rt_UpdateDiscreteEvents`, `rt_UpdateDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. Change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass into them.

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

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

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

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

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

The approach you take to ERT compatibility will depend on the feature set required by your custom target. 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 and release notes.

Comparison of Features Licensed with Real-Time Workshop vs. Real-Time Workshop Embedded Coder

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

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

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
Comments generation	Basic options to include or suppress comment generation.	Options to include Simulink block descriptions, Stateflow object descriptions, and Simulink data object descriptions in comments.
Module Packaging Features (MPF)	Not supported.	Extensive code customization features. See 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.

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

Feature	Real-Time Workshop License	Real-Time Workshop Embedded Coder License
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.
Software-in-the-loop (SIL) testing	Model reference simulation target can be used for SIL testing.	Additional SIL testing support via auto-generation of Simulink S-Function block.
ANSI-C code generation	Supported	Supported
ISO-C code generation	Supported	Supported
GNU-C code generation	Supported	Supported
Generate scalar inlined parameters	Not supported	Supported
MAT-file variable name modifier	Supported	Supported
Data exchange: C-API, External Mode, ASAP2	Supported	Supported

Symbol Formatting Options Replaced

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

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

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

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

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

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

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

Model Referencing Considerations. Within a model that uses model referencing, there can be no collisions between the names of the constituent models. When generating code from a model that uses model referencing, the **Maximum identifier length** must be large enough to accommodate full the root model name and the name mangling string (if

any). A code generation error occurs if **Maximum identifier length** is not large enough.

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

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

Major Bug Fixes

Real-Time Workshop 6.0 includes several bug fixes made since Version 5.1. This section describes the particularly important Version 6.0 bug fixes. Some of these were fixed in beta versions, others more recently.

Lists of bug fixes for 5.0.1 and prior versions are available at the MathWorks web site Real-Time Workshop Release Notes pages.

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

If you are upgrading from a release earlier than Version 5.1, then you should also see “Major Bug Fixes” on page 2-3 of the Real-Time Workshop 5.1 Release Notes.

Upgrading from an Earlier Release

This section describes the upgrade issues involved in moving to Real-Time Workshop 6.0 from Version 5.0.

If you are upgrading from Release 13, you may wish to read the Real-Time Workshop 5.0.1 Release Notes.

If you are upgrading from Release 12, you may wish to read the Real-Time Workshop 5.0 Release Notes.

Global Data Identifiers for Targets Now Incorporate Model Name

Global data structures, such as `rtB`, `rtP` and `rtY` now have new identifiers in ERT and GRT generated code. For GRT, these names now include the model name followed by `_B`, `_P`, `_Y`, etc. (ERT targets provide you with flexible naming options; see "Symbol Formatting Options Replaced" on page 1-37). The construction of identifiers was changed to prevent name clashes when code for models containing Model blocks is generated and linked. If you are interfacing external code to any Simulink global data, you will probably need to use the GRT compatible calling interface for ERT-based targets (see "Generating GRT Wrapper Code from the ERT target" on page 1-34 for more information). The GRT interface enables you to access global data using the deprecated symbols via a set of macros that map old-style to new-style identifiers. See "Backwards Compatibility of Code Formats" in the Real-Time Workshop documentation for specific details.

Accessing the rtwOptions Structure Correctly

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

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

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

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

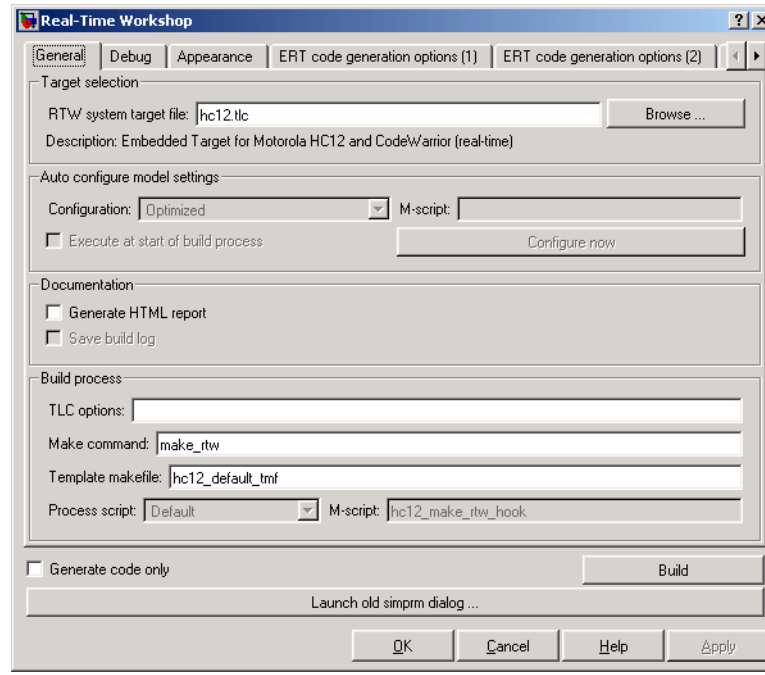
If you have written code that accesses the `rtwOptions` structure directly, as in the above example, you should update your code to use `getActiveConfigSet` instead. Due to changes in underlying data structures, code that accesses `rtwOptions` directly as above will no longer work correctly.

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

Defining and Displaying Custom Target Options

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

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



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

See the Real-Time Workshop Embedded Coder 4.0 release notes for details.

SelectCallback Function for System Target Files

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

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

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

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

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

If you have developed a custom target and you want it to be compatible with model referencing, you must implement a `SelectCallback` function to declare model reference compatibility. See “Model Reference Compatibility for Custom Targets” on page 1-47 for an example.

Supporting the Shared Utilities Directory in the Build Process

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

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

Targets" on page 1-47 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/archiver 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 sourcefiles. 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.

Model Reference Compatibility for Custom Targets

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

General Considerations

- A model reference compatible target must be derived from the ERT or GRT targets.
- When generating code from a model that references another model, both the top-level model and the referenced models must be configured for the same code generation target.

- Note that the **External mode** option is not supported in model reference Real-Time Workshop target builds. If the user has selected this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities directory, as described in "Supporting the Shared Utilities Directory in the Build Process" on page 1-44.

System Target File Modifications

- Your STF must implement a `SelectCallback` function (see "SelectCallback Function for System Target Files" on page 1-44). Your `SelectCallback` function must declare model reference compatibility by setting the `ModelReferenceCompliant` flag.

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

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

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

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

  See "Compatibility Issues for rtwoptions Callbacks" in the Real-Time Workshop Embedded Coder 4.0 release notes for details on the callback API, including `slConfigUISetVal`.

Template Makefile Modifications

In addition to the TMF modifications described in "Supporting the Shared Utilities Directory in the Build Process" on page 1-44, you must modify your TMF variables and rules as described below.

