Real-Time Workshop Embedded Coder Release Notes

The “Real-Time Workshop Embedded Coder 4.0 Release Notes” on page 1-1 describe the changes introduced in the Release 14 version of the Real-Time Workshop Embedded Coder. The following topics are discussed in these Release Notes:

- “New Features” on page 1-2
- “Major Bug Fixes” on page 1-25
- “Upgrading from an Earlier Release” on page 1-26

The Real-Time Workshop Embedded Coder 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.

- “Real-Time Workshop Embedded Coder 3.2 Release Notes” on page 2-1
- “Real-Time Workshop Embedded Coder 3.1 Release Notes” on page 3-1

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 Embedded Coder 4.0 Release Notes

Real-Time Workshop Embedded Coder 3.2 Release Notes

2

Real-Time Workshop Embedded Coder 3.1 Release Notes

3

Real-Time Workshop Embedded Coder 4.0 Release Notes

New Features

This section summarizes the new features and enhancements introduced in the Real-Time Workshop Embedded Coder 4.0.

The new features include:

- “Expanded Documentation Collection” on page 1-2
- “New ERT Target Options User Interface” on page 1-3
- “New and Revised ERT Code Generation Options” on page 1-4
- “GRT and ERT Target Unification” on page 1-14
- “Support for Continuous Time Blocks and Solvers” on page 1-15
- “Noninlined S-Functions Supported” on page 1-16
- “Module Packaging Features” on page 1-16
- “ASAP2 File Generation Changes” on page 1-18
- “Code Generation With User-Defined Data Types” on page 1-18
- “Enhanced Custom Storage Classes” on page 1-18
- “More Efficient Multi-Rate Multitasking Code Generation” on page 1-19
- “More Efficient Task Scheduling for RTOS Targets” on page 1-20
- “New Callbacks Defined for System Target Files” on page 1-22
- “New Option to Control Template Makefile Output Display” on page 1-23
- “Demo Updates” on page 1-23

If you are upgrading from a version prior to Version 4.0, then also see “New Features” on page 2-2 in the Real-Time Workshop Embedded Coder 3.2 Release Notes.

Expanded Documentation Collection

The Real-Time Workshop Embedded Coder documentation collection has been expanded, and now comprises the following documents:

- Using Real-Time Workshop Embedded Coder: describes ERT model execution, timing, and task management; the `rtModel` data structure; how to interface to and call model code; ERT code generation options and optimization tips; custom storage classes; and advanced code generation techniques.

- The Module Packaging Features documentation describes the operation of an extensive set of new code customization features.
- The Developing Embedded Targets document describes requirements and implementation details for creation of custom embedded targets based on the ERT target. It includes detailed information on the structure and organization of target directories, system target files and template makefiles; how to support the RTW model referencing feature; device driver implementation; and on the operation of the build process and how to customize it.

New ERT Target Options User Interface

You can now configure Embedded Real-Time (ERT) target code generation options in the Simulink Model Explorer and the **Configuration Parameters** dialog box. Before you work the ERT target in this new environment, you should become familiar with:

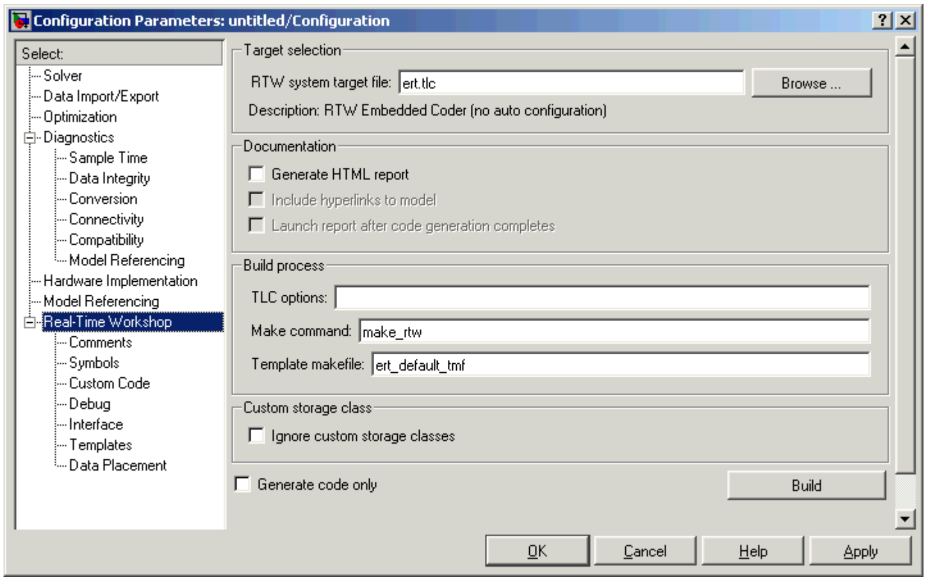
- Configuration sets and how to view and edit them in the Model Explorer and the **Configuration Parameters** dialog box. The Using Simulink documentation and the Simulink 6.0 Release Notes contain detailed information on these topics.
- The general Real-Time Workshop code generation options and the use of the System Target File Browser. The Real-Time Workshop documentation and the Real-Time Workshop 6.0 Release Notes contain detailed information on these topics.

Detailed descriptions of options specific to the ERT target are given in:

- The “Code Generation Options and Optimizations” chapter of the Real-Time Workshop Embedded Coder documentation.
- The Module Packaging Features documentation.

Illustrations throughout these release notes section will show the **Configuration Parameters** view of model parameters (see Figure 1-1) unless otherwise noted. Some panes of the **Configuration Parameters** dialog box (e.g., the **Templates** and **Interface** panes) contain only ERT-specific options. Others (such as the **Real-Time Workshop** pane, shown below) display a combination of general Real-Time Workshop options and ERT target options.

The next section, "New and Revised ERT Code Generation Options" on page 1-4 summarizes ERT target options that have been added or changed.


Configuration Parameters: untitled/Configuration

Target selection
RTW system target file: ert.tlc
Description: RTW Embedded Coder (no auto configuration)

Documentation
Generate HTML report
Include hyperlinks to model
Launch report after code generation completes

Build process
TLC options:
Make command: make_rtw
Template makefile: ert_default_tmf

Custom storage class
Ignore custom storage classes

Generate code only

Figure 1-1: Real-Time Workshop Pane of a Configuration Set, Viewed in Configuration Parameters Dialog Box

---

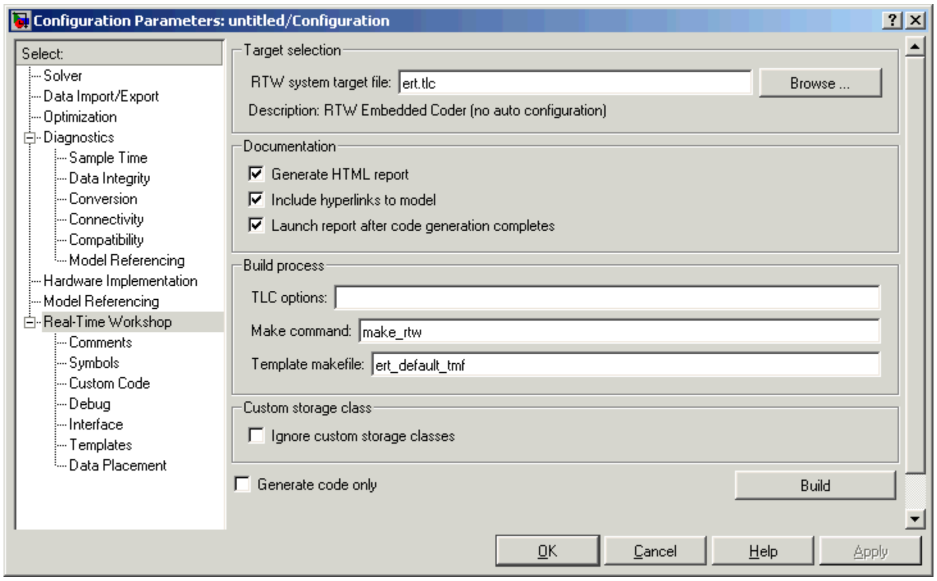
Note If you have developed a custom target based on the ERT target (or any other Real-Time Workshop target) see "Defining and Displaying Custom Target Options" on page 1-27 for a discussion of compatibility issues that may affect the appearance and operation of your target.

New and Revised ERT Code Generation Options

The ERT target supports several new code generation options. Other options are labeled differently, and some options default to different values. This note summarizes new and revised options and shows the panes containing each option discussed.

Detailed descriptions of all options are given in the “Code Generation Options and Optimizations” chapter of the Real-Time Workshop Embedded Coder documentation.

