Real-Time Workshop® Embedded Coder

For Use with Real-Time Workshop®

- Modeling
- Simulation
- Implementation

Module Packaging Features

Version 4

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

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

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

Real-Time Workshop Embedded Coder Module Packaging Features

Revision History

June 2004	Online only	New for Version 4.0 (Release 14)
October 2004	Online only	Revised for Version 4.1 (Release 14SP1)
March 2005	Online only	Revised for Version 4.2 (Release 14SP2)
September 2005	Online only	Revised for Version 4.3 (Release 14SP3)

Contents

1 Getting Started

2 Selecting and Defining Templates

3 Managing the Data Dictionary

4 Customizing with Additional Options

5 Managing File Placement of Data Definitions and Declarations

A Referenced Tables

1

Getting Started

What Is MPF?

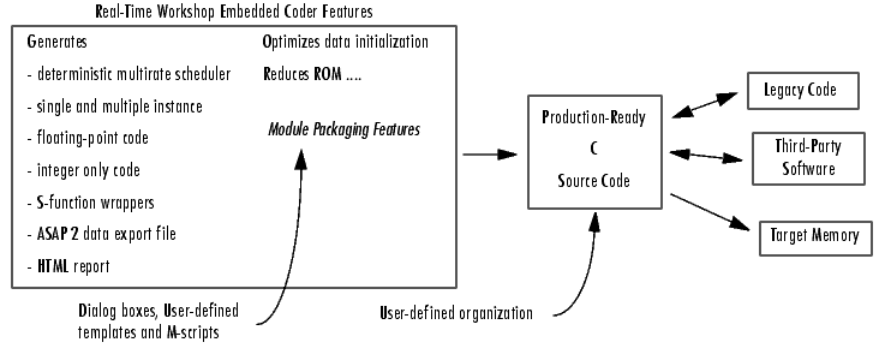
The Real-Time Workshop® Embedded Coder generates C/C++ code for a Simulink® or Stateflow® model. Module packaging features (MPF) extend the code customization and formatting controls of Real-Time Workshop Embedded Coder. It allows you to work collaboratively to develop and deploy large-scale, multimodel control system applications. With MPF, you can control packaging needs, with the following features:

- Package generated code into a desired number of `.c/.cpp` and `.h` files.
- Control the *internal* organization of each generated file by choosing a MathWorks supplied template. Or, if you know TLC (Target Language Compiler), you can modify a MathWorks supplied template or create a new template. For example, for readability, your company may have software standards that define where to place comments and sections of code within files.
- Control whether generated files contain definitions for a model's global identifiers. And, if 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, MPF allows you to

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

The MPF interface consists of dialog boxes, templates you can define, and the use of M-scripts for applying these features to your application.



Module Packaging Features in Code-Generation Process

The term *module* (in *module packaging features*) refers to one or more models. For example, a module might be named `Fuel` and the model files associated with it might be named `open_loop_fuel.mdl` and `closed_loop_fuel.mdl`. Thus, "module" captures the fact that many users generate code for a multimodel system. Using MPF, users generate code for one model at a time. The term "packaging" refers to the ability to organize files.

When this document refers to a variable, it follows the distinction made in C/C++ programming texts between *declaring* and *defining*. Declaring names the variable and specifies its type, but does not allocate memory. Defining names, specifies the type, and allocates memory for the variable. A variable is declared in one of two ways: by placing an `extern` statement in a `.h` file or by placing the `extern` statement at the top of the `.c/.cpp` file that references that variable. A variable is defined in a `.c/.cpp` file.

When Do I Need to Use MPF?

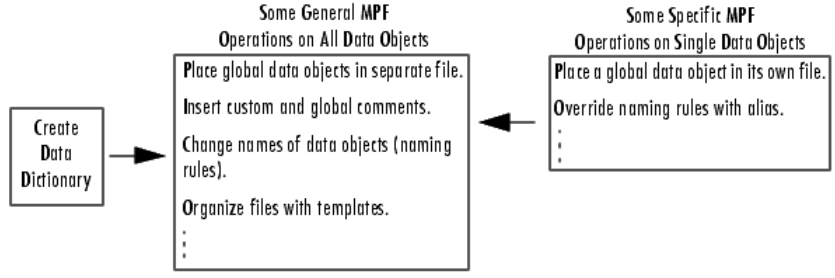
Real-Time Workshop is the foundation for Simulink code generation. It generates ANSI/ISO-C compliant code for an entire model or for an individual subsystem. The code runs on any microprocessor or real-time operating system. Real-Time Workshop Embedded Coder extends Real-Time Workshop. It generates C/C++ code from Simulink and Stateflow models that has the clarity and efficiency of professional handwritten code. This code is compact in size and fast in execution time, meeting the needs of embedded systems, on-target rapid prototyping boards, microprocessors used in mass production, and real-time simulators. MPF extends the code customization and formatting controls of Real-Time Workshop Embedded Coder.

Use MPF if you answer yes to any question like the following:

- Do you need to control the organization of one or more generated files?
- Do you need to control where (which file) the code generator places definitions of global identifiers?
- Do you need to insert any kind of comment into a generated file?
- Do you need to control how model parameters and signals are named in generated files?

MPF General Operations and Specific Overrides

The figure below shows an overview of some of the typical tasks you can perform using module packaging features. First, you can create a data dictionary for a model. The data dictionary consists of data objects that are created from a model's signals, parameters, data stores, and states. You can apply one of more module packaging features to *all* of these data objects in one general operation. You can also override a general operation for specific data objects.