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

```
MODELREFS               = |>MODELREFS<|
MODELLIB                = |>MODELLIB<|
MODELREF_LINK_LIBS      = |>MODELREF_LINK_LIBS<|
MODELREF_INC_PATH       = |>MODELREF_INC_PATH<|
```

```
RELATIVE_PATH_TO_ANCHOR    = |>RELATIVE_PATH_TO_ANCHOR<|
MODELREF_TARGET_TYPE       = |>MODELREF_TARGET_TYPE<|
```

The following code excerpts show how makefile tokens are expanded for a referenced model, and for the top-level model that references it.

```
 Example of how tokens are expanded for a referenced model
MODELREFS                =
MODELLIB                 = engine3200cc_rtwlib.a
MODELREF_LINK_LIBS       =
MODELREF_INC_PATH        =
RELATIVE_PATH_TO_ANCHOR  = ../../..
MODELREF_TARGET_TYPE     = RTW

 Example of how tokens are expanded for the top-level model
MODELREFS                = engine3200cc transmission
MODELLIB                 = archlib.a
MODELREF_LINK_LIBS       = engine3200cc_rtwlib.a transmission_rtwlib.a
MODELREF_INC_PATH      = -I../slprj/ert/engine3200cc
-I../slprj/ert/transmission
RELATIVE_PATH_TO_ANCHOR  = ..
MODELREF_TARGET_TYPE     = NONE
```

The `MODELREFS` token for the top-level model expands to a list of referenced model names.

The `MODELLIB` token expands to the name of the library generated for the model.

The `MODELREF_LINK_LIBS` token for the top-level model expands to a list of referenced model libraries that the top-level model will link against.

The `MODELREF_LINK_LIBS` token for the top-level model expands to the include path to the referenced models.

The `RELATIVE_PATH_TO_ANCHOR` token expands to the relative path, from the location of the generated makefile, to the MATLAB working directory (`pwd`).

The `MODELREF_TARGET_TYPE` token signifies the type of target being built. Possible values are

- `NONE`: Standalone model or top-level model referencing other model(s).

- `RTW`: Model reference Real-Time Workshop target build.
- `SIM`: Model reference simulation target build.

2 Add `RELATIVE_PATH_TO_ANCHOR` and `MODELREF_INC_PATH` include paths to the overall `INCLUDES` variable.

```
INCLUDES = -I. -I$(RELATIVE_PATH_TO_ANCHOR) $(MATLAB_INCLUDES)
$(ADD_INCLUDES) \
                                $(USER_INCLUDES) $(MODELREF_INC_PATH)
$(SHARED_INCLUDES)
```

3 Change the `SRCS` variable in your TMF so that it initially lists only common modules. Further modules will then be appended conditionally, as described in step 4 below. For example, change

```
SRCS  = $(MODEL).c $(MODULES) ert_main.c $(ADD_SRCS) $(EXT_SRC)
```

to

```
SRCS = $(MODULES) $(S_FUNCTIONS)
```

4 Create variables to define the final target of the makefile. You can remove any variables that may have existed for defining the final target. For example, remove

```
PROGRAM    = ../$(MODEL)
```

and replace it with

```
ifeq ($(MODELREF_TARGET_TYPE), NONE)
  # Top-level model for RTW
  PRODUCT            = $(RELATIVE_PATH_TO_ANCHOR)/$(MODEL)
  BIN_SETTING        = $(LD) $(LDFLAGS) -o $(PRODUCT) $(SYSLIBS)
  BUILD_PRODUCT_TYPE = "executable"
  # ERT based targets
  SRCS               += $(MODEL).c ert_main.c $(EXT_SRC)
  # GRT based targets
  # SRCS             += $(MODEL).c grt_main.c rt_sim.c $(EXT_SRC) $(SOLVER)

else
  # sub-model for RTW
  PRODUCT            = $(MODELLIB)
  BUILD_PRODUCT_TYPE = "library"
endif
```

5 Create rules for final target of makefile (replace any existing final target rule). For example:

```
ifeq ($(MODELREF_TARGET_TYPE),NONE)
# Top-level model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_OBJS) $(MODELREF_LINK_LIBS) $(LIBS)
                            $(BIN_SETTING) $(LINK_OBJS) $(SHARED_OBJS)
$(MODELREF_LINK_LIBS) $(LIBS)
                            @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
else
# sub-model for RTW
$(PRODUCT) : $(OBJS) $(SHARED_OBJS)
                            @rm -f $(MODELLIB)
                            $(AR) ruv $(MODELLIB) $(LINK_OBJS)
                            @echo "### $(MODELLIB) Created"
                            @echo "### Created $(BUILD_PRODUCT_TYPE): $@"
endif
```

6 Create rule to allow submodels to compile files that reside in the MATLAB working directory (`pwd`).

```
%.o : $(RELATIVE_PATH_TO_ANCHOR)/%.c
                            $(CC) -c $(CFLAGS) $<
```

Custom Storage Classes Can No Longer Be Used with GRT Targets

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

For information on how GRT and ERT targets now compare, see “GRT and ERT Target Unification” on page 1-29. See the "Code Generation Options and Optimizations" chapter in the Real-Time Workshop Embedded Coder documentation for detailed information on the GRT compatible call interface option.

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

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

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

Known Software and Documentation Problems

Real-Time Workshop Documentation Status

The Real-Time Workshop Getting Started Guide has been fully updated for prerelease, and includes a new tutorial on generating code for referenced models. The Real-Time Workshop User's Guide is mostly updated for prerelease, and includes information on most new features described in this chapter. Some Simulink an The Real-Time Workshop Embedded Coder documentation is also relevant to Real-Time Workshop users. This chapter includes links to sections of those documents.

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

- "User Interface and Configuration Enhancements" on page 1-3
- "Model Referencing (Model Block) Enhancements" on page 1-9
- "Signal, Parameter Handling and Interfacing Enhancements" on page 1-10
- "External Mode Enhancements" on page 1-16
- "Code Customization Enhancements" on page 1-20
- "Timing-Related Enhancements" on page 1-24
- "GRT and ERT Target Unification" on page 1-29

Also refer the release notes in "Upgrading from an Earlier Release" for further details on compatibility issues between this and previous versions, particularly with respect to target customizations. You can find related details in the Real-Time Workshop Embedded Coder 4.0 release notes.

DSP Support Documentation Error

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

DSP targets may use registers with sizes other than 32 bits and vary in their saturation and overflow behavior. In Version 5 (Release 13), these characteristics were specified by target-specific hookfiles, which were

provided for all Version 5 targets supplied by The MathWorks. The `%assign DSP32=1` command to the system target makefile and the `-DDSP32=1` command to the template makefile that formerly handled DSP targets were deprecated in Version 5 and no longer have any effect. However, the documentation did not reflect that fact.

In the current version of Real-Time Workshop, hardware word sizes and other characteristics are specified on a per-processor basis using the **Hardware** configuration dialog. For further information, see the release note "Hook Files Describing Hardware Characteristics Are Deprecated" on page 1-24.