Real-Time Workshop Pane


Configuration Parameters: untitled/Configuration

Target selection
RTW system target file: ert.tlc
Description: RTW Embedded Coder (no auto configuration)

Documentation
[✓] Generate HTML report
[✓] Include hyperlinks to model
[✓] Launch report after code generation completes

Build process
TLC options:
Make command: make_rtw
Template makefile: ert_default_tmf

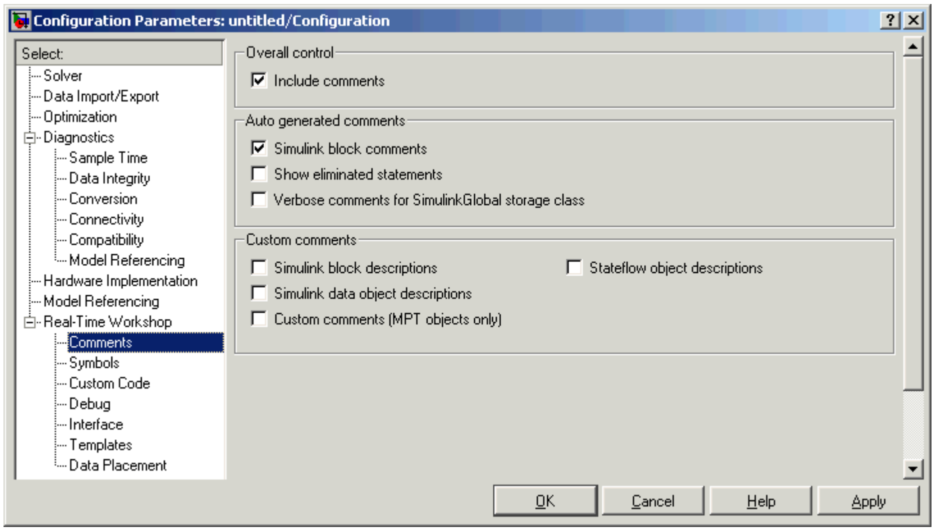
Custom storage class
[ ] Ignore custom storage classes

[ ] Generate code only

When **Generate HTML Report** is selected as shown above, two new options are enabled:

- **Include hyperlinks to model**: When you select this option, the HTML report includes hyperlinks from the code to the generating blocks in the model. By deselecting this option, you can speed up code generation, as generation of hyperlinks can be time-consuming for very large models.
- **Launch report after code generation completes**: When you select this option, the HTML report is automatically displayed in a MATLAB web browser window after code generation.

Comments Pane



Overall control

- [✓] Include comments

Auto generated comments

- [✓] Simulink block comments
- [ ] Show eliminated statements
- [ ] Verbose comments for SimulinkGlobal storage class

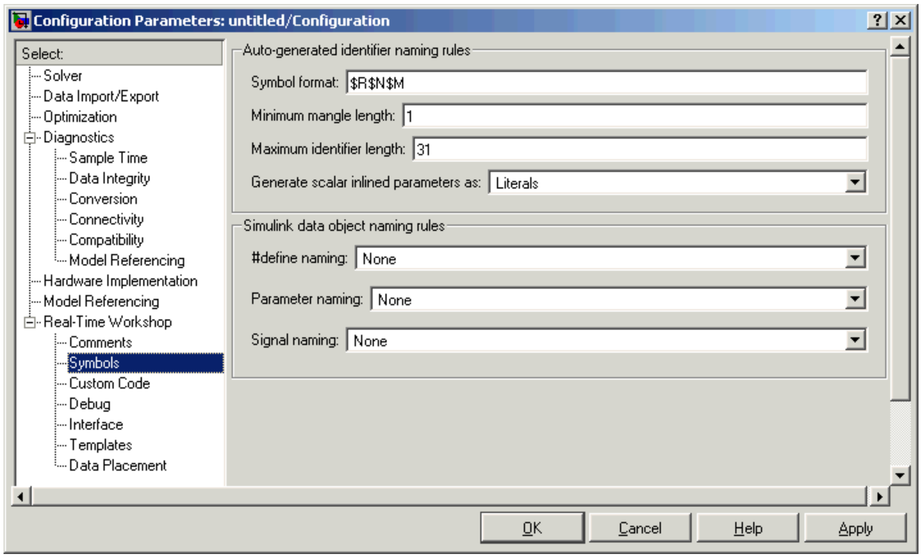
Custom comments

- [ ] Simulink block descriptions
- [ ] Stateflow object descriptions
- [ ] Simulink data object descriptions
- [ ] Custom comments (MPT objects only)

The **Custom Comments** subpane supports options that are specific to the ERT target:

- **Simulink block / Stateflow object / Simulink data object descriptions**: These options let you enable or suppress generation of descriptive information in comments for blocks, Stateflow charts, and data objects (such as signals and parameters).
- **Custom comments**: See the Module Packaging Features document for a description of this option.

Symbols Pane



Auto-generated identifier naming rules

Symbol format: $R$N$M

Minimum mangle length: 1

Maximum identifier length: 31

Generate scalar inlined parameters as: Literals

Simulink data object naming rules

#define naming: None

Parameter naming: None

Signal naming: None

The **Symbols** pane contains options that control the generation of symbols (such as variable names) in generated code. Most of these options are specific to the ERT target. New options are:

- **Simulink Data Object Naming Rules**: See the Module Packaging Features document for a description of the options in this subpane.
- **Symbol format**: The **Symbol format** field lets you enter a macro string that specifies whether, and in what order, certain substrings are included within generated symbols for signals, parameters, and other objects in the model. This lets you customize generated symbols. For example, you can specify that a model name or block name be inserted into each symbol.

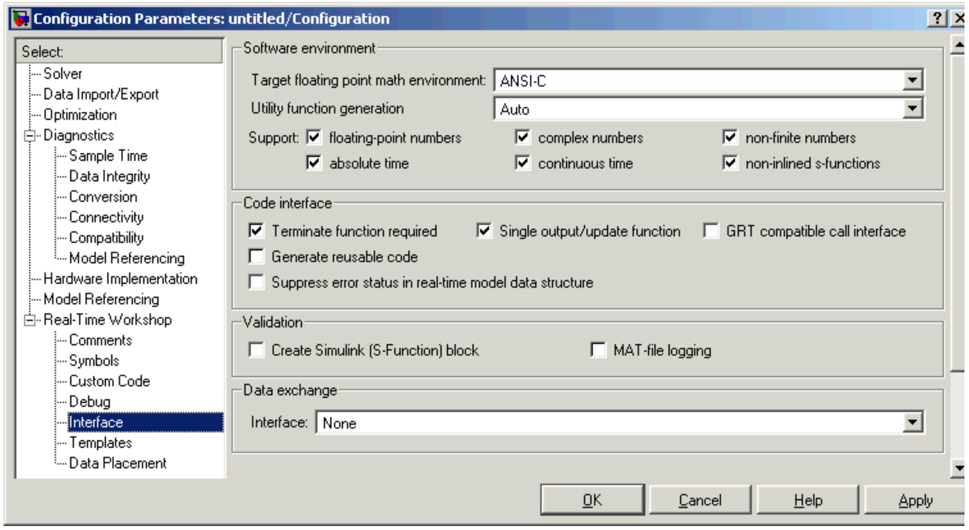
  The symbol generation process avoids name collisions by generating a *name mangling* string that is guaranteed to be unique for each generated identifier. The **Maximum identifier length** and **Minimum mangle length** options interact with the **Symbol format** specification to control the name mangling process.

The symbol generation process is described in detail in the “Symbols Pane” section of the “Code Generation Options and Optimizations” chapter of the Real-Time Workshop Embedded Coder documentation.

Note The **Symbol format** field supports (in a more compact form) all functions previously implemented by the **Prefix model name to global identifiers**, **Include System Hierarchy Number in Identifiers**, and **Include data type acronym in identifier** options. The **Symbol format** field replaces all these options. Existing models will continue to generate code that respects the previous settings of these options.

- **Generate scalar inlined parameters as**: For scalar inlined parameters, this pull-down menu lets you control whether parameter values are expressed: either as literals, or as symbols defined by `#define` directives. Note that this option is now licensed to Real-Time Workshop Embedded Coder.

Interface Pane


Configuration Parameters: untitled/Configuration

Software environment
- Target floating point math environment: ANSI-C
- Utility function generation: Auto
- Support: [✓] floating-point numbers [✓] complex numbers [✓] non-finite numbers [✓] absolute time [✓] continuous time [✓] non-inlined s-functions

Code interface
- [✓] Terminate function required
- [✓] Single output/update function
- [ ] GRT compatible call interface
- [ ] Generate reusable code
- [ ] Suppress error status in real-time model data structure

Validation
- [ ] Create Simulink (S-Function) block
- [ ] MAT-file logging

Data exchange
- Interface: None

- **Target floating point math environment**: In addition to the previously supported math libraries (`ANSI_C` and `ISO_C`), this pop-up menu now lets you select `GNU` to generate calls to the GNU C math library.
- **Utility function generation**: See the Real-Time Workshop 6.0 Release Notes for information.
- **Support floating point / complex / non-finite numbers**: These three options let you enable or suppress the generation of floating point, complex, or nonfinite numbers. By default, all three options are selected.