MPF Settings

To enable module packaging features, the Configuration Parameters dialog box must have the settings indicated in the table below:

MPF Settings

Setting on Configuration Parameters Dialog Box	Purpose
Select `Fixed-step` in the **Type** field of the **Solver** pane.	Allows you to choose one of the set of fixed-step solvers that Simulink provides: discrete or continuous. Required to enable any module packaging feature.
Select the **Inline parameters** check box on the **Optimization** pane.	Instructs Real-Time Workshop to embed the numerical values of model parameters (constants), instead of symbolic parameter names, in the generated code. This improves code efficiency, because the constants become nontunable. Then, you can specify individual parameters to be tunable, if desired. Preferred for MPF.
Select an `ert.tlc` (or a system target file derived from an `ert.tlc`) in the **System target file** field on the general **Real-time Workshop** pane.	Sets code generation parameters for your embedded target. (The Target Language Compiler generates target-specific C/C++ code from an intermediate description of your Simulink block diagram (*`model`*`.rtw`). The system target file, at the top level of this program, controls the code generation process.) Required to enable any module packaging feature.

MPF Settings (Continued)

Setting on Configuration Parameters Dialog Box	Purpose
Clear the **Ignore custom storage classes** check box.	Supports all custom storage classes. Required to enable any module packaging feature.
Select the **Include comments** check box on the **Comments** pane, and click the **Apply** button, if it is available.	Makes available all other options on the **Comments** pane. Required to enable the adding custom comments feature of MPF.

Basic Tutorial

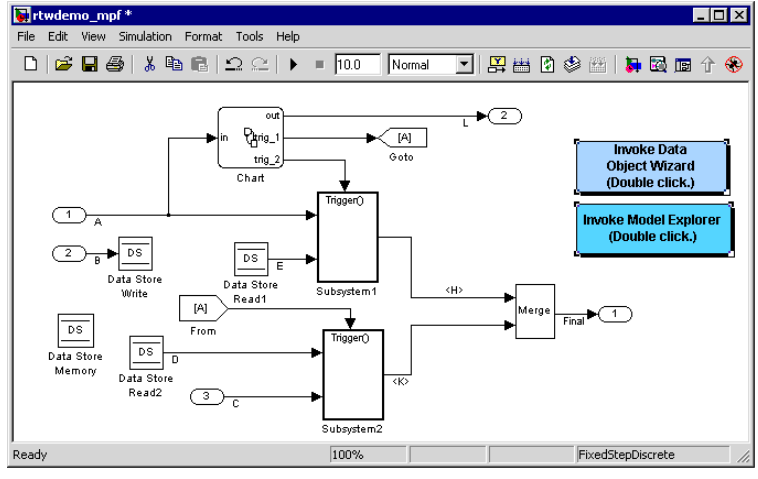
This section explains some basic MPF tasks, using a simple model:

Creating a Data Dictionary for a Model

In this procedure, you create a data dictionary for a model using the Data Object Wizard, inspect the data dictionary, and generate code. Notice that the data objects in the dictionary are defined in the generated file:

Using the Data Object Wizard

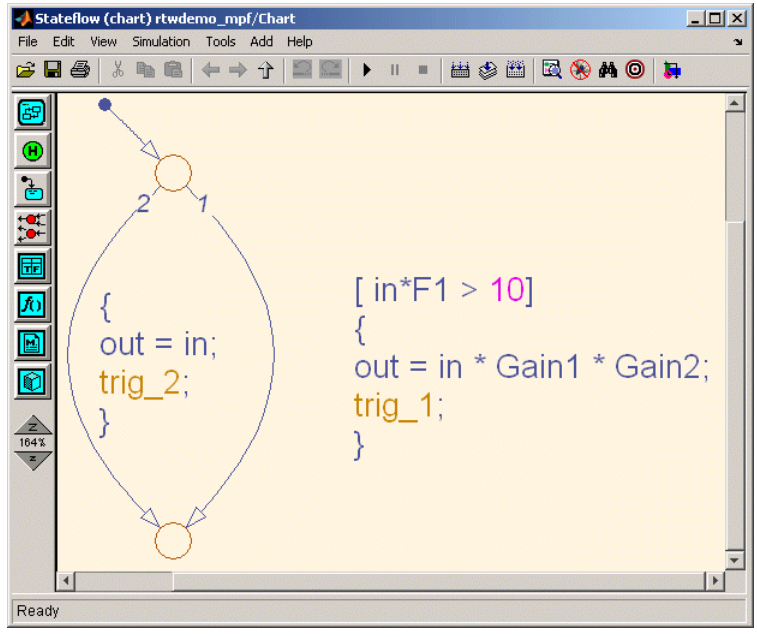
1 Open the demo model `rtwdemo_mpf` by clicking the link or by typing `rtwdemo_mpf` in the MATLAB® Command Window.



In this model,

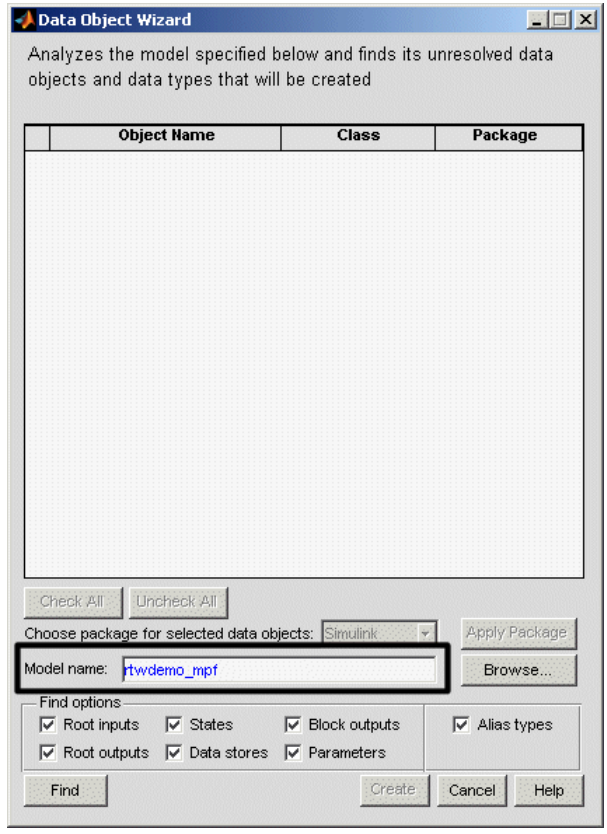
- `A`, `B`, and `C` are input signals, and `L` and `Final` are output signals. (Signals `H` and `K` are inputs to a Merge block. So they will not appear as identifiers in the generated code.)
- Subsystem1 receives inputs `A` and `E`, and contains constants `G1` and `G2`. Signal `E` is an output from Data Store Read1.
- Subsystem2 receives inputs `C` and `D`. Signal `D` is an output from Data Store Read2. There is a constant in Subsystem2 named `G3`. Also, this subsystem has a Unit Delay block whose state name is `SS`.