Blocks That Depend on Absolute Time

Appendix B of the Real-Time Workshop User's Guide lists Simulink blocks that depend on absolute time. In previous releases of this documentation, this list was incomplete. It should contain the following:

- Backlash
- Chirp Signal
- Clock
- Derivative
- Digital Clock
- Discrete-Time Integrator
- From File
- From Workspace
- Pulse Generator
- Ramp
- Rate Limiter
- Repeating Sequence
- Scope
- Signal Generator
- Sine Wave
- Step
- To File
- To Workspace

Model Parameter Configuration Dialog Source List Panel Description

The description of the **Source List Panel** of the **Model Parameter Configuration** dialog on p. 5-10 of the Real-Time Workshop User's Guide for Release 13 failed to describe one of the two menus on that panel, `Referenced workspace variables`. The updated description is as follows:

Source List Panel. The **Source list** panel displays a menu and a scrolling table of numerical workspace variables.

The menu lets you choose the source of the variables to be displayed in the list. There are two choices: `MATLAB workspace` (lists all variables in the MATLAB workspace that have numeric values), and `Referenced workspace variables` (lists only those variables referenced by the model). The source list displays names of variables defined in the MATLAB base workspace.

Error and Changes in Descriptions of Storage Class Declarations for Tunable Parameters

Table 5-2 in the Release 13 Real-Time Workshop Users Guide (Signal Properties Options and Generated Code) had errors. Furthermore, most of the identifiers for parameters have changed. In the current release, parameter structures are identified as *`model`*`_P` (where *`model`* stands for the—possibly mangled—name of the model), rather than as `rtP`. See the section "Storage Classes of Tunable Parameters" in the Real-Time Workshop documentation for a complete updated list of parameter declarations.

Error and Changes in Descriptions of Storage Class Declarations for Signal Properties

Table 5-5 in the Real-Time Workshop Users Guide (Signal Properties Options and Generated Code) had errors. As in the case of parameters, most of the identifiers for signals have changed. In the current release, `blockIO` structures are identified as *`model`*`_B` (where *`model`* stands for the—possibly mangled—name of the model), rather than as `rtB`. See the section "Summary of Signal Storage Class Options" in the Real-Time

Workshop documentation for a complete updated list of signal declarations.

Included Files Documentation Error

The section "Application Modules for Application Components" on page 7-33 of the Program Architecture chapter of the Release 13 Real-Time Workshop documentation incorrectly stated that `model_private.h` is sub-included by `model.h`.

The section “Building an Application: Summary of Files Created by the Build Procedure” in the Real-Time Workshop Getting Started guide also incorrectly states that `model.h` includes `model_private.h`.

The diagram of file inclusions in the figure on page 2-50 is correct, however.

No Code Generation Support for 64-bit Integer Values

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

Setting Environment Variable to Run Rapid Simulation Target Executables on Solaris

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

Limitation Affecting Rolling Regions of Noncontiguous Signals

This note describes a limitation affecting discontiguous signals that have regions that have a width greater than or equal to the **Loop unrolling**

threshold. This parameter is set in the **OPtimizations** pane of the **Configuration parameters** dialog.)

Such signal regions are called *rolling* regions.

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

Code Generation Failure in Nested Directories Under Windows 98

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

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

Turn the New Wrap Lines Option Off

The MATLAB Command Window has a new **Wrap lines** option. Real-Time Workshop frequently displays very long message lines as a build progresses. This can cause some display problems. Therefore, when using Real-Time Workshop, you can turn the **Wrap lines** option off using the **Preferences** setting. For more information on this issue, see the Technical Support Solution 29082 from the MathWorks Web page.

ASAP2 File Generation Changes

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

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

The ASAP2 file generation feature is available to Real-Time Workshop users who do not have a Real-Time Workshop Embedded Coder licence. So we have added the procedure, Generating ASAP2 Files Using the Generic Real-Time Target.

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

The following ASAP2 object properties have been replaced with standard Simulink object properties: `ASAP2.Parameter` and `ASAP2.Signal` property names have changed from `LONGIG_ASAP2` to `Description`, `PhysicalMin_ASAP2` to `Min`, `PhysicalMax_ASAP2` to `Max` and `Units_ASAP2` to `DocUnits`.

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

2

Real-Time Workshop 5.1 Release Notes

New Features

If you are upgrading from a release earlier than Version 5.0.1, then you should also see “New Features” on page 3-2 of the Real-Time Workshop 5.0.1 Release Notes.

Major Bug Fixes

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

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

If you are upgrading from a release earlier than Version 5.0.1, then you should also see “Major Bug Fixes” on page 3-4 of the Real-Time Workshop 5.0.1 Release Notes.

3

Real-Time Workshop 5.0.1 Release Notes

New Features

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

Expanded Hookfile Options

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

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

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

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

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

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

  This code requires the preprocessor to compare signed 64-bit integers. Some preprocessors have bugs that cause such comparisons to yield

incorrect results. The preprocessor math may only be fully correct for say 32-bit signed integers. To specify, this `PreprocMaxBitsSint` would be set to `32`. Generating the code with this setting causes problematic size checks to be skipped.

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

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

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

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

Hookfiles for Customizing Make Commands

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

Major Bug Fixes

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

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

If you are upgrading from a release earlier than Release 13, then you should also see “Major Bug Fixes” on page 4-24 of the Real-Time Workshop 5.0 Release Notes.

4

Real-Time Workshop 5.0 Release Notes

Release Summary

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

New Features and Enhancements

Code Generation Infrastructure Enhancements

Code Generation Configuration Features

Block-level Enhancements

Target and Mode Enhancements

TLC, model.rtw, and Library Enhancements

Documentation Enhancements

Major Bug Fixes

Upgrading from an Earlier Release

New Features and Enhancements

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

For information about Real-Time Workshop features that are incorporated from recent releases, see the "Real-Time Workshop 4.1 Release Notes" and the "Real-Time Workshop 4.0 Release Notes" documentation.

Note For information about closely related products that extend the Real-Time Workshop, see the Release Notes sections about the Real-Time Workshop Embedded Coder and the xPC Target.