  Note that the **Support floating point numbers** option replaces, and inverts the logic of, the **Integer code only** option that was supported in previous releases. To generate pure integer code, deselect the **Support floating point numbers** option.
- **Support continuous time**: By default, this option is selected, and the ERT target supports code generation for continuous time blocks. If this option is deselected, the build process generates an error if any continuous time blocks are present in the model.
- **Support non-inlined S-functions**: By default, this option is selected, and the ERT target supports code generation for non-inlined S-functions. If this option is selected, the build process generates an error if any non-inlined S-functions are present in the model.
- **Support absolute time**: 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). These related options determine how the ERT target provides absolute or elapsed time values to blocks in the model.

  By default, **Support absolute time** is selected. In this case, the ERT target generates integer counters that provide absolute or elapsed time values to blocks in the model. When **Support absolute time** is deselected, an error is raised at code generation time if any blocks requiring absolute or elapsed time values are present in the model.

  For further information on the allocation and operation of absolute and elapsed timers, see the “Timer Services” chapter of the Real-Time Workshop documentation.
- **Pass root-level I/O as**: This option is enabled only if the **Generate reusable code** option is selected. The **Pass root-level I/O as**: menu provides options that control how model inputs and outputs at the root level of the model are passed in to the `model_step` function. The options are

- `Individual Arguments`: This option is the default. Each root-level model input and output is passed to `model_step` as a separate argument.
- `Structure Reference`: When this option is selected, all root-level inputs are packed into a struct that is passed to `model_step` as an argument. Likewise, all root-level outputs are packed into a struct that is also passed to `model_step` as an argument.

- **GRT compatible call interface**: 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 also the note "GRT and ERT Target Unification" in the Real-Time Workshop 6.0 Release Notes.

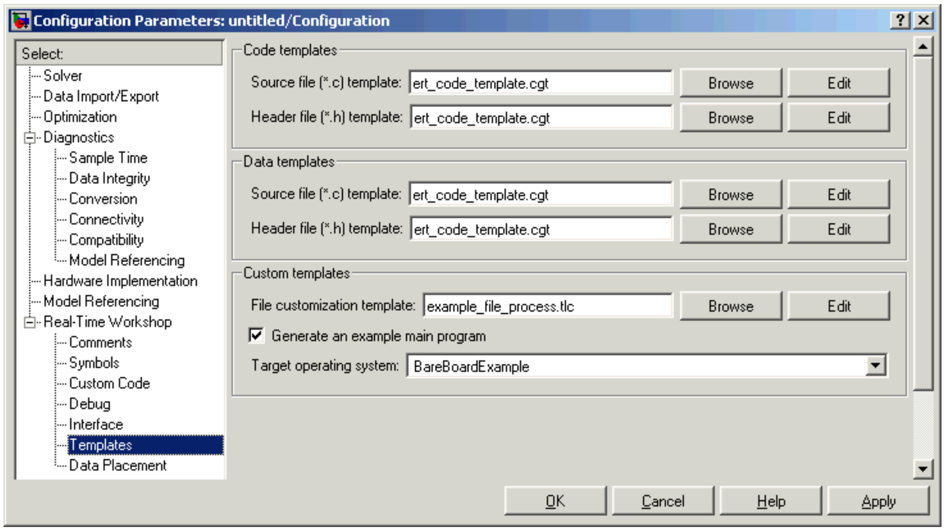
- **Data Exchange** options: The **Data Exchange** panel contains the **Interface** menu, which lets you select options related to interfacing model data to systems external to the generated code. These options include generation of external mode support code, generation of ASAP2 data files, and generation of C API code for monitoring signals and/or parameters.

  All **Data Exchange** options are described in the Real-Time Workshop documentation.

  For information on deplyment of external mode on an embedded target, see also "Using External Mode With the ERT Target" in the Real-Time Workshop Embedded Coder documentation.

Templates Pane

This pane contains advanced options that enable you to customize generated code.


Configuration Parameters: untitled/Configuration

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Integrity
  - Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
- Hardware Implementation
- Model Referencing
- Real-Time Workshop
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Templates
  - Data Placement

Code templates

Source file (*.c) template: ert_code_template.cgt

Header file (*.h) template: ert_code_template.cgt

Data templates

Source file (*.c) template: ert_code_template.cgt

Header file (*.h) template: ert_code_template.cgt

Custom templates

File customization template: example_file_process.tlc

☑ Generate an example main program

Target operating system: BareBoardExample

Code Templates and Data Templates Subpanes

Code and data templates provide extensive code customization features, which are described in the separate Module Packaging Features document.

The “Advanced Code Generation Techniques” chapter of the Real-Time Workshop Embedded Coder documentation contains a simple example of how a code template can be applied to generate customized comment sections in generated code files.

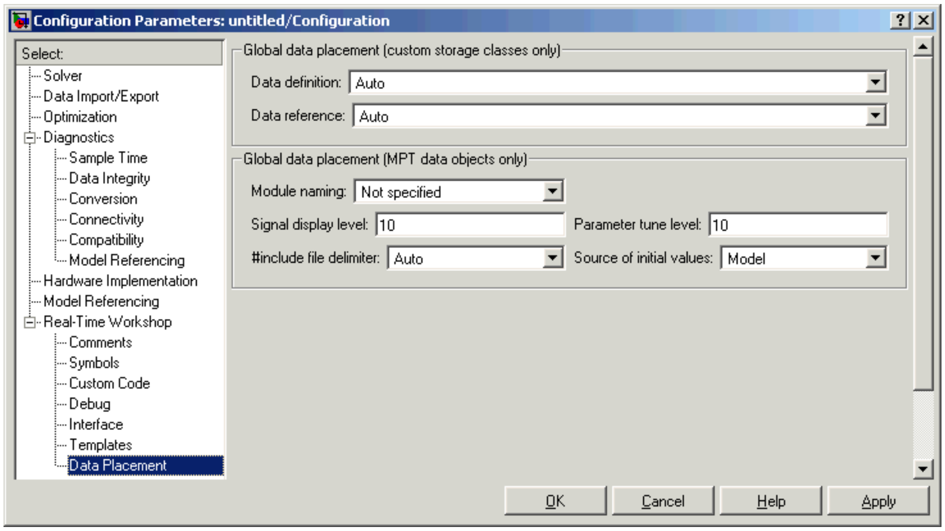
Custom Templates Subpane

- **File customization template**: This field lets you specify a *custom file processing template* (CFP) template file. CFP templates let you customize generated code by organizing generated code into sections (such as includes, typedefs, functions, and more). A CFP template can emit code, directives, or comments into each section as required. See the “Advanced Code Generation Techniques” chapter of the Real-Time Workshop Embedded Coder documentation for detailed information.
- **Generate an example main program**: This option and the related **Target operating system** pop-up menu let you generate a model-specific example main program module. See the “Data Structures and Program Execution”

chapter of the Real-Time Workshop Embedded Coder documentation for more information.

Data Placement Pane

This pane contains advanced options that are described in the separate Module Packaging Features document.



Global data placement (custom storage classes only)

Data definition: Auto

Data reference: Auto

Global data placement (MPT data objects only)

Module naming: Not specified

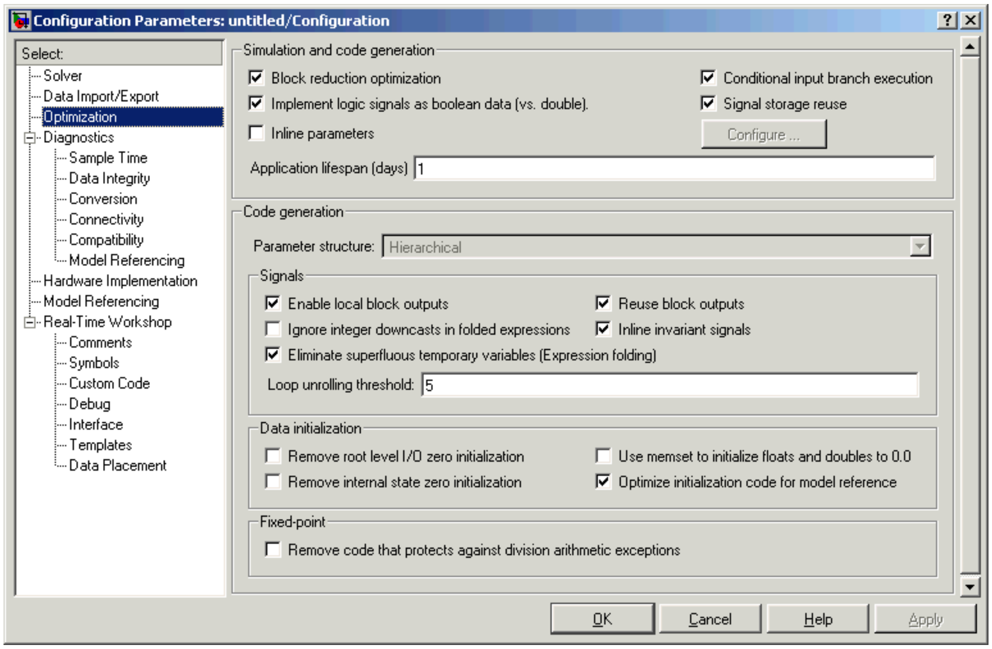
Signal display level: 10

Parameter tune level: 10

#include file delimiter: Auto

Source of initial values: Model

Optimization Pane



Simulation and code generation

- [✓] Block reduction optimization
- [✓] Conditional input branch execution
- [✓] Implement logic signals as boolean data (vs. double).
- [✓] Signal storage reuse
- [ ] Inline parameters

Application lifespan (days) 1

Code generation

Parameter structure: Hierarchical

Signals

- [✓] Enable local block outputs
- [✓] Reuse block outputs
- [ ] Ignore integer downcasts in folded expressions
- [✓] Inline invariant signals
- [✓] Eliminate superfluous temporary variables (Expression folding)

Loop unrolling threshold: 5

Data initialization

- [ ] Remove root level I/O zero initialization
- [ ] Use memset to initialize floats and doubles to 0.0
- [ ] Remove internal state zero initialization
- [✓] Optimize initialization code for model reference

Fixed-point

- [ ] Remove code that protects against division arithmetic exceptions

The **Data initialization** and **Fixed-point** subpanes contain options that are specific to the ERT target. New and revised options are:

- **Remove root-level I/O zero initialization**: This option was previously labeled **Initialize external data**. When this option is off (the default), initialization code for root-level inports and outports whose value is zero is generated. Otherwise, initialization code for such inports and outports is not generated.