2 Double-click the Stateflow chart and notice it has constants `F1`, `Gain1`, and `Gain2`, as shown below:



3 Change to a work directory that is not on an installation path and save the model in that work directory. Real-Time Workshop does not allow you to generate code from an installation directory.

4 Double-click the **Invoke Data Object Wizard** button on the model. Or, type `dataobjectwizard('rtwdemo_mpf')` in the MATLAB Command Window. The Data Object Wizard opens and `rtwdemo_mpf` appears in the **Model name** field, as shown below.

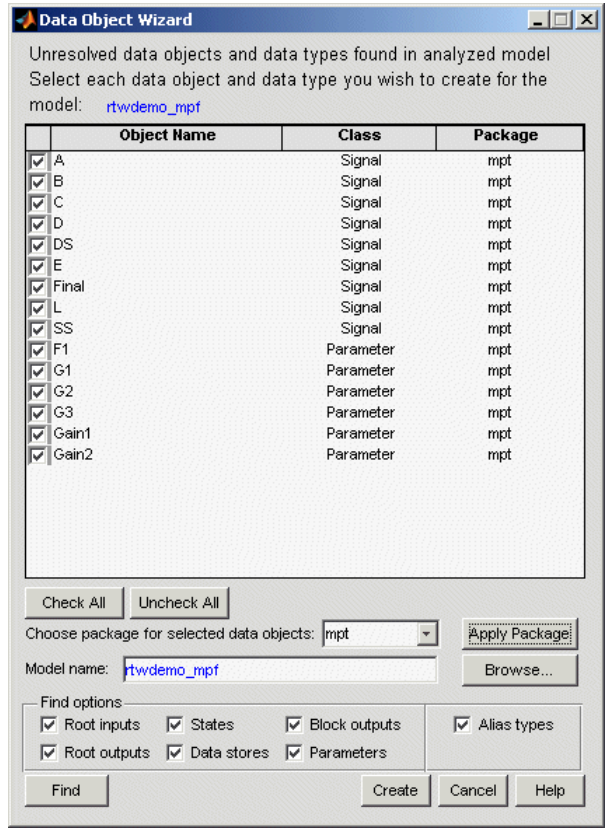


5 Click **Find** on the Data Object Wizard. After a moment, the model's parameters and signals appear in the Data Object Wizard. These "data objects" make up the data dictionary.

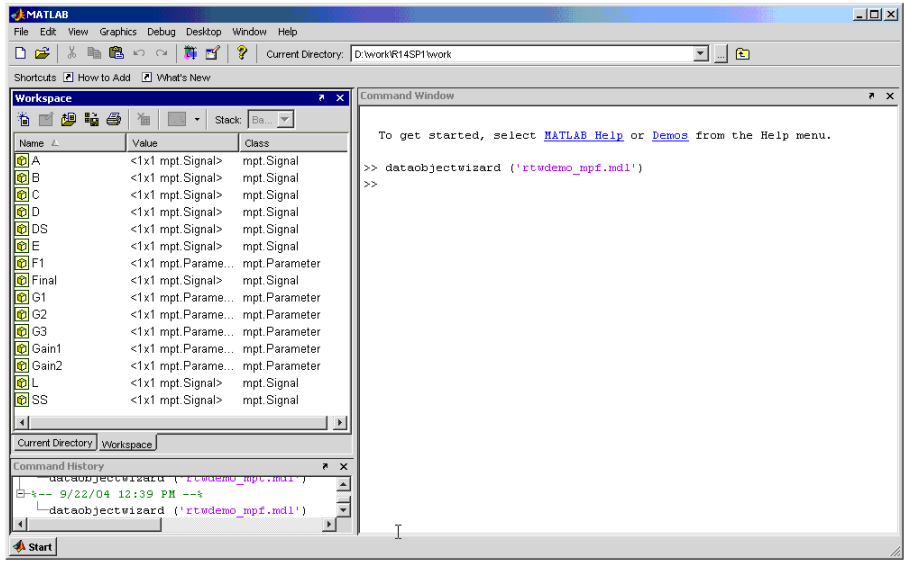
6 Click **Check All**, to select all data objects for the data dictionary.

7 In the **Choose package for selected objects** field, select `mpt`. For an explanation of "package," see "Overview of the Data Dictionary" on page 3-3.

8 Click **Apply Package**. The Data Object Wizard associates the selected data objects with the `mpt` package, as shown below.



9 Click **Create**. The Data Object Wizard creates a data dictionary, consisting of data objects for the selected parameters and signals. The Data Object Wizard removes the objects from its object view. Also, the objects are added to the MATLAB workspace, as shown below.