Code Generation Infrastructure Enhancements

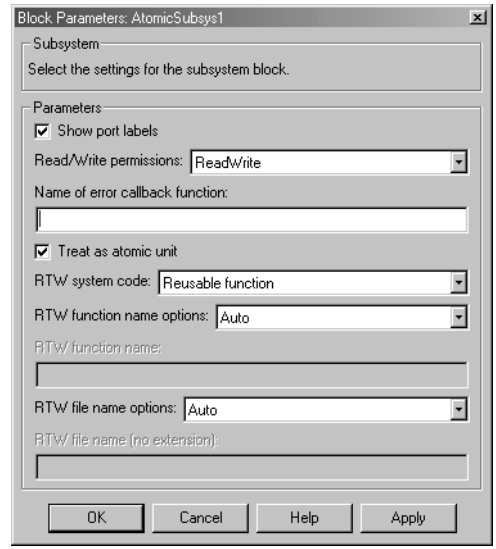
Code for Nonvirtual Subsystems Is Now Reusable

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

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

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

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



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

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

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

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

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

Packaging of Generated Code Files Simplified

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

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

Table 4-1: Real-Time Workshop File Packaging

File	Description
model`.c`	Contains entry points for all code implementing the model algorithm (`MdlStart`, `MdlOutputs`, `MdlUpdate`, `MdlInitializeSizes`, `MdlInitializeSampleTimes`). Also contains model registration code.
model`_private.h`	Contains local defines and local data that are required by the model and subsystems. This file is included by *subsystem*`.c` files in the model. You do not need to include *model*`_private.h` when interfacing hand-written code to a model.

Table 4-1: Real-Time Workshop File Packaging

File	Description
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 accessor macros. *model*.h is included by *subsystem*.c files in the model. If you are interfacing your hand-written 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.
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. To learn how to generate and use the model_pt.c file, see "C API for Parameter Tuning."
model_bio.c (optional)	Provides data structures that enable your code to access block outputs. To learn how to generate and use the model_bio.c file, see "Signal Monitoring via Block Outputs."

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

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

Most Targets Use rtModel Instead of Root SimStruct

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

Note If you have previously customized GRT, GRT-Malloc, or Tornado targets, you should upgrade each customized target to use the rtModel instead of the SimStruct. You can find guidelines for this upgrade path in "Upgrading Customized GRT and GRT-Malloc Targets to Work with Release 13" on page 4-32.

Hook Files for Communicating Target-specific Word Characteristics

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

Code Generation Unified for Real-Time Workshop and Stateflow

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

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

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

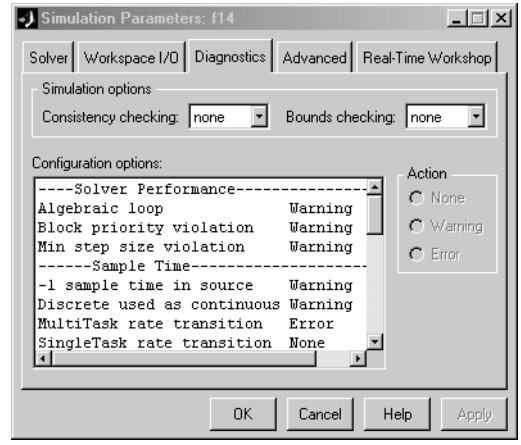
Conditional Input Branch Execution Optimization

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

Code Generation Configuration Features

Diagnostics Pane Items Classified into Logical Groups

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



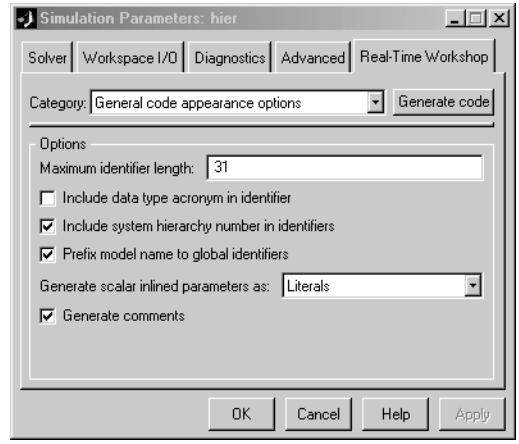
Comments Not Generated for Reduced Blocks When "Show eliminated statements" Is Off

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

New General Code Appearance Options

A new category has been added to the **Real-Time Workshop** dialog box, named `General code appearance options`. This pane adds four new

code formatting options to two existing options that formerly occupied other categories. The `General code appearance` dialog is shown below.



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

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

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

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

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

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

The default is to generate scalar inline parameters as literals.

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

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

Identifier Construction for Generated Code Has Been Simplified

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

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

- Use of _a, _b, … postfixes to identifiers to prevent name clashes has been dramatically reduced.

See also "New General Code Appearance Options" on page 4-12 for related information.

GUI Control over Behavior of Assertion Blocks in Generated Code

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

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

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

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

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

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

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

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

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

GUI Control Over TLC %assert Directive Evaluation

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

`set_param(`*`model`*`, 'TLCAssertion', 'on|off')` to set or unset. Default is Off.

`get_param(`*`model`*`, 'TLCAssertion')` to see the current setting.

Block-level Enhancements

New Rate Transition Block

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

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

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

The Rate Transition block supports the following modes of operation:

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

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

  The drawbacks of this mode are its non-deterministic timing and its use of extra memory buffers. The advantage of this mode is its low latency.
- Unprotected/Non-Deterministic: This mode is the least safe, and is not recommended for mission-critical applications. The latency of this mode is the same as for Protected/Non-Deterministic mode, but memory requirements are reduced since there is no double-buffering.

For further information on the use of the Rate Transition block with the Real-Time Workshop, see “Models With Multiple Sample Rates” in the Real-Time Workshop User’s Guide. For information on the use of the Rate Transition block GUI and its use in simulation, see Using Simulink.

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

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

- `ssGetDWorkRTWIdentifier(S,idx)`
- `ssSetDWorkRTWIdentifier(S,idx,val)`

- `ssGetDWorkRTWStorageClass(S,idx)`
- `ssSetDWorkRTWStorageClass(S,idx,val)`
- `ssGetDWorkRTWTypeQualifier(S,idx)`
- `ssSetDWorkRTWTypeQualifier(S,idx,val)`

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

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

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

Relay Block Now Supports Frame-based Processing

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

Transport Delay and Variable Transport Delay Improvements

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

Storage Classes for Data Store Memory Blocks

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

See “Storage Classes for Data Store Memory Blocks” in the Real-Time Workshop User’s Guide for further information.

Target and Mode Enhancements

Rapid Simulation Target Now Supports Variable-step Solvers

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

External Mode Support for Rapid Simulation Target

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

External Mode Support for ERT

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

External Mode Supports Uploading Signals of All Storage Classes

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

Expanded Support for Borland C Compilers

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

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

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

TLC, model.rtw, and Library Enhancements

New Simulink Data Object Properties Mapped to model.rtw Files

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

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

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

SPRINTF Built-in Function Added to TLC

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

`$assign str = SPRINTF(format,var,...)` formats the data in variable `var` (and in any additional variable arguments) under control of the specified `format` string, and returns a string variable containing the

values. Operates like C library `sprintf()`, except that output is the return value rather than contained in an argument to sprintf.