  Note that the default for this option is now off rather than on.

- **Remove internal state zero initialization**: This option was previously labeled **Initialize internal data**. When this option is off (the default), initialization code for internal work structures (e.g., block states and block outputs) is generated. Otherwise, the initialization code is not generated.

  Note that the default for this option is now off rather than on.

- **Use memset to initialize floats and doubles**: This option was previously labeled **Initialize Floats and Doubles to 0.0**.

  When **Use memset to initialize floats and doubles** is off (the default), all internal storage (regardless of type) is cleared to the integer bit pattern `0` (that is, all bits are off). When this option is on, additional code is generated to set float and double storage explicitly to the value `0.0`, using the `memset` function. This additional code is slightly less efficient.

  Note that the default for this option is now off rather than on.
- **Optimize initialization code for model reference**: You should deselect this option if your model meets both the following conditions:
  - The model contains an enabled subsystem
  - The model will be referenced from another model via a Model block.

  Otherwise, you should leave the option selected (the default).
- **Remove code that protects against division arithmetic exceptions**: Select this option to suppress generation of code that guards against fixed-point division by zero exceptions. By default, this option is deselected.

Note The **Application lifespan (days)** parameter in the **Simulation and code generation** subpane lets you minimize the allocation of memory for absolute and elapsed time counters that are generated for blocks that require the value of either absolute or elapsed time. The word size of the counters (8,16, 32, or 64 bits), is allocated optimally, to accommodate the maximum value specified in **Application lifespan (days)** field. For further information on the allocation and operation of absolute and elapsed timers, see the “Timer Services” chapter of the Real-Time Workshop documentation.

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 now use a common backend generated code format.
- The set of features common to both products is expanded.

- Some features and efficiencies formerly exclusive to Real-Time Workshop Embedded Coder 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.
- Conversion from GRT-based targets to ERT-based targets is greatly simplified.

See the Real-Time Workshop 6.0 Release Notes for 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.

Support for Continuous Time Blocks and Solvers

Continuous Block Support. The ERT target now supports code generation for continuous time blocks. If the **Support continuous time** option is selected (the default) you can use any such blocks in your models, without restriction.

Note that use of certain blocks is not recommended for production code generation for embedded systems. The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Fixed-Point block libraries, including whether or not they are recommended for use in production code generation. The view this table, execute the following command at the MATLAB command line:

```
showblockdatatypetable
```

Then, refer to the “Recommended for Production Code?” column of the table.

Continuous Solver Support. The ERT target also now supports continuous solvers. In the **Solver** options dialog, you can select any available solver in the **Solver** menu. (Note that the solver **Type** must be `fixed-step` for use with the ERT target, as in previous releases.)

Note Custom targets must be modified to support continuous time. The required modifications are described in “Supporting Continuous Time in Custom Targets” on page 1-29.

Noninlined S-Functions Supported

In previous releases, the ERT target required that all S-functions in a model be inlined with a corresponding TLC file for code generation. This restriction has been removed.

When the **Support noninlined S-functions** option is selected (the default) the ERT target now supports use of noninlined S-functions.

Note, however, that inlining S-functions is often advantageous in production code generation, for example in implementing device drivers. See “Tradeoffs in Device Driver Development” in the Developing Embedded Targets document for a discussion of the pros and cons.

Module Packaging Features

The Module Packaging Features (MPF) are a major subcomponent of the Real-Time Workshop Embedded Coder. The Module Packaging Features documentation describes these features in detail. This note summarizes the capabilities of MPF.

Introduction

With MPF, you can

- Package the generated code into the desired number of `.c` and `.h` files.
- Control the *internal* organization of each of the generated files. For example, for readability, your company may have software standards that define where to place comments and sections of code within files.
- Control whether or not the generated files contain definitions for a model’s global identifiers. If such definitions exist, you determine the files in which the code generator places them. Also, you can specify the generated files where the code generator places global data (`extern`) declarations.

In addition to meeting such packaging needs, MPF allows you to implement these and other features to meet your needs:

- Register user-defined data types.
- Customize comments.
- Locate variables in target memory where desired.

By providing dialog boxes, user-definable templates and the ability to use M-scripts, MPF allows you to implement these and other features discussed in this document to meet your special needs..

MPF Feature Summary

This section summarizes the module packaging features introduced in Real-Time Workshop Embedded Coder Version 4.0. MPF allows you to

- Select or define MPF template files. You can generate the desired `.c` and `.h` files and organize them the way you want. Also, these templates include template symbols whose locations in a template file determine where comments and code will be located *in* the individual generated files.
- Manage the code generation data dictionary. This allows
  - Registering user-defined data types
  - Importing data objects into the code generation data dictionary from a `.mat` file of a previous Simulink session or from an external data dictionary (such as an Excel file)
  - Adding Simulink data objects using the **Data Object Wizard**
  - Changing the alphabetical case and spellings that identifier names will have in the generated code
- Select additional miscellaneous and advanced options. These include
  - Instructing the code generator to use the angle-bracket delimiter (for multiple data objects), instead of the double-quotation delimiter.
  - Selecting the source that initializes each of the model's signals in the generated code.
  - Adding a selected data object's property values as a comment in a generated file above that data object's identifier.
  - Adding a comment to the model using the Simulink DocBlock so that this comment appears in the generated file where desired.
- Manage file placement of data declarations. You can determine whether or not the generated files contain defining declarations for a model's global identifiers. If defining declarations exist, you can determine the files in which the code generator places them. Also, you can determine the files where the code generator places global data reference (`extern`) declarations.

ASAP2 File Generation Changes

ASAP2 file generation is now generally available to all Real-Time Workshop targets. The documentation for this feature has been relocated to “Generating ASAP2 Files” in the Real-Time Workshop documentation.

Code Generation With User-Defined Data Types

Real-Time Workshop Embedded Coder now supports use of user-defined data type objects in code generation. These include objects of the following classes:

- `Simulink.NumericType`
- `Simulink.StructType`
- `Simulink.Bus`
- `Simulink.Aliastype`

In code generation, you can use user-defined data type objects to map your own data type definitions to Simulink built-in data types, and to generate `#include` directives specifying your own header files, containing your data type definitions.

See the “Advanced Code Generation Techniques” chapter of the Real-Time Workshop Embedded Coder documentation for details.

Enhanced Custom Storage Classes

The Real-Time Workshop Embedded Coder has extended the built-in storage classes provided by Real-Time Workshop. The Real-Time Workshop Embedded Coder now includes:

- A set of *custom storage classes* (CSCs). CSCs are designed to be useful in code generation for embedded systems development. The new enhanced and expanded CSC functionality has been incorporated into the `Simulink.Signal` and `Simulink.Parameter` classes. This simplifies code generation with CSCs, since you can use familiar signal and parameter objects for this purpose.
- The new Custom Storage Class Designer (`cscdesigner`) tool. The Custom Storage Class Designer lets you define additional CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that lets you implement CSCs quickly and easily. You can use your CSCs in code generation immediately, without any TLC or other programming.

CSCs give you extended control over the constructs required to represent data in an embedded algorithm. For example, you can use CSCs to

- Define structures for storage of parameter or signal data.
- Conserve memory by storing Boolean data in bit fields.
- Integrate generated code with legacy software whose interfaces cannot be modified.
- Generate data structures and definitions that comply with your organization's software engineering guidelines for safety-critical code.

See the "Custom Storage Classes" chapter of the Real-Time Workshop Embedded Coder User's Guide for a complete description of CSCs and the Custom Storage Class Designer.

Compatibility with Previous CSCs

In prior releases, CSCs were implemented via special `Simulink.CustomSignal` and `Simulink.CustomParameter` classes. We recommend that you consider replacing `Simulink.CustomSignal` and `Simulink.CustomParameter` objects in your models with equivalent `Simulink.Signal` and `Simulink.Parameter` objects.

Minor changes have been made in the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes. See "Custom Storage Class Compatibility Issues" on page 1-26 for information on these changes.