10 Close the Data Object Wizard.

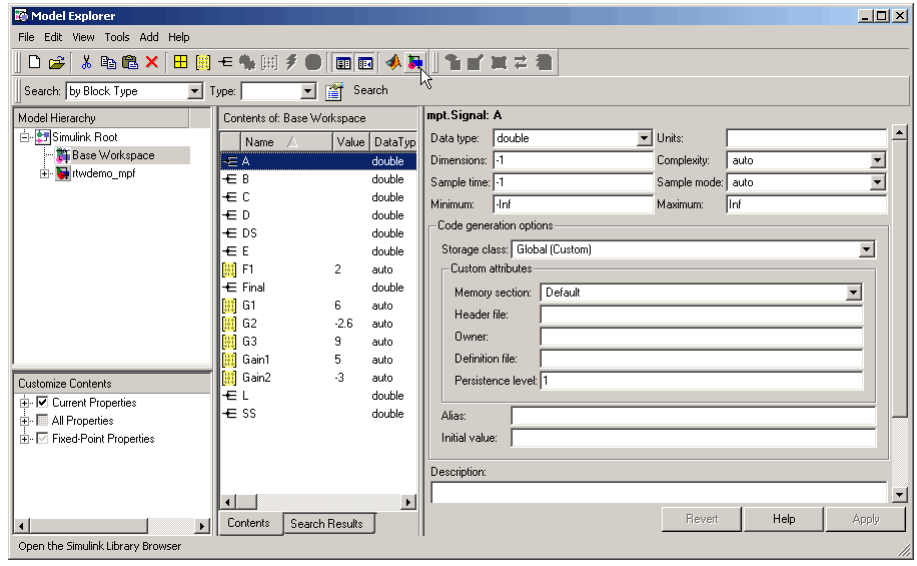
Inspect the Data Dictionary

You can verify that each data object you selected in the Data Object Wizard is in the data dictionary, using the Model Explorer:

1 Open the Model Explorer.

2 In the left pane, select **Base Workspace**. Notice that all data objects that you placed in the data dictionary appear in the middle pane.

3 In the middle pane, select data objects one at a time, and notice their property values in the right pane. The figure below shows this for signal `A`. All of the data objects have default property values. Note that for an `mpt` data object, the default in the **Storage class** field is `Global (Custom)`. For descriptions of the properties on the Model Explorer, see Parameter and Signal Property Values on page A-20.



Generate and Inspect Code

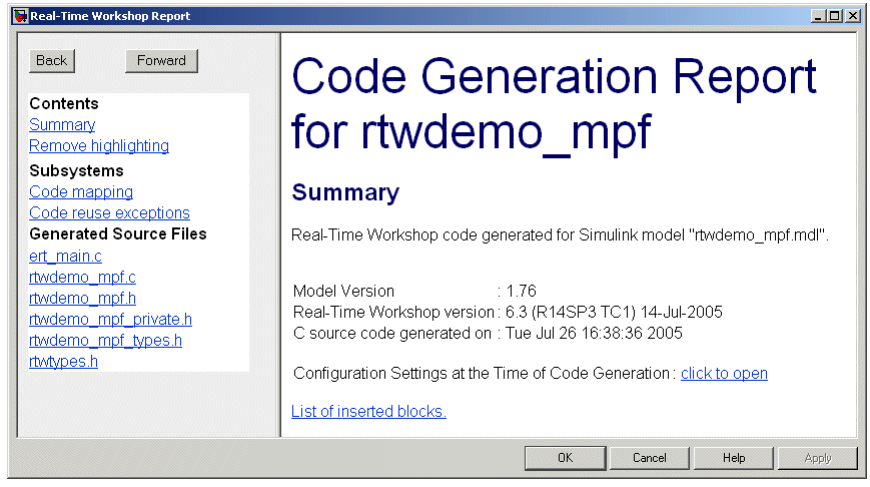
1 In the left pane of the Model Explorer, expand the **rtwdemo_mpf** node.

2 In the left pane, click **Configuration (Active)**.

3 In the center pane, click **Real-Time Workshop**. The active Real-Time Workshop configuration parameters appear in the right pane.

4 Click the **General** tab.

5 In the General pane, select **Generate HTML report** and **Generate code only**, and then click **Generate code**. After a few moments, the names of the generated files are listed on the Real-Time Workshop Report, as shown below.



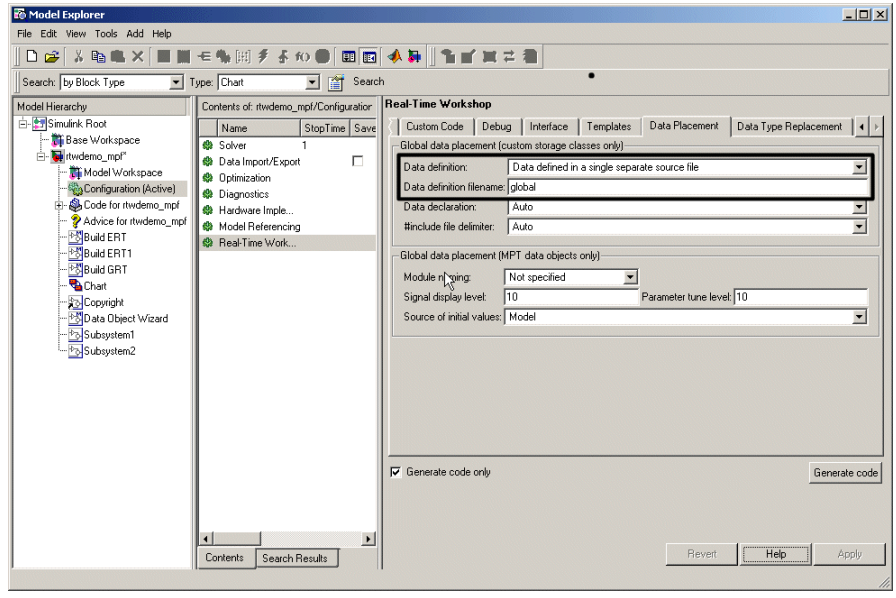
6 Open and inspect the content of the model source file `rtwdemo_mpf.c`. The data objects in the data dictionary are defined in this file.

```
real_T A;
real_T B;
real_T C;
real_T D;
real_T DS;
real_T E;
real_T Final;
real_T L;
real_T SS;
real_T F1 = 2.0;
real_T G1 = 6.0;
real_T G2 = -2.6;
real_T G3 = 9.0;
real_T Gain1 = 5.0;
real_T Gain2 = -3.0;
```

Defining All Global Data Objects in a Separate File

The previous procedure placed all of the model's data objects in the model source file. Now you place all of the global data objects in a file separate from the model source file:

1. In the center pane of the Model Explorer, select **Real-Time Workshop**.
2. In the right pane , select the**Data Placement** tab.
3. Set **Data definition** to `Data defined in single separate source file` and accept the default for **Data definition filename**, `global.c`.