LCC Now Links Libraries in Directory sys/lcc/lib

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

The BlockInstanceData Function has been Deprecated

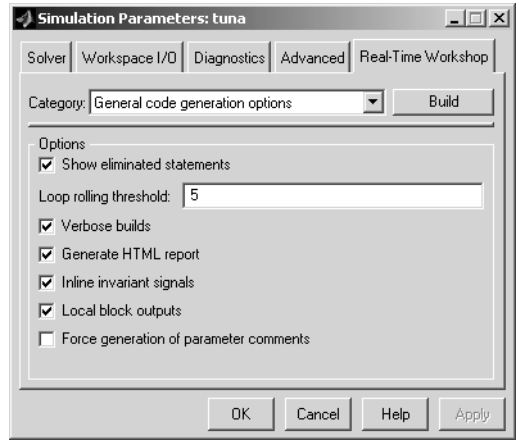
S-function TLC files should no longer use the BlockInstanceData method. All data used by a block should be declared using data type work vectors (DWork).

Documentation Enhancements

Generate HTML Report Option Available for Additional Targets

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

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



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

Expression Folding API Documentation Available

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

Real-Time Workshop Documentation

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

Target Language Compiler Documentation

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

Major Bug Fixes

Real-Time Workshop 5.0 includes several bug fixes made since Version 4.1. This section describes the particularly important Version 5.0 bug fixes. If you are upgrading from a release earlier than Release 12.1, then you should also see "Bug Fixes" on page 5-11 of the Release 12.1 Release Notes.

ImportedExtern and ImportedExternPointer Storage Class Data No Longer Initialized

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

External Mode Properly Handles Systems with no Uploadable Blocks

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

Nondefault Ports Now Usable for External Mode on Tornado Platform

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

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

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

Code Is Generated Without Errors for Single Precision Datatype Block Outputs

In cases where a reused block outputs entry is the first single-precision datatype 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. We urge you to update your custom code formats to use the `(S)LibAddToCommonIncludes()` functions instead of `LibCacheIncludes()`, which has been deprecated.

Custom Storage Classes Ignored When Unlicensed for Embedded Coder

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

Erroneous Sample Time Warning Messages No Longer Issued

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

Discrete Integrator Block with Rolled Reset No Longer Errors Out

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

Rate Limiter Block Code Generation Limitation Removed

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

Multiport Switch with Expression Folding Limitation Removed

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

Pulse Generator Code Generation Failures Rectified

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

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

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

Stateflow I/O with ImportedExternPointer Storage Class Now Handled Correctly

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

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

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

PreLook-up Index Search Block Now Handles Discontiguous Wide Input

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

SimViewingDevice Subsystem No Longer Fails to Generate Code

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

Accelerator Now Works with GCC Compiler on UNIX

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

Expression Folding Behavior for Action Subsystems Stabilized

When a model contains an action subsystem (e.g., a for-loop or while-iterator subsystem) and expression folding is enabled, invalid or

inefficient code sometimes was generated for the model. This problem has been fixed.

Dirty Flag No Longer Set During Code Generation

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

Subsystem Filenames Now Completely Checked for Illegal Characters

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

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

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

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

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

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

Note This is also a fix for the Simulink Accelerator.

Report Error when Code Generation Requested for Models with Algebraic Loops

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

Platform Limitations for HP and IBM

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

On the HP and IBM platforms, the Real-Time Workshop opens the Release 11 **Tunable Parameters** dialog 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.

Upgrading from an Earlier Release

This section describes the upgrade issues involved in moving from Real-Time Workshop 4.1 to Version 5.0.

If you are upgrading from a version earlier than 4.1, then you should see "Upgrading from an Earlier Release" on page 5-15 in the Real-Time Workshop 4.1 Release Notes.

Replacing Obsolete Header File #includes

Generated code is packaged into fewer files in this release (see "Packaging of Generated Code Files Simplified" on page 4-8). If you have interfaced code to code generated by previous releases of Real-Time Workshop, you may need to remove dependencies on header files that are no longer generated (such as *model*_common.h, *model*_export.h, *model*_prm.h, and *model*_reg.h) and add #include *model*.h directives.

Custom Code Blocks Moved from Simulink Library

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

Updating Custom TLC Code

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

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

Substantial changes have been made to the GRT and GRT-Malloc targets in Release 13 to improve the efficiency of generated code. If you have customized either type of target, you should make changes to your

modified files to ensure that your target works properly with Release 13 (Real-Time Workshop Version 5.0).

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

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

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

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

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

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

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

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

- Change the definitions of `rt_CreateIntegrationData` and `rt_UpdateContinuousStates` to be as shown in the Release 13 version of `grt_main.c` (or `grt_malloc_main.c`).
- Change all function prototypes to have the argument `'RT_MODEL'` instead of the argument `'SimStruct'`.
- The prototypes for the functions `rt_GetNextSampleHit`, `rt_UpdateDiscreteTaskSampleHits`, `rt_UpdateContinuousStates`,

`rt_UpdateDiscreteEvents`, `rt_UpdateDiscreteTaskTime`, and `rt_InitTimingEngine` have changed. You need to change their names to use the prefix `rt_Sim` instead of `rt_` and then change the arguments you pass into them.

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

- You need to modify the all macros that refer to the SimStruct to now refer to the rtModel. Examples of these modifications include changing
  - `ssGetErrorStatus` to `rtmGetErrorStatus`
  - `ssGetSampleTime` to `rtmGetSampleTime`
  - `ssGetSampleHitPtr` to `rtmGetSampleHitPtr`
  - `ssGetStopRequested` to `rtmGetStopRequested`
  - `ssGetTFinal` to `rtmGetTFinal`
  - `ssGetT` to `rtmGetT`

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

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

```
%assign GenRTModel = TLC_TRUE
```

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

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

- Remove `rtwlog.c` from all of your template make files.
- In your target's main C file (which was derived from `grt_main.c` or `grt_malloc_main.c`), include `rt_logging.h` instead of `rtwlog.h`.

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

The BlockInstanceData Function has been Deprecated

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

5

Real-Time Workshop 4.1 Release Notes

Release Summary

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

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

New Features

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

For information about Real-Time Workshop features that are incorporated from recent releases, see "Release Summary" on page 6-2.

Note For information about closely related products that extend the Real-Time Workshop, see the Release Notes sections about the Real-Time Workshop Embedded Coder and xPC Target.

Block Reduction Option On by Default

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

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

See "Block Reduction Option" in the *Real-Time Workshop User's Guide* for further information.

Buffer Reuse Code Generation Option

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

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

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

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

See "Buffer Reuse Option" and "Signals: Storage, Optimization, and Interfacing" in the *Real-Time Workshop User's Guide* for further information.

Build Directory Validation

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

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

Build Subsystem Enhancements

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

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

See "Generating Code and Executables from Subsystems" in the *Real-Time Workshop User's Guide* for further information.

C API for Parameter Tuning Documented

The Real-Time Workshop provides data structures and a C API that enable a running program to access model parameters without use of external mode. This API has now been fully documented.

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

See "C API for Parameter Tuning" in the *Real-Time Workshop User's Guide* for information on how to generate and use the parameter mapping file.

Code Readability Improvements

Improvements to the readability of generated code include:

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

Control Flow Blocks Support

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

For further information on the flow control blocks, see "Control Flow Statements" in *Using Simulink*.