More Efficient Multi-Rate Multitasking Code Generation

Real-Time Workshop Embedded Coder now generates significantly faster code for multi-rate multitasking models.

For multi-rate multitasking models, Real-Time Workshop Embedded Coder uses a strategy called *rate grouping*. Rate grouping generates separate *model*`_step` functions for the base rate task and each sub-rate task in the model. The function naming convention for these functions is

```
model_stepN
```

where *N* is a task identifier. For example, for a model named `my_model` that has three rates, the following functions are generated:

```
void my_model_step0 (void);
void my_model_step1 (void);
void my_model_step2 (void);
```

Each `model_stepN` function executes all blocks sharing `tid N`; in other words, all block code that executes within task `N` is grouped into the associated `model_stepN` function.

For other cases, Real-Time Workshop Embedded Coder generates a single `model_step` function. This `model_step` function uses the same scheduling technique (called *rate guarding*) as in previous versions of the product. when rate guarding is used, a task identifier is passed in to the `model_step` function.

To take advantage of rate grouping for existing multi-rate multitasking models models, you must regenerate code, including the main program, `ert_main.c`.

See the “Data Structures and Program Execution” chapter of the Real-Time Workshop Embedded Coder documentation for a complete discussion of rate grouping.

More Efficient Task Scheduling for RTOS Targets

This note will be of interest primarily to developers of real-time operating system (RTOS) targets based on the ERT target. Using the new `rtmStepTask` macro described in this note, targets that employ the task management mechanisms of an RTOS can eliminate certain redundant scheduling calls during the execution of tasks in a multi-rate, multitasking model, thereby improving performance of the generated code.

In order to understand the optimization that is available for an RTOS target, first consider how the ERT target schedules tasks for bare-board targets (where no RTOS is present). The ERT target maintains *scheduling counters* and *event flags* for each sub-rate task. The scheduling counters are implemented within the real-time model (`rtM`) data structure. The event flags are implemented as arrays, indexed on task identifier (`tid`).

The scheduling counters are updated by the base-rate task. The counters are, in effect, clock rate dividers that count up the sample period associated with each sub-rate task. When a given sub-rate counter reaches a value that indicates it has a hit, the sample period for that rate has elapsed and the counter is reset to zero. When this occurs, the sub-rate task must be scheduled for execution.

The event flags indicate whether or not a given task is scheduled for execution. For a multi-rate, multitasking model, the event flags are maintained by the `model_SetEventsForThisBaseStep` function. `model_SetEventsForThisBaseStep` invokes the macro `rtmStepTask` to test the value of each counter. `rtmStepTask` returns `TRUE` when a counter indicates that a task's sample period has elapsed. When this occurs, `model_SetEventsForThisBaseStep` sets the event flag for that task.

On each time step, the counters and event flags are updated and the base-rate task executes. Then, the scheduling flags are checked in `tid` order, and any task whose event flag is set is executed. This ensures that tasks are executed in order of priority.

For bare-board targets that cannot rely on an external RTOS, the event flags are mandatory in order to allow overlapping task preemption. However, an RTOS target uses the operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant. An RTOS target can eliminate the call to `model_SetEventsForThisBaseStep`, and examine the counters by invoking `rtmStepTask` directly, as described in the next sections.

Using rtmStepTask

The `rtmStepTask` macro is defined in `model.h`. The syntax of `rtmStepTask` is:

```
boolean task_ready = rtmStepTask(rtm, idx);
```

The arguments are:

- `rtm`: pointer to the real-time model structure (`rtM`)
- `idx`: task identifier (`tid`) of the task whose scheduling counter is to be tested.

`rtmStepTask` returns `TRUE` if the task's scheduling counter equals zero, indicating that the task should be scheduled for execution on the current time step. Otherwise, it returns `FALSE`.

If your target supports the **Generate an example main program** option, you can generate calls to `rtmStepTask` using the TLC function `RTMTaskRunsThisBaseStep`. The following example, from `ertmainlib.tlc`, is designed for the VxWorks RTOS. A loop iterates over each subrate task. `rtmStepTask` is called for each task. If `rtmStepTask` returns `TRUE`, the VxWorks `semGive()` function is called, and VxWorks schedules the task to run.

```
%assign ifarg = RTMTaskRunsThisBaseStep("i")
for (i = 1; i < %<FcnNumST()>; i++) {
   if (%<ifarg>) {
     semGive(taskSemList[i]);
     if (semTake(taskSemList[i],NO_WAIT) != ERROR) {
       logMsg("Rate for SubRate task %d is too
fast.\n",i,0,0,0,0,0);
       semGive(taskSemList[i]);
     }
   }
}
```

Suppressing model_SetEventsForThisBaseStep

By default, the `model_SetEventsForThisBaseStep` is still generated for backward compatibility. The TLC variable `SuppressSetEventsForThisBaseStepFcn` is provided to control generation of this function. To suppress generation of this function, add the following statement to your system target file (before the `%include "codegenentry.tlc"` statement):

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

New Callbacks Defined for System Target Files

The release 14 API for system target file callbacks provides three new callback functions for use in system target files. Unlike `rtwoptions` callbacks, these functions are associated with the target, not with its individual options. The callbacks are installed as fields in the `rtwgensettings` structure of the system target file. The callbacks, summarized below, are fully described in the “System Target Files” chapter of the Developing Embedded Targets document.

In summary, the callbacks are:

`rtwgensettings.SelectCallback`: The `SelectCallback` function is triggered during model loading, and also when the user selects a target via the System Target File browser.

Note 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 the “Supporting Model Referencing” chapter of the Developing Embedded Targets document.

`rtwgensettings.ActivateCallback`: The `ActivateCallback` function is triggered when the active configuration set of the model changes. This could happen during model loading, and also when the user changes the active configuration set

`rtwgensettings.postapplyCallback`: The `PostApplyCallback` function is triggered when the user clicks the **Apply** or **OK** buttons after editing options in the **Configuration Parameters** dialog or the Model Explorer. The `PostApplyCallback` function is called after the changes have been applied to the configuration set.

New Option to Control Template Makefile Output Display

A new template makefile option lets you control whether or not template makefile output is displayed during the build process. To enable makefile output display at all times (regardless of the setting of the **Verbose build** option in the Real-Time Workshop **Debugging** pane) add the following macro to your template makefile:

```
VERBOSE_BUILD_OFF_TREATMENT = PRINT_OUTPUT_ALWAYS
```

When your template makefile is configured in this way, the **Verbose build** option control display of other build process output (such as TLC messages), but template makefile output is always displayed.

This macro should be added in the template makefile section that includes other macros such as `BUILD_SUCCESS`, etc.

Demo Updates

This release includes a major update and reorganization of the Real-Time Workshop and Real-Time Workshop Embedded Coder demo collection. If you are reading this document online in the MATLAB Help browser, you can open the demo suite by clicking on this link: `rtwdemos`

Alternatively, you can access the demo suite by typing the name of the demo library at the MATLAB command prompt:

```
rtwdemos
```

Major Bug Fixes

The Real-Time Workshop Embedded Coder 4.0 includes several bug fixes made since Version 3.2. This section describes the particularly important Version 4.0 bug fixes.

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.

Upgrading from an Earlier Release

The issues involved in upgrading from the Real-Time Workshop Embedded Coder 3.2 to Version 4.0 are described below:

- "Custom Storage Class Compatibility Issues" on page 1-26
- "Defining and Displaying Custom Target Options" on page 1-27
- "Supporting Model Referencing in Custom Targets" on page 1-28
- "Supporting Continuous Time in Custom Targets" on page 1-29
- "rtwtypes.h Replaces tmwtypes.h" on page 1-30
- "Updating Customized Static Main Program Modules" on page 1-30
- "Integer Code Only Option Replaced" on page 1-32
- "Rate Grouping Compatibility Issues" on page 1-32
- "Real-Time Object Structure Obsoleted by Real-Time Model Structure" on page 1-32
- "rtmIsSampleHit and rtmIsSpecialSampleHit Macros Obsolete" on page 1-33
- "RTWInfo Properties Assignment Warning Message" on page 1-33

This section describes upgrade issues involved in moving to the Real-Time Workshop Embedded Coder 4.0 from Version 3.2.

Custom Storage Class Compatibility Issues

In prior releases, custom storage classes were implemented via special `Simulink.CustomSignal` and `Simulink.CustomParameter` classes. (See the "Pre-Release 14 Custom Storage Classes" of the "Custom Storage Classes" chapter in the Real-Time Workshop Embedded Coder documentation.)

In Release 14, the full functionality of the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes has been added to the `Simulink.Signal` and `Simulink.Parameter` classes. The new custom storage classes are described in "Enhanced Custom Storage Classes" on page 1-18. We recommend that you consider replacing `Simulink.CustomSignal` and `Simulink.CustomParameter` objects in your models with equivalent `Simulink.Signal` and `Simulink.Parameter` objects.