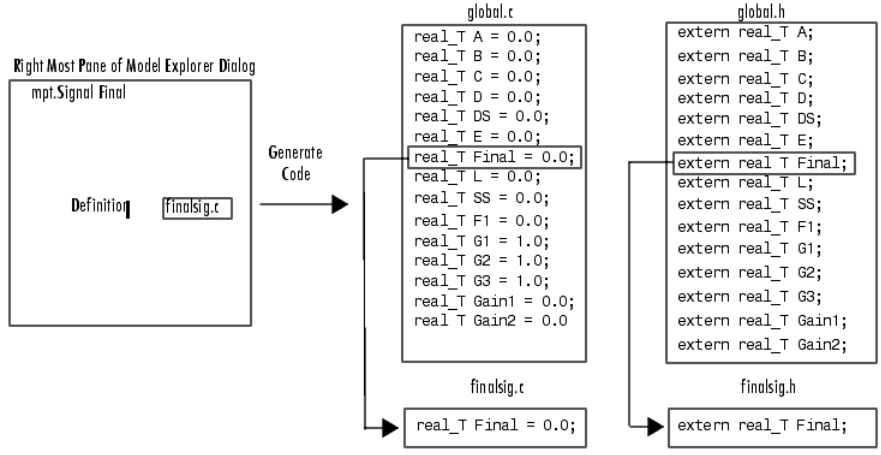
4. Set **Data declaration** to `Data declared in a single separate header file` and accept the default for **Data declaration filename**, `global.h`. Then, click **Apply**.
5. Click **Generate code**. Notice that the code generation report lists `global.c` and `global.h` files.

6 Open `global.c` and `rtwdemo_mpf.c`. Notice that

- The data objects are defined in `global.c` and not in `rtwdemo_mpf.c`.
- The file `rtwdemo_mpf.c` includes `rtwdemo_mpf.h`.
- A `#include "global.h"` statement appears in the file `rtwdemo_mpf.h` .

Defining a Specific Global Data Object in Its Own File

The previous procedure placed all global data objects in a separate definition file, in one operation. You named that file `global.c`. (You named the corresponding declaration file `global.h`.) MPF allows you to override this and place a specific data object in its own definition file. In this procedure, you move the `Final` signal to a file called `finalsig.c`, and keep all the other data objects defined in `global.c`:



1 In the Model Explorer, display the base workspace and select the `Final signal` object. The **mpt.Signal** properties appear in the right pane.

2 In the **Code generation options** section, type `finalsig.c` in the **Definition file** text box, and click **Apply**.

3 Display the active Real-Time Workshop configuration parameters.

4 In the right pane, click **Generate code**. The code generation report still lists `global.c` and `global.h`, but adds `finalsig.c`.

5 Open all four files to inspect them. Notice that the `Final` signal is defined in `finalsig.c`. All other data objects in the dictionary are defined in `global.c`.

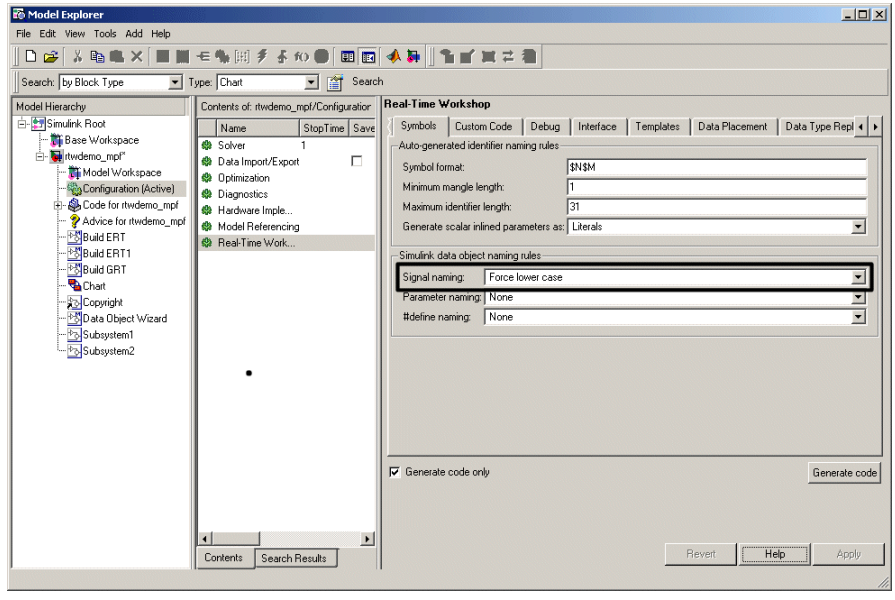
Changing Names of Identifiers

This procedure changes the names of all signal identifiers, except one, so that they are spelled with all lowercase letters. For example, `A` in the definition statement located in `global.c` is changed to `a`. The one exception is the `Final` signal in the `finalsig.c` file. You change this identifier name to `Final_Signal`. The names of the rest of the identifiers in the generated files remain the same:

1 In the center pane of the Model Explorer, click **Real-Time Workshop**.

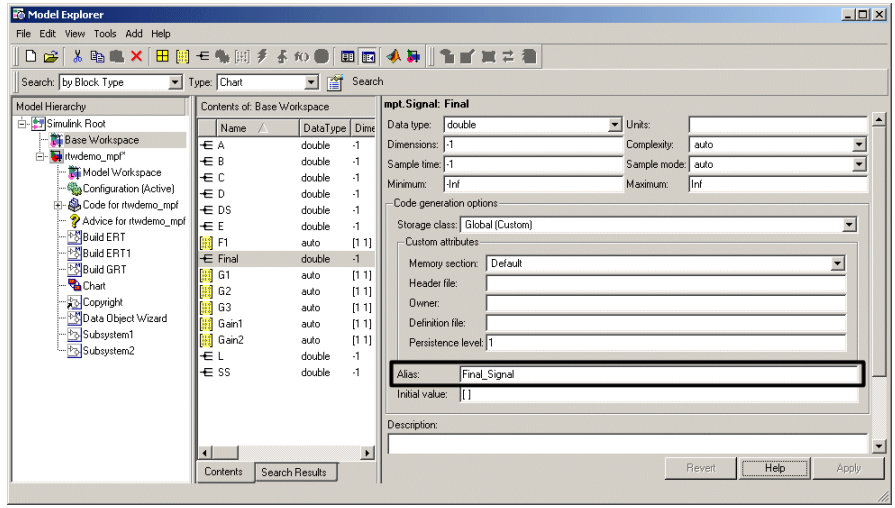
2 In the right pane, click the **Symbols** tab.

3 In the **Simulink data object naming rules** section, set **Signal naming** to `Force lower case`, and click **Apply**.



4 Display the base workspace and select `Final`.

5 In the right pane, type `Final_Signal` in the **Alias** text box, then click **Apply**.



6 Display the active Real-Time Workshop configuration parameters.

7 Click **Generate code**. Now the signal identifiers in `global.c` and `global.h` appear with lowercase letters.

```
real_T F1 = 0.0;
real_T G1 = 1.0;
real_T G2 = 1.0;
real_T G3 = 1.0;
real_T Gain1 = 0.0;
real_T Gain2 = 0.0;
real_T a;
real_T b;
real_T c;
real_T d;
real_T ds;
real_T e;
real_T l;
real_T ss;
```

The statement defining the `Final` signal in `finalsig.c` looks like this:

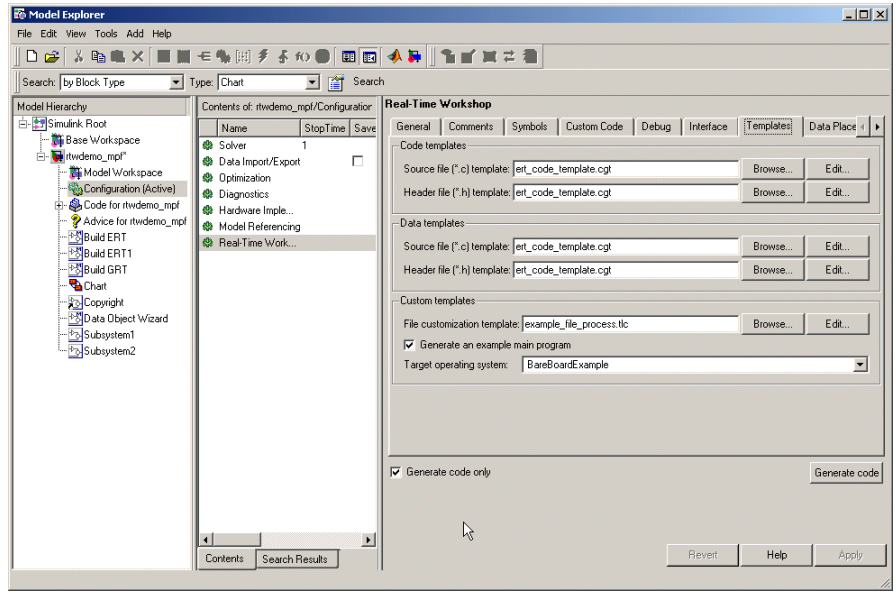
```
real T Final_Signal;
```

The statement declaring this identifier in `finalsig.h` looks like this:

```
extern real_T Final_Signal;
```

Changing the Organization of a Generated File

The files you generated in the previous procedures are organized according to the general template that Real-Time Workshop Embedded Coder provides. This template has the filename `ert_code_template.cgt`, and is specified by default in templates panes of the Configuration Parameters dialog box.



The following fragment illustrates the organization that results from using this default template:

```
/*
 * File: rtwdemo_mpf.c
 *
 * Real-Time Workshop code generated for ... model rtwdemo_mpf.
 *
 * Model version:                       : 1.36
 * Real-Time Workshop file version     : 6.0 (R14) 5 May-2004
 * Real-Time Workshop file generated on : Wed Aug 18 17:27:20 2004
 * TLC version                          : 6.0 (Apr 27 2004)
 * C source code generated on           : Wed Aug 18 17:27:22
 * ...
```

You can change the organization of generated files using code templates and data templates. Code templates organize the files that contain functions, primarily. Data templates organize the files that contain identifiers. In this procedure, you organize the generated files using the supplied MPF code template and data template:

1. Display the active Real-Time Workshop **Templates** configuration parameters.
2. In the **Code templates** section of the **Templates** pane, type `code_c_template.cgt` into the **Source file (*.c) templates** text box.
3. Type `code_h_template.cgt` into the **Header file (*.h) templates** text box.
4. In the **Data templates** section, type `data_c_template.cgt` into the **Source file (*.c) templates** text box.
5. Type `data_h_template.cgt` into the **Header file (*.h) templates** text box, and click **Apply**.

6 Click **Generate code**. Now the files are organized using the templates you specified. For example, `rtwdemo_mpf.c` now is organized like this:

```
/**
 ************************************************
 ** FILE INFORMATION:
 ** Filename:                rtwdemo_mpf.c
 ** File Creation Date:      18-August-2004
 **
 ** ABSTRACT:
 **
 **
 ...
```

Inserting a Comment into Generated Files

MPF provides a variety of ways to enter comments in the generated files, as explained in Chapter 4, “Customizing with Additional Options”. In this final step of the basic tutorial, you place a Simulink annotation on the model so that it also appears as a comment in the "NOTES" section of a generated file.