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 further information, see "Expression Folding" in the *Real-Time Workshop User's Guide.*

External Mode Enhancements

Inline Parameters Support

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

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

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

See "Overview of External Mode Communications" in the *Real-Time Workshop User's Guide* for further information.

Status Bar Display

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

Generate Comments Option

This option lets you control whether or not comments are written in the generated code. See "Generate Comments" in the *Real-Time Workshop User's Guide* for further information.

Include System Hierarchy in Identifiers

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

See "Include System Hierarchy in Identifiers" in the *Real-Time Workshop User's Guide* for further information.

Rapid Simulation Target Supports Inline Parameters

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

S-Function Target Enhancements

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

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

See "Automated S-Function Generation" in the *Real-Time Workshop User's Guide* for details.

Storage Classes for Block States

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

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

For further information, see "Block States: Storing and Interfacing" in the *Real-Time Workshop User's Guide*.

Support for tilde (~) in Filenames on UNIX Platforms

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

Target Language Compiler 4.1

This section summarizes changes that have been made to the Target Language Compiler in this release. See also "TLC Compatibility Issues" on page 5-16.

New TLC Debugger

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

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

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

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

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

For further information, see "Debugging TLC" in the *Target Language Compiler Reference Guide*.

model.rtw File Format Changes

The format of the *`model`*`.rtw` file has changed. See "TLC Compatibility Issues" on page 5-16.

Cleanup of Block I/O Connection Handling in TLC

The handling of signal connections in `rtw/c/tlc/blkiolib.tlc` and `rtw/ada/tlc/blkiolib.tlc` was reworked. See the discussion of `LibBlockInputSignal` in "TLC Function Library Reference" in the *Target Language Compiler Reference Guide*.

Literal String Support

If a string constant is prefixed by the `L` format specifier, then no escape character processing is performed on that string. This is useful for specifying PC style path or directory names, as in this example.

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

New TLC Library Functions

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

- `LibBlockInputSignalConnected`
- `LibBlockInputSignalLocalSampleTimeIndex`
- `LibBlockInputSignalOffsetTime`
- `LibBlockInputSignalSampleTime`
- `LibBlockInputSignalSampleTimeIndex`
- `LibBlockOutputSignalOffsetTime`
- `LibBlockOutputSignalSampleTime`
- `LibBlockOutputSignalSampleTimeIndex`
- `LibBlockMatrixParameterBaseAddr`
- `LibBlockParameterBaseAddr`
- `LibBlockNonSampledZC`

See "TLC Function Library Reference" in the *Target Language Compiler Reference Guide* for information on these functions.

TLC Bug Fixes

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

- Zero indexing on complex numbers is now supported.

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

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

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

Bug Fixes

The Real-Time Workshop 4.1 includes the bug fixes described in this section. See also "TLC Bug Fixes" on page 5-9 for bug fixes specific to the Target Language Compiler.

Block Reduction Crash Fixed

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

Build Subsystem Gives Better Error Message for Function Call Subsystems

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

Check Consistency of Parameter Storage Class and Type Qualifier

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

Code Optimization for Unsigned Saturation and DeadZone Blocks

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

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

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

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

Correct Compilation with Green Hills and DDI Compilers

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

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

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

Fixed Error Copying Custom Code Blocks

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

Fixed Error in commonmaplib.tlc

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

Fixed Name Clashes with Run-Time Library Functions

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

Improved Handling of Sample Times

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

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

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

Parenthesize Negative Numerics in Fcn Block Expressions

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

```
-v*u
```

Such expressions are now enclosed in parentheses, as in

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

Removed Unnecessary Warnings and Declarations from Generated Code

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

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

  ```
  (void)tid
  ```

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

Retain .rtw File Option Now Works in Accelerator Mode

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

S-Function Target Memory Allocation Bug Fix

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

Upgrading from an Earlier Release

This section describes the upgrade issues involved in moving from the Real-Time Workshop 4.0 (Release 12.0) to the Real-Time Workshop 4.1.

For information about upgrading from a release earlier than 4.0, see the "Upgrading from an Earlier Release" on page 6-12 in the Real-Time Workshop 4.0 Release Notes.

RTWInfo Property Changes

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

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

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

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

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

or

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

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

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

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

For further information see “Simulink Data Objects and Code Generation” in the *Real-Time Workshop User's Guide*.

S-Function Target MEX-Files Must Be Rebuilt

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

If you have built S-function MEX-files with the S-function target under Release 11, you must rebuild them. See “The S-Function Target” in the *Real-Time Workshop User's Guide* for further information.

TLC Compatibility Issues

model.rtw File Format Changes

The format of the *model*`.rtw` file has changed. For further information, see “model.rtw Changes Between Real-Time Workshop 4.0 and 4.1” in the *Target Language Compiler Reference Guide*.

Reordering of BlockTypeSetup and BlockInstanceSetup Calls

During the initialization phase of code generation, the Target Language Compiler makes a pass over all blocks in the model and executes several functions, including:

- Each block's `BlockTypeSetup` function the first time that block type is encountered.
- Each block's `BlockInstanceSetup` function. `BlockInstanceSetup` is called for all instances of a given block type in the model.

The order in which these calls are made is significant, because the `BlockInstanceSetup` function may depend upon global variables that are initialized by the `BlockTypeSetup` function.

In Release 12.1, the `BlockTypeSetup` function is called before the `BlockInstanceSetup`. This corrects a problem in previous releases, where `BlockInstanceSetup` was erroneously called first. You may need to change your S-functions or block implementations if they depend upon the previous behavior.

Obsolete Code Generation Variables

The code generation variables `FunctionInlineType` and `PragmaInlineString` are now obsolete. These variables controlled the generation of inlined functions. In the current release, you can generate inlined functions from subsystems, as described in "Nonvirtual Subsystem Code Generation" in the *Real-Time Workshop User's Guide*.

6

Real-Time Workshop 4.0 Release Notes

Release Summary

Release 4.0 of the Real-Time Workshop is a major upgrade, incorporating significant new and enhanced features and many improvements in the quality of generated code. These include:

- Significantly faster Target Language Compiler (TLC) code generation process
- TLC Profiler report for debugging TLC programs
- New efficiencies in generated code include improved signal storage reuse, constant block elimination, and parameter pooling.
- New Real-Time Workshop Embedded Coder add-on product replaces and significantly enhances the Embedded Real-Time (ERT) target.
- User interface improvements, including a redesigned Real-Time Workshop page and Model Parameter Configuration (tunable parameters) dialog
- Support for additional Simulink blocks, including Look-Up table blocks with very efficient generated code
- S-Function Target support for variable step solvers and parameter tuning
- Support for matrix operations for most Simulink blocks
- Support for frame-based processing for DSP blocks
- External mode support for many additional block types for signal uploading
- Automatic generation of S-function wrappers for embedded code, allowing for validation of generated code in Simulink
- Support for generation of code and executables from subsystems
- Support for Simulink data objects in code generation
- Support for generation of ASAP2 data definition files

New Features

This section introduces the new features and enhancements added in the Real-Time Workshop 4.0 since the Real-Time Workshop 3.0.1.