If you prefer, you can continue to use the `Simulink.CustomSignal` and `Simulink.CustomParameter` classes in the current release. Note that the following changes have been implemented in these classes:

- The `Internal` storage class has been removed from the enumerated values of the `RTWInfo.CustomStorageClass` property. `Internal` storage class is no longer supported.
- For the `ExportToFile` and `ImportFromFile` storage classes, the `RTWInfo.CustomAttributes.FileName` and `RTWInfo.CustomAttributes.IncludeDelimeter` properties have been combined into a single property, `RTWInfo.CustomAttributes.HeaderFile`. When specifying a header file, include both the filename and the required delimiter as you want them to appear in generated code, as in the following example:

  ```
  myobj.RTWInfo.CustomAttributes.HeaderFile = '<myheader.h>';
  ```

- Prior to Release 14, user-defined CSCs were created by designing custom packages that included the CSC definitions (as described in the `cscdesignintro` tutorial demo). This technique for creating CSCs is obsolete. For a description of the current procedure, which is much simpler, see “Creating Packages with CSC Definitions” in the “Custom Storage Classes” chapter in the Real-Time Workshop Embedded Coder documentation.

  If you designed your own custom packages containing CSCs prior to Release 14 we strongly recommend that you convert them to Release 14 CSCs. The conversion procedure is described in the next section, “Converting pre-Release 14 Packages to Use CSC Registration Files” in the “Custom Storage Classes” chapter in the Real-Time Workshop Embedded Coder documentation.

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 and **Configuration Parameters** dialogs to present target options to end users. If you do not want to do so, a mechanism for using the old-style **Simulation Parameters** dialog is available for backwards compatibility.

The “System Target Files” chapter of the Developing Embedded Targets document discusses compatibility issues and solutions related to the definition and display of target-specific options for custom targets.

- Callback compatibility: If the `rtwoptions` array in your custom system target file contains callbacks, you must convert your callbacks to use the callback compatibility API provided in this release. See “Compatibility Issues for rtwoptions Callbacks” in the “System Target Files” chapter of the Developing Embedded Targets document.
- Target options inheritance: if your custom target is derived from another target and inherits options, you will need change your system target file to use a new inheritance mechanism. See “Inheritance of Target Options” in the “System Target Files” chapter of the Developing Embedded Targets document.
- Display of target options: Your target options will be displayed differently, and you may want to reorganize them. See “Appearance of Target Options in New Model Explorer Dialog View” in the “System Target Files” chapter of the Developing Embedded Targets document for information on how custom target options are displayed.

Supporting Model Referencing in Custom Targets

Existing custom targets require a number of modifications for code generation compatibility with the model reference features introduced in Release 14. The “Supporting Model Referencing” chapter of the Developing Embedded Targets document provides the information you will need to adapt your target to support model referencing. Most of the guidelines concern required modifications to the system target file and template makefile.

The list below summarizes general requirements and issues for model reference compatibility that are discussed in the “Supporting Model Referencing” chapter:

- A model reference compatible target must be derived from the ERT or GRT targets.
- Your system target file must declare model reference compatibility.
- Your template makefile must define a number of makefile tokens, variables and rules specifically for model referencing support.

- To support model reference builds, your template makefile must support use of the shared utilities directory.
- 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.

For general information about model referencing, see the Real-Time Workshop documentation.

Supporting Continuous Time in Custom Targets

The ERT target now supports continuous time (see "Support for Continuous Time Blocks and Solvers" on page 1-15). If you want your custom ERT-based target to take advantage of this new feature, you must update your template makefile (TMF) and the static main program module (e.g., `mytarget_main.c`) for your target.

Template Makefile Modifications

Add the `NCSTATES` token expansion after the `NUMST` token expansion, as follows:

```
NUMST = |>NUMST<|
NCSTATES = |>NCSTATES<|
```

In addition, add `NCSTATES` to the `CPP_REQ_DEFINES` macro, as in the following example:

```
CPP_REQ_DEFINES = -DMODEL=$(MODEL) -DNUMST=$(NUMST) -DNCSTATES=$(NCSTATES) \
-DMAT_FILE=$(MAT_FILE)
-DINTEGER_CODE=$(INTEGER_CODE) \
-DONESTEPFCN=$(ONESTEPFCN) -DTERMFCN=$(TERMFCN) \
-DHAVESTDIO
-DMULTI_INSTANCE_CODE=$(MULTI_INSTANCE_CODE) \
-DADD_MDL_NAME_TO_GLOBALS=$(ADD_MDL_NAME_TO_GLOBALS)
```

Modifications to Main Program Module

The main program module defines a static main function that manages task scheduling for all supported tasking modes of single- and multiple-rate models. `NUMST` (the number of sample times in the model) determines whether the main function calls multirate or singlerate code.

However, when the model model has continuous time, it is incorrect to rely on NUMST directly.

When the model has continuous time and the flag TID01EQ is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and TID01EQ is true, the generated code has a single-rate call interface.

To support models that have continuous time, update the static main module to take TID01EQ into account, as follows:

1 Before NUMST is referenced in the file, add the following code:

```
#if defined(TID01EQ) && TID01EQ == 1 && NCSTATES == 0
#define DISC_NUMST (NUMST - 1)
#else
#define DISC_NUMST NUMST
#endif
```

2 Replace all instances of NUMST in the file by DISC_NUMST.

rtwtypes.h Replaces tmwtypes.h

The ERT target now generates an optimized rtwtypes.h header file which includes only the necessary definitions required by the target. Most generated code modules require these definitions. This header file replaces the static tmwtypes.h header file. Note that non-ERT targets still use the tmwtypes.h header file.

Updating Customized Static Main Program Modules

If you are upgrading and your application uses a customized version of the static main program module ert_main.c, open the module and make the following changes:

1 Search for RT_MDL. This search brings you to the “Required defines” section.

2 Replace

```
#define RT_MDL        CONCAT(MODEL,_rtO)
```

with

```
#define RT_MDL        CONCAT(MODEL,_M)
```

3 Search for `tmwtypes.h`. This search brings you to the "Includes" section.

4 Add the following include statement.

```
#include "rtwtypes.h"
```

5 Delete the following include statements.

```
#include "tmwtypes.h"
#include "simstruc_types.h"
```

6 Just below the "Includes" section, add the following preprocessor conditional code, which determines whether to set up multitasking mode. Previously, this code resided in `simstruct_types.h`.

```
/*========================*
 * Setup for multitasking *
 *========================*/
#if defined(MT)
# if MT == 0
#   undef MT
# else
#   define MULTITASKING 1
# endif
#endif
```

For more information about `ert_main.c`, see The Static Main Program Module.

The MathWorks recommends that you generate a target-specific main program module rather than use a customized version of the static module, `ert_main.c`. For details, see Generating the Main Program and Custom File Processing Generation in the Real-Time Workshop Embedded Coder documentation.

Integer Code Only Option Replaced

The **Support floating point numbers** option (see “New and Revised ERT Code Generation Options” on page 1-4) replaces, and inverts the logic of, the **Integer code only** option that was supported in previous releases. To generate pure integer code in new models, deselect the **Support floating point numbers** option.

Note that for compatibility, models that were configured for **Integer code only** prior to release 14 will automatically be configured with **Support floating point numbers** deselected, and will therefore continue to generate pure integer code.

Rate Grouping Compatibility Issues

To take full advantage of the efficiency of rate grouping (see “More Efficient Multi-Rate Multitasking Code Generation” on page 1-19):

- Your multi-rate inlined S-functions must be upgraded to be fully rate grouping compliant. Existing S-functions will continue to operate correctly without change, but we strongly recommend that you upgrade your TLC S-function implementations. See “Rate Grouping Compliance and Compatibility Issues” in the “Data Structures and Program Execution” chapter of the Real-Time Workshop Embedded Coder documentation.
- If you have previously generated and modified `ert_main.c` (as is typical of many ERT-based custom targets) take care to preserve your modifications and make equivalent changes to the regenerated `ert_main.c`. After you have done so, set the TLC variable `RateBasedStepFcn` to `1`, as described in “Rate Grouping and the Static Main Program” in the “Data Structures and Program Execution” chapter of the Real-Time Workshop Embedded Coder documentation.

Real-Time Object Structure Obsoleted by Real-Time Model Structure

In MATLAB release 13, the real-time model (*`model`*`_M`) data structure replaced the real-time object (*`model`*`_rtO`) data structure. However, use of use of the older structure was still supported for backward compatibility.

Real-Time Workshop Embedded Coder 4.0 requires use of the real-time model data structure. If you have developed a custom target that references `model_rtO` (for example, in a customized `ert_main.c` module) you must replace them with references to `model_M`.

See the “Data Structures and Program Execution” chapter of the Real-Time Workshop Embedded Coder documentation for further information about the real-time model data structure.

rtmIsSampleHit and rtmIsSpecialSampleHit Macros Obsolete

The following macros are now obsolete and should not be used with the ERT target:

- `rtmIsSampleHit`
- `rtmIsSpecialSampleHit`

This will not cause a problem unless you have coded these macros directly into your TLC files. The recommended practice is to use the following TLC library functions:

- `%<LibIsSFcnSampleHit(tid)>`
- `%<LibIsSFcnSpecialSampleHit(tid)>`

If you have used these functions, they will operate transparently. See also “More Efficient Multi-Rate Multitasking Code Generation” on page 1-19.