Recall the templates that you specified in the previous procedure. Below is a list of generated files and templates used to organize them:

Generated File	Template Used
finalsig.c	data_c_template.cgt
global.c	data_c_template.cgt
rtwdemo_mpf.c	code_c_template.cgt
global.h	data_h_template.cgt
rtwdemo_mpf.h	code_h_template.cgt

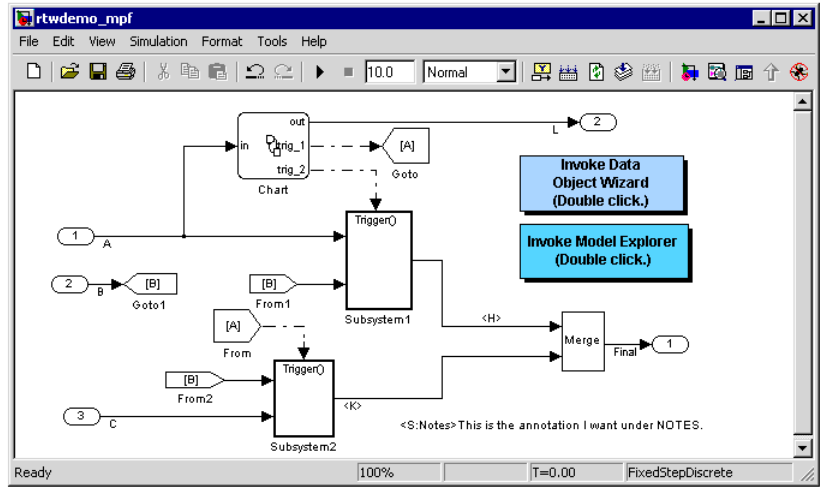
Of the templates you used, only the `code_c_template.cgt` file has the `<%Notes>` template symbol, as shown in the file fragment below:

```
/**
 ************************************************
 ** FILE INFORMATION:
 ** Filename:           %<FileName>
 ** File Creation Date: %<Date>
 **
 ** ABSTRACT:
<%Abstract>
 **
 ** NOTES:
 %<Notes>
 **
 ** MODEL INFORMATION:
 ** Model Name:           %<ModelName>
 ...
```

This template was used to organize `rtwdemo_mpf.c`. So the annotation you are about to add using the `<%Notes>` template symbol will appear in this file only.

1 Double-click the unoccupied area on the model where you want to place the annotation, and type the following, as shown in the figure below:

```
<S:Notes>This is the annotation I want under NOTES.
```



2 Click outside the annotation rectangle and save the model.

3 Generate code. The annotation appears under `NOTES`, in `rtwdemo_mpf.c`:

```
...
** FILE INFORMATION:
** Filename:           rtwdemo_mpf.c
** File Creation Date: 18-August-2004
**
** ABSTRACT:
**
** NOTES:
**This is the annotation I want under NOTES.**
** MODEL INFORMATION:
** Model Name:           rtwdemo_mpf
...
```

Selecting the Desired MPF Procedure

The following chapters document MPF tasks in detail:

- Chapter 2, “Selecting and Defining Templates”
- Chapter 3, “Managing the Data Dictionary”
- Chapter 4, “Customizing with Additional Options”
- Chapter 5, “Managing File Placement of Data Definitions and Declarations”

2

Selecting and Defining Templates

Overview of Templates (p. 2-2)	Explains what a template is.
Selecting Preexisting Templates (p. 2-5)	Explains how to select default templates or user-defined templates that already exist.
Defining Templates (p. 2-8)	Explains how to create a new template or edit an existing template.

Overview of Templates

You can select and define (create) templates so that the code you generate is organized the way you want. A template defines exactly where all parts of a generated file's contents will be placed. Then, when you instruct Real-Time Workshop Embedded Coder to generate code, it will organize all generated files according to the templates you selected.

The table below lists all of the files that Real-Time Workshop Embedded Coder generates, and the *supplied* MPF templates that organize them. The *MPF* template files are `code_c_template.cgt`, `code_h_template.cgt`, `data_c_template.cgt`, and `data_h_template.cgt`. (The `ert_code_template.cgt` file is the default template that Real-Time Workshop Embedded Coder provides. The `example_file_process.tlc` file is the custom template, referenced below.)

Generated Files and Templates That Organize Them

Generated File	ert_code_template.cgt	code_c_template.cgt	code_h_template.cgt	data_c_template.cgt	data_h_template.cgt	example_file_process.tlc
`your_code.c` file or files	x	x				x
`your_code.h` file	x		x			x
`your_data.c` file	x			x		x
`your_data.h` file	x				x	x

Template files are grouped into three types: code, data, and custom.

A Code template organizes all of the generated files that, primarily, contain functions but not identifiers. The *source* code template organizes C/C++ code files. These include, for example, the main `.c` or any of the `.c` files that contain functions that Real-Time Workshop Embedded Coder generates for the open model.

The quantity and filenames of these `.c` files are based on the function partitioning selected in Simulink for the model. See "Nonvirtual Subsystem Code Generation" in the Real-Time Workshop documentation and "Generated Code Modules" in the Real-Time Workshop Embedded Coder documentation.

There will always be at least one `.c` file generated that contains the model's functions. The code generator uses the source code template that you select to organize all of the function `.c` files, regardless of how many there are for this model. The *header* code template, on the other hand, organizes the `.h` file that includes the prototypes of these functions.

A Data template organizes all of the generated files that contain only identifiers (data), not functions (code). The *source* data template organizes the `.c` file that contains definitions of variables of global scope. The *header* data template organizes the `.h` file that can contain declarations to those definitions.