Real-Time Workshop Embedded Coder

The Real-Time Workshop Embedded Coder is a new add-on product that replaces and enhances the Embedded Real-Time (ERT) target.

The Real-Time Workshop Embedded Coder is 100% compatible with the ERT target. In addition to supporting all previous functions of the ERT target, the Real-Time Workshop Embedded Coder includes many enhancements.

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

Simulink Data Object Support

The Real-Time Workshop supports the new Simulink data objects feature. Simulink provides the built-in `Simulink.Parameter` and `Simulink.Signal` classes for use with the Real-Time Workshop. Using these classes, you can create parameter and signal objects and assign storage classes and storage type qualifiers to the objects. These properties control how the generated code represents signals and parameters. The `Simulink.Parameter` and `Simulink.Signal` classes can be extended to include user-defined properties.

See "Simulink Data Objects and Code Generation" in the *Real-Time Workshop User's Guide* for complete details.

ASAP2 Support

ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). This standard is used for data measurement, calibration, and diagnostic systems.

The Real-Time Workshop now lets you export an ASAP2 file containing information about your model during the code generation process. See "Generating ASAP2 Files" in the Real-Time Workshop online documentation.

Enhanced Real-Time Workshop Page

The Real-Time Workshop page of the **Simulation Parameters** dialog box has been reorganized and made easier to use. See "Overview of the Real-Time Workshop User Interface" in the *Real-Time Workshop User's Guide* for complete details.

Other User Interface Enhancements

The **Tools** menu of the Simulink window now contains a **Real-Time Workshop** submenu with shortcuts to frequently used features. See "Real-Time Workshop Submenu" in the *Real-Time Workshop User's Guide* for details.

You can now select a target configuration from the System Target File Browser by double-clicking on the desired entry in the target list. The previous selection method — selecting an entry and clicking the **OK** button — is still supported.

Advanced Options Page

An Advanced options page has been added to the **Simulation Parameters** dialog box. The Advanced page contains new code generation options, as well as options formerly located in the Diagnostics and Real-Time Workshop pages. See "Advanced Options Page" for details.

Model Parameter Configuration Dialog

The **Model Parameter Configuration** dialog box replaces the **Tunable Parameters** dialog box. The **Model Parameter Configuration** dialog box enables you to declare individual parameters to be tunable and to control the generated storage declarations for each parameter. See "Parameters: Storage, Interfacing and Tuning" in the *Real-Time Workshop User's Guide* for details.

Tunable Expressions Supported

A tunable expression is an expression that contains one or more tunable parameters. Tunable expressions are now supported during simulation and in generated code.

Tunable expressions are allowed in masked subsystems. You can use tunable parameter names or tunable expressions in a masked subsystem dialog. When referenced in lower-level subsystems, such parameters remain tunable.

See “Tunable Expressions” in the *Real-Time Workshop User’s Guide* for a detailed description of the use of tunable parameters in expressions.

S-Function Target Enhancements

S-function target enhancements include:

- The S-function target now supports variable-step solvers.
- The S-function target now supports tunable parameters.
- The new **Generate S-function** feature lets you automatically generate an S-function from a subsystem.

The S-function target is now documented the *Real-Time Workshop User’s Guide.* See the chapter “The S-Function Target” for a full description of S-function target features.

External Mode Enhancements

Several new features have been added to external mode:

- The default operation of the **External Signal & Triggering** dialog box has been changed to make monitoring the target program simpler. See “External Signal & Triggering Dialog Box” in the *Real-Time Workshop User’s Guide* for details.
- Signal Viewing Subsystems have been implemented to let you encapsulate processing and viewing of signals received from the target system. Signal Viewing Subsystems run only on the host, generating no code in the target system. This is useful in situations where you want to process or condition signals before viewing or logging them, but you do not want to perform these tasks on the target system. See “Signal Viewing Subsystems” in the *Real-Time Workshop User’s Guide* for details.
- Previously, only Scope blocks could be used in external mode to receive and view signals uploaded from the target program. The following now support external mode:
  - Dials & Gauges Blockset
  - Display blocks
  - To Workspace blocks
  - Signal Viewing Subsystems

- S-functions

  See "External Mode Compatible Blocks and Subsystems" in the *Real-Time Workshop User's Guide* for details.

- In Release 12, we have documented the external mode communications application program interface (API). If you want to implement external mode communications via your own low-level protocol, see "Creating an External Mode Communications Channel" in the *Real-Time Workshop User's Guide*.

Build Directory

The Real-Time Workshop now creates a *build directory* within your working directory. The build directory stores generated source code and other files created during the build process. The build directory name, `model_target_rtw`, derives from the name of the source model and the chosen target.

See "Directories Used in the Build Process" in the *Real-Time Workshop User's Guide* for details.

Note If you have created custom targets for the Real-Time Workshop under Release 11, you must update your custom system target files and template makefiles to create and utilize the build directory. See "Updating Release 11 Custom Targets" on page 6-12.

Code Optimization Features

This section describes new or modified code generation options that are designed to help you optimize your generated code. The options described are located on the Advanced page of the **Simulation Parameters** dialog box. See "Advanced Options Page" for full details.

- **Block reduction**: When the **Block reduction** option is selected, Simulink collapses certain groups of blocks into a single, more efficient block, or removes them entirely. This results in faster model execution during simulation and in generated code.
- **Parameter pooling**: When multiple block parameters refer to storage locations that are separately defined but structurally identical, you can use this option to save memory.

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

See the chapter "Optimizing the Model for Code Generation" in the *Real-Time Workshop User's Guide* for further information on code optimization.

Subsystem Based Code Generation

The Real-Time Workshop now generates code and builds an executable from any subsystem within a model. The build process uses the code generation and build parameters of the root model. See "Generating Code and Executables from Subsystems" in the *Real-Time Workshop User's Guide* for details.

Nonvirtual Subsystem Code Generation

The Real-Time Workshop now lets you generate code modules at the subsystem level. This feature applies only to nonvirtual subsystems. With nonvirtual subsystem code generation, you control how many files are generated, as well as the file and function names. To set options for nonvirtual subsystem code generation, you use the subsystem's **Block Parameters** dialog.

Nonvirtual subsystem code generation is a more general and flexible method of controlling the number and size of generated files than the **Function management** code generation options (**File splitting** and **Function splitting**) used in previous releases. The **Function management** code generation options have been replaced by nonvirtual subsystem code generation.

See "Nonvirtual Subsystem Code Generation" in the *Real-Time Workshop User's Guide* for details.