RTWInfo Properties Assignment Warning Message

This note describes a minor change in behavior when the `RTWInfo` properties of a data object are assigned incorrectly.

You can assign a custom storage class to a data object either by using the Simulink Model Explorer, or by setting the `RTWInfo` properties via MATLAB commands. (See also the “Custom Storage Classes” chapter in the Real-Time Workshop Embedded Coder documentation). If you use MATLAB commands to assign a custom storage class, you must set both the `RTWInfo.CustomStorageClass` and `RTWInfo.StorageClass` fields. Make sure that the `RTWInfo.StorageClass` property is set to `'Custom'`, as in the following example.

```
aa = Simulink.Signal;
aa.RTWInfo.StorageClass = 'Custom';
aa.RTWInfo.CustomStorageClass = 'Struct';
aa.RTWInfo.CustomAttributes.StructName = 'mySignals';
```

If the `RTWInfo.StorageClass` is not set correctly as shown above, the assigned custom storage class (`RTWInfo.CustomStorageClass`) will be ignored during code generation. In such cases, a warning is displayed at the time `RTWInfo.CustomStorageClass` is assigned, for example

```
foo = Simulink.Signal
foo.RTWInfo.CustomStorageClass = 'Struct'

Warning: The 'CustomStorageClass' property of RTWInfo will have
no effect unless the 'StorageClass' property is set to 'Custom'.
```

Previously, the warning was displayed at the time `RTWInfo.StorageClass` was assigned.

Real-Time Workshop Embedded Coder 3.2 Release Notes

New Features

This section summarizes the new features and enhancements introduced in the Real-Time Workshop Embedded Coder 3.2.

Advanced Code Generation Techniques Documented

A new chapter, “Advanced Code Generation Techniques,” has been added to the the Real-Time Workshop Embedded Coder User’s Guide. This chapter contains complete information on the new features that are summarized in these release notes. In addition, the chapter documents useful code generation, optimization, and customization techniques that have not received wide exposure in previous releases. These include

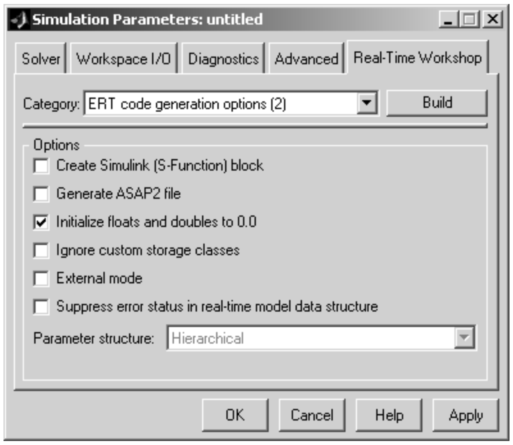
- How to specify target characteristics (such as word sizes for C data types) for the build process, so that generated code is correct for deployment on target hardware
- A general hook file mechanism for adding target-specific customizations to the build process

New Code Generation Options

Several new code generation options have been added, and some changes have been made to the layout of Embedded Real-Time (ERT) target code generation options in the **Real-Time Workshop** pane of the **Simulation Parameters** dialog box.

Options Layout Changes and Additions

The **Suppress error status in real-time model data structure** option has been relocated to the `ERT code generation options (2)` category, as shown in this figure.



Simulation Parameters: untitled

Solver | Workspace I/O | Diagnostics | Advanced | Real-Time Workshop

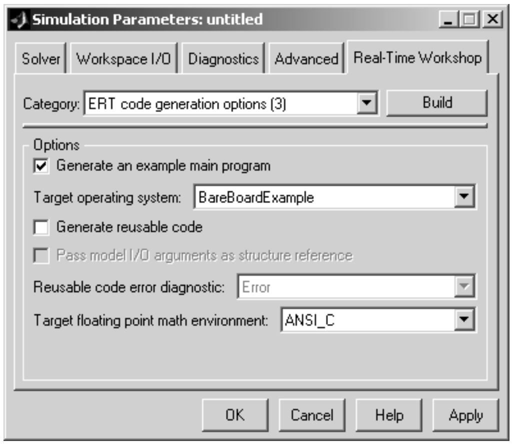
Category: ERT code generation options (2) | Build

Options
- [ ] Create Simulink (S-Function) block
- [ ] Generate ASAP2 file
- [x] Initialize floats and doubles to 0.0
- [ ] Ignore custom storage classes
- [ ] External mode
- [ ] Suppress error status in real-time model data structure

Parameter structure: Hierarchical

OK | Cancel | Help | Apply

A new code generation option, **Pass model I/O arguments as structure reference**, is now available in the `ERT code generation options (3)` category, as shown below. This option is described in “Passing Model I/O Arguments to the model_step Function” on page 2-6.



Simulation Parameters: untitled

Solver | Workspace I/O | Diagnostics | Advanced | Real-Time Workshop

Category: ERT code generation options (3) | Build

Options
- [x] Generate an example main program

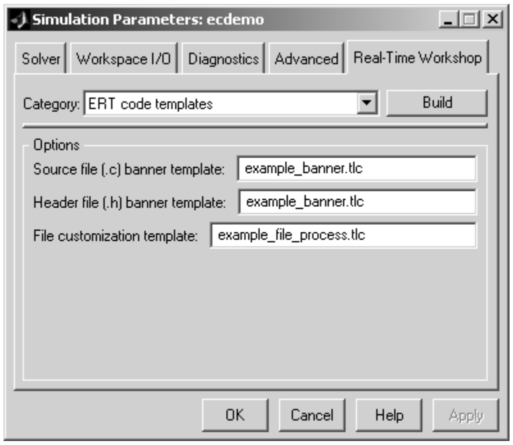
Target operating system: BareBoardExample
- [ ] Generate reusable code
- [ ] Pass model I/O arguments as structure reference

Reusable code error diagnostic: Error

Target floating point math environment: ANSI_C

OK | Cancel | Help | Apply

A new group of options supporting use of *code templates*, a powerful and simple technique for customizing generated code, has been added. These options are available in the `ERT code templates` category of the **Real-Time Workshop** pane of the **Simulation Parameters** dialog (see the figure below). Code templates are summarized in “Code Templates for Customizing Generated Code” on page 2-5.



Category: ERT code templates

Source file (.c) banner template: example_banner.tlc

Header file (.h) banner template: example_banner.tlc

File customization template: example_file_process.tlc

Auto-Configuration of Models for Code Generation

The Real-Time Workshop Embedded Coder now supports automated configuration of all (or selected) model parameters during the code generation process. By automatically configuring a model in this way, you can avoid manually configuring models. This saves time and eliminates potential errors.

Auto-configuration is performed by executing an M-file (referred to as a *hook file*) that is executed as part of the the target build process. Therefore, auto-configuration becomes a function of the target that invokes the hook file. You can direct the automatic configuration process to save existing model settings before code generation and restore them afterwards, so that the user’s manually chosen options are not disturbed.

The automatic configuration process, and utilities provided to support auto-configuration, are described in the “Advanced Code Generation Features” chapter of the Real-Time Workshop Embedded Coder User’s Guide.

Optimized ERT Targets for Fixed-Point and Floating-Point Code Generation

To make it easier for you to customize a hook file that is optimized for your target hardware, Real-Time Workshop Embedded Coder provides two variants of the ERT target:

- `RTW Embedded Coder (auto configures for optimized fixed-point code)`: To optimize for fixed-point code generation, select this target from the System Target File Browser.
- `RTW Embedded Coder (auto configures for optimized floating-point code)`: To optimize for floating-point code generation, select this target from the System Target File Browser.

The use of these targets is detailed in the "Advanced Code Generation Features" chapter of the Real-Time Workshop Embedded Coder User's Guide.

Code Templates for Customizing Generated Code

The ERT target now supports use of *custom file processing templates* (CFP templates).

A CFP template is a Target Language Compiler (TLC) file that calls a high-level applications programming interface (API), referred to as the *code template* API. The code template API simplifies generation of custom source code by letting you

- Generate virtually any type of source (`.c`) or header (`.h`) file. A CFP template can emit code to the standard generated model files (e.g., `model.c`, `model.h`, etc.) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, typedefs, functions, and more). Your CFP template can emit code (e.g., functions), directives (such as `#define` or `#include` statements), or comments into each section as required.
- Generate code to call model functions such as `model_initialize`, `model_step`, etc.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the files being generated from it.

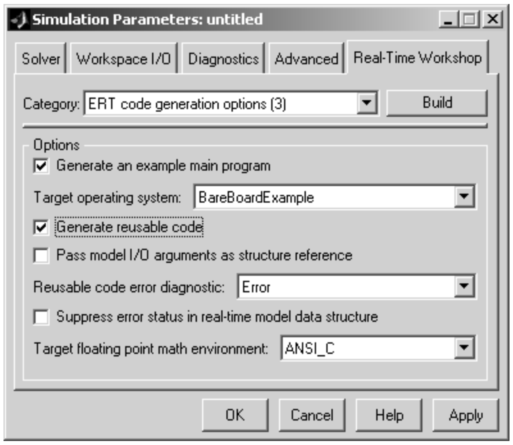
CFP templates are described in the “Advanced Code Generation Features” chapter of the Real-Time Workshop Embedded Coder User’s Guide.