A Custom template has priority over the code and data templates in organizing the generated files. As its name suggests, this template is for advanced users who want to customize how the generated files are organized, by using this one template. A custom template lets you

- Generate virtually any type of source (`.c`) or header (`.h`) file.
- Organize generated code into sections (such as `#include` preprocessor directives, `typedef` statements, functions, and more).
- Generate code to call model functions such as `model_initialize` and `model_step`.
- 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.

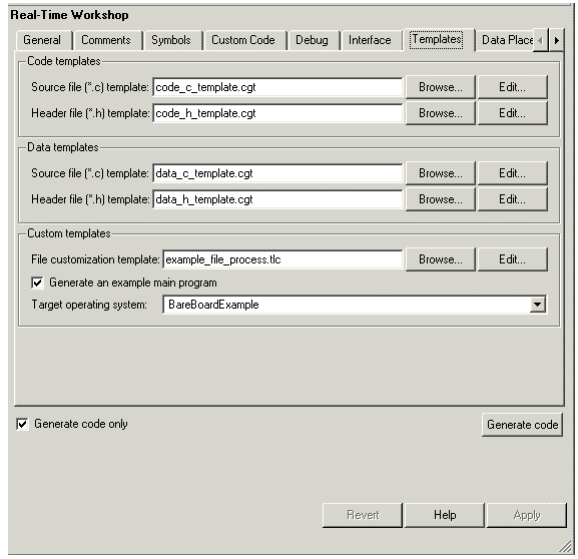
The chapter has two main subprocedures:

- "Selecting Preexisting Templates" on page 2-5 describes how to select preexisting code and data templates.
- "Defining Templates" on page 2-8 describes how to create your own code or data templates.

For details describing the custom template, see the discussion of "CFP Template Structure" in the Real-Time Workshop Embedded Coder documentation.

Selecting Preexisting Templates

The following figure shows the **Templates** pane of the Configuration Parameters dialog box, as it appears in the Model Explorer.



The fields on this pane allow you to specify template files that Real-Time Workshop Embedded Coder uses to organize all of the generated `.c` or `.h` files.

To modify template options.

1. Open the Configuration Parameters dialog box.
2. Select the **Real-Time Workshop > Templates** pane. For an explanation of fields on this pane, see “MPF Panes on the Configuration Parameters Dialog Box” on page A-2.

Note A directory path to the MPF templates is created during installation in the MATLAB folder. It is `toolbox/rtw/targets/ecoder`. For a filename that you typed, that filename must be in either the current MATLAB working directory or on the MATLAB path.

3. In the **Source file (*.c) template** field of the **Code templates** pane, select the desired filename. Real-Time Workshop Embedded Coder uses this file to organize the `.c` file or files that contain the source code for the model's functions. The supplied MPF source file template is `code_c_template.cgt`.

4. In the **Header file (*.h) template** field of the **Code templates** pane, select the desired filename. Real-Time Workshop Embedded Coder uses this file to organize the `.h` header file that contains the model's function prototypes. The supplied MPF header file template is `code_h_template.cgt`.

5. In the **Source file (*.c) template** field of the **Data templates** pane, select the desired filename. Real-Time Workshop Embedded Coder uses this file to organize the `.c` file that contains the definitions of variables of global scope. The supplied MPF source file template is `data_c_template.cgt`.

6. In the **Header file (*.h) template** field of the **Data templates** pane, select the desired filename. Real-Time Workshop Embedded Coder uses this file to organize the `.h` file that contains declaration statements (`extern`, `typedef`, `#define`). The supplied MPF header file template is `data_h_template.cgt`.

   If you want to use a custom template, follow the "Custom File Processing" instructions in the Real-Time Workshop Embedded Coder documentation. Otherwise, proceed to the next step.

7. Click **Apply** to save all your choices on the pane and keep it open. (Clicking **OK** saves the choices but closes the pane.)

Now you can generate code using the selected template files.

Generating Code and Inspecting Files

You have selected the desired templates. Now you can generate code and inspect the files to ensure they are what you want:

1. On the Configuration Parameters dialog box, click **Real-Time Workshop** on the left pane.

2. In the **Documentation** pane, select the **Generate HTML report** check box.

   When you select the **Generate HTML report** check box, Real-Time Workshop Embedded Coder automatically selects the two check boxes under it: **Include hyperlinks to model** and **Launch report after code generation completes**. For large models, you may find that HTML report generation takes longer than you want, after performing step 4 below. In this case, consider clearing the **Include hyperlinks to model** check box. The report will be generated faster.

3. On the Configuration Parameters dialog box, select the **Generate code only** check box. The **Build** button changes to **Generate code**.

   The generate code process generates the `.c` and `.h` files. The build process adds compiling and linking to generate the executable. For details on building, see "Build Process" in the Real-Time Workshop documentation.

4. Click the **Generate code** button. After a moment, Real-Time Workshop Embedded Coder creates all files according to the Simulink partitioning for the model. It organizes each file according to the respective template you have chosen. The HTML report appears, listing the generated files on the left pane (under Generated Source Files).

5. To inspect a file, click its filename on this window.

6. If you want a file to be organized using a different existing template, close the file, and repeat the relevant steps in "Selecting Preexisting Templates" on page 2-5.

7. If you want to change a template, or create a new one, close the file, and follow "Defining Templates" on page 2-8.

Defining Templates

Follow this procedure to create a new template or edit an existing template. When creating a new template, we recommend that you modify its supplied template and save it with a new filename. Templates have the extension `.cgt`, which stands for "code generation template." A default path to templates is created during installation in the MATLAB folder: `toolbox/rtw/targets/ecoder`. So templates are located there (unless you changed this path). For a filename typed in a template field on the **Templates** pane to be selected, the file must be in either the current MATLAB work directory or on the MATLAB path. For an example that compares a template with its associated generated file, see “Comparison of