Filename Extensions for Generated Files

In previous releases, some generated files were given special filename extensions, such as `.prm` or `.reg`. All the Real-Time Workshop generated code and header files now use standard filename extensions (`.c` and `.h`). The file naming conventions for the following generated files have changed:

- Model registration file (formerly `model.reg`) is now named `model_reg.h`.
- Model parameter file (formerly `model.prm`) is now named `model_prm.h`.
- `BlockIOSignals` struct file (formerly `model.bio`) is now named `model_bio.c`.
- `ParameterTuning` file (formerly `model.pt`) is now named `model_pt.c`.
- External mode data type transition file (formerly `model.dt`) is now named `model_dt.c`.

hilite_system and Code Tracing

The Real-Time Workshop writes system/block identification tags in the generated code. The tags are designed to help you identify the block, in your source model, that generated a given line of code. In previous releases, the `locate_system` command was used to trace a tag back to the generating block.

The new `hilite_system` command replaces `locate_system`, for the purposes of tracing the Real-Time Workshop identification tags. You should use the `hilite_system` command to trace a tag back to the generating block. For further information on identification tags and code tracing, see "Tracing Generated Code Back to Your Simulink Model."

Generation of Parameter Comments

The **Force generation of parameter comments** option in the **General code generation options** category of the Real-Time Workshop page controls the generation of comments in the model parameter structure (`rtP`) declaration in `model_prm.h`. This lets you reduce the size of the generated file for models with a large number of parameters. See "Force Generation of Parameter Comments Option" in the *Real-Time Workshop User's Guide* for details.

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.

See “Customizing the Makefile Include Path” in the Real-Time Workshop online documentation for details.

Target Language Compiler 4.0

TLC File Parsing Before Execution

The Target Language Compiler 4.0 completes parsing of the TLC file just before execution. This aids development because syntax errors are caught the first time the TLC file is run instead of the first time the offending line is reached.

Enhanced Speed

The Target Language Compiler 4.0 features speed improvements throughout the software. In particular, the speed of block parameter generation has been enhanced.

Build Directory

The Target Language Compiler 4.0 creates and uses a `build` directory. The `build` directory is in the current directory and prevents generated code from clashing with other files generated for other targets, and keeps your model directories maintenance to a minimum.

TLC Profiler

An entirely new TLC Profiler has been added to the Target Language Compiler to help you find performance problems in your TLC code.

model.rtw Changes

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

Block Parameter Aliases

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

Improved Text Expansion

This release of the Target Language Compiler contains new, flexible methods for text expansion from within strings.

Column-Major Ordering

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

Improved Record Handling

The Target Language Compiler 4.0 utilizes new record data handling.

New TLC Language Semantics

Many changes have been made to the language including:

- Improved `EXISTS` behavior (see "TLC Compatibility Issues" on page 6-13)
- New TLC primitives for record handling
- Functions can return records.
- Records can be printed.
- Records can be empty.
- Record aliases are available.
- 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. (See "FEVAL Function" in the Target Language Compiler documentation for more information.)
- Relational operators can be used with nonfinite values.
- Loop control variables are local to loop bodies.

New Built-In Functions

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

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

New Built-In Values

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

`INTMAX`, `INTMIN`, `TLC_FALSE`, `TLC_TRUE`, `UINTMAX`

Added Support for Inlined Code

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

Upgrading from an Earlier Release

This section describes the upgrade issues involved in moving from the Real-Time Workshop 3.0 (Release 11.0) to the Real-Time Workshop 4.0.

Column-Major Matrix Ordering

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

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

Including Generated Files

Filename extensions for certain generated files have changed. If your application code uses `#include` statements to include the Real-Time Workshop generated files (such as `model.prm`), you may need to modify these statements. See "Filename Extensions for Generated Files" on page 6-8.

Updating Release 11 Custom Targets

If you have created custom targets for the Real-Time Workshop under Release 11, you must update your custom system target files and template makefiles to create and utilize the build directory. See `matlabroot/rtw/c/grt` for examples.

To update a Release 11 target:

1 Add the following to your system target file.

```
/%
BEGIN_RTW_OPTIONS
...
rtwgensettings.BuildDirSuffix = '_grt_rtw';
END_RTW_OPTIONS
%/
```

2 Add `".."` to the `INCLUDES` rule in your template makefile. The following example is from `grt_lcc.tmf`.

```
INCLUDES = -I. -I.. $(MATLAB_INCLUDES) $(USER_INCLUDES)
```

The first `-I.` gets files from the build directory, and the second `-I..` gets files (e.g., user written S-functions) from the current working directory.

Conceptually, think of the current directory and the build directory as the same (as it was in Release 11). The current working directory contains items like user written S-functions. The reason `".."` must be added to the `INCLUDES` rule is that `make` is invoked in the build directory (i.e., the current directory was temporarily moved).

3 Place the generated executable in your current working directory. The following example is from `grt_lcc.tmf`.

```
PROGRAM = ../$(MODEL).exe
$(PROGRAM) : $(OBJS) $(RTWLIB)
   $(LD) $(LDFLAGS) -o $@ $(LINK_OBJS) $(RTWLIB) $(LIBS)
```

hilite_system Replaces locate_system

If you use the `locate_system` command, in MATLAB programs for tracing the Real-Time Workshop system/block identification tags, you should use `hilite_system` instead. See “hilite_system and Code Tracing” on page 6-8.

TLC Compatibility Issues

In bringing Target Language Compiler files from Release 11 to Release 12, the following changes may affect your TLC code base:

- Nested evaluations are no longer supported. Expressions such as

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

  are no longer supported. You will have to convert these expressions into equivalent non-nested expressions.

- Aliases are no longer automatically created for Parameter blocks while reading in the Real-Time Workshop files.

- You cannot change the contents of a "Default" record after it has been created. In the previous TLC, you could change a "Default" record and see the change in all the records that inherited from that default record.
- The `%codeblock` and `%endcodeblock` constructs are no longer supported.
- `%defines` and macro constructs are no longer supported.
- Use of line continuation characters (`...` and `\`) are not allowed inside of strings. Also, to place a double quote (`"`) character inside a string, you must use `\"`. Previously, the Target Language Compiler allowed you to use `"""` to get a double quote in a string.
- Semantics have been formalized to `%include` files in different contexts (e.g., from generate files, inside of `%with` blocks, etc.) `%include` statements are now treated as if they were read in from the global scope.
- The previous the Target Language Compiler had the ability to split function definitions (and other directives) across include file boundaries (e.g., you could start a `%function` in one file and `%include` a file that had the `%endfunction`). This no longer works.
- Nested functions are no longer allowed. For example,

```
%function foo ()
  %function bar ()
  %endfunction
%endfunction
```

- Built-in functions cannot be undefined via `%undef`. It is possible to undefine built in values, but this practice is not encouraged.
- Recursive records are no longer allowed. For example,

```
Record1    {
  Val      2
  Ref      Record2
}
Record2    {
  Val      3
  Ref      Record1
}
```

- 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>")
```