Custom File Banner Generation

The ERT target now supports use of *banner templates* during code generation. A banner template is a TLC file that specifies banner and trailer comments that are emitted to generated source (`.c`) and header (`.h`) files. Banner templates are described in the “Advanced Code Generation Features” chapter of the Real-Time Workshop Embedded Coder User’s Guide.

Passing Model I/O Arguments to the model_step Function

A new code generation option, **Pass model I/O arguments as structure reference**, lets you control how model inputs and outputs at the root level of the model are passed in to the `model_step` function. This option is available in the `ERT code generation options (3)` category of the **Real-Time Workshop** pane of the **Simulation Parameters** dialog box. When **Generate reusable code** is selected, **Pass model I/O arguments as structure reference** is enabled, as shown in this figure.


Simulation Parameters: untitled

Category: ERT code generation options (3)

Options

✓ Generate an example main program

Target operating system: BareBoardExample

✓ Generate reusable code

☐ Pass model I/O arguments as structure reference

Reusable code error diagnostic: Error

☐ Suppress error status in real-time model data structure

Target floating point math environment: ANSI_C

When **Pass model I/O arguments as structure reference** is deselected (the default), each root-level model input and output is passed to `model_step` as a separate argument. When this option is selected, all root-level inputs are packed into a `struct` that is passed to `model_step` as an argument. Likewise, all root-level outputs are packed into a `struct` that is also passed to `model_step` as an argument. Selecting **Pass model I/O arguments as structure reference** can reduce the number of arguments passed in to `model_step`.

See the “Code Generation Options and Optimizations” chapter of the Real-Time Workshop Embedded Coder User’s Guide documentation for further details.

Real-Time Workshop Embedded Coder 3.1 Release Notes

New Features

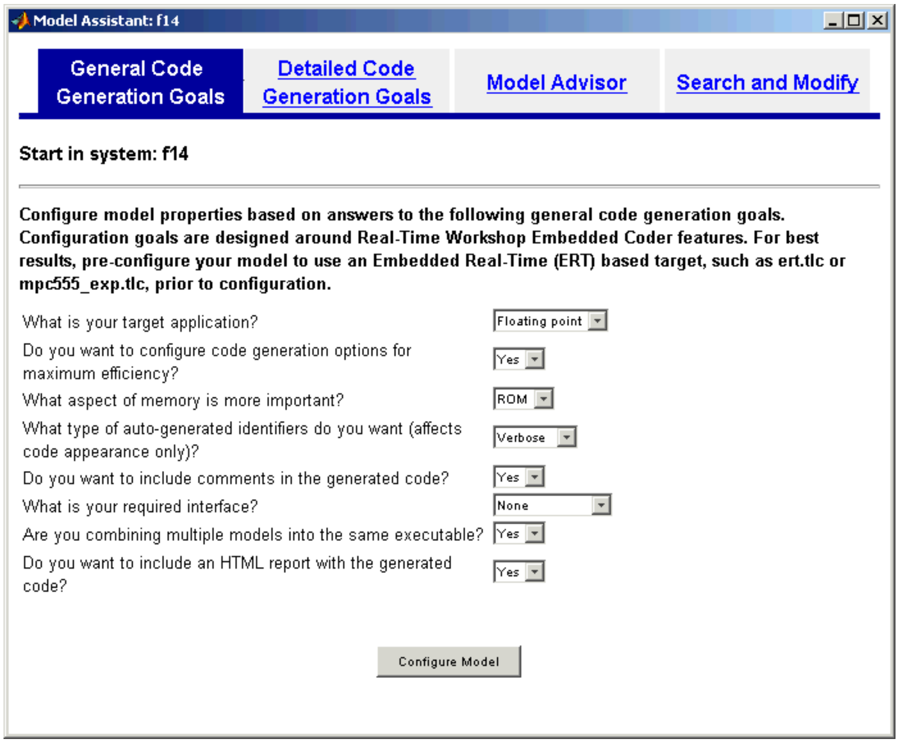
This section summarizes the new features and enhancements introduced in the Real-Time Workshop Embedded Coder 3.1.

Model Assistant Tool

The Model Assistant Tool is a utility that lets you configure a model for code generation quickly. The Model Assistant Tool also helps you to identify aspects of your model that impede production deployment or limit code efficiency. You can use the Model Assistant Tool at any point in your design cycle, as it is completely independent from the code generation process.

The Model Assistant Tool is designed primarily for use with Real-Time Workshop Embedded Coder. It works most effectively with the Embedded Real-Time (ERT) target and with ERT-based targets (such as the Embedded Target for Motorola MPC555). It will also operate with other targets.

The figure below shows the top-level window of the Model Assistant Tool.



Start in system: f14

Configure model properties based on answers to the following general code generation goals. Configuration goals are designed around Real-Time Workshop Embedded Coder features. For best results, pre-configure your model to use an Embedded Real-Time (ERT) based target, such as ert.tlc or mpc555_exp.tlc, prior to configuration.

What is your target application?	Floating point
Do you want to configure code generation options for maximum efficiency?	Yes
What aspect of memory is more important?	ROM
What type of auto-generated identifiers do you want (affects code appearance only)?	Verbose
Do you want to include comments in the generated code?	Yes
What is your required interface?	None
Are you combining multiple models into the same executable?	Yes
Do you want to include an HTML report with the generated code?	Yes

Four main components of the Model Assistant Tool provide a powerful and centralized interface for configuring settings for Simulink blocks, Stateflow charts, models and subsystems. You select these components via the four buttons at the top of the Model Assistant display:

- **General Code Generation Goals**
- **Detailed Code Generation Goals**
- **Model Advisor**
- **Search and Modify**

These components are summarized in the next sections.

General Code Generation Goals

This component lets you quickly configure code generation settings based on specific goals, such as whether to optimize for RAM or ROM usage. Once you have decided the overall optimization and trade-offs for your application, the Model Assistant Tool will select the model settings that best suit your goals.

Detailed Code Generation Goals

This component presents a centralized interface to the available code generation options. Options are grouped by category, and are applied across products.

Model Advisor

The Model Advisor component is is particularly useful early in the design cycle. It provides an analysis of your model to ensure that you best utilize Real-Time Workshop Embedded Coder. You can check selected aspects of your model settings (for example, to identify possible inefficiencies such as blocks that generate saturation and rounding code) or choose **Select All** for a comprehensive analysis.

Search and Modify

This component is a powerful model search and modify engine. It reduces the effort of configuring a model block by block. The search feature helps you find attributes of blocks, lines, input ports, output ports, and annotations quickly. The modify feature lets you perform rapid batch operations on the search results. Frequently performed tasks are packaged conveniently into a single button click.

The **Search and Modify** component includes the following features:

- The **Frequent tasks** page lets you quickly perform common actions.
- The **Simulink object search** page lets you specify a general Simulink object search and modify action. This search mechanism is useful when you know the specific names of underlying attributes.

- The **Stateflow object search** page lets you quickly configure the Stateflow data in your model. This is particularly useful for converting data from floating point to fixed-point types.
- The **Search and replace Simulink text** page lets you quickly modify text for objects in Simulink. For example, you can change all occurrences of `'K1'` to `'K2'`. The semantics of the search and replace are the same as for the Stateflow search and replace tool that ships with Stateflow.
- Two **Parameter name search** mechanisms are provided:
  - Search and modify parameters using prompt strings. This search mechanism is useful when you know the parameter by its dialog prompt string, but you don't know the name of the underlying attribute.
  - "Fuzzy" search using property and/or value pairs. This search mechanism is useful for isolating the name of an underlying attribute.

Using the Model Assistant Tool

You run Model Assistant Tool from the MATLAB command line, via the `modelassistant` command. Before invoking the Model Assistant Tool, make sure that the desired target (such as the ERT target) is selected in the **Target Configuration** section of the **Real-Time Workshop** pane of the **Simulation Parameters** dialog box.

The following examples illustrate the `modelassistant` command syntax and its possible arguments.

To obtain detailed help on the Model Assistant Tool, type

```
modelassistant('help')
```

To invoke the Model Assistant Tool for the root system of a model, type

```
modelassistant('model_name')
```

where `model_name` is the name of the model.

To invoke the Model Assistant Tool for a particular system in a model, type

```
modelassistant('system_name')
```

where `systen_name` is the name of the system.

You can also invoke the Model Assistant Tool for models and systems using the built-in Simulink `bdroot`, `gcb`, and `gcs` commands. For example:

```
modelassistant(gcs)
```

Further Help and Demos

The above sections have summarized the main features of the Model Assistant Tool. To obtain full online documentation on the Model Assistant Tool, type

```
modelassistant('help')
```

There are also three demo models available for the Model Assistant Tool: `advisordemo1`, `advisordemo2`, and `advisordemo3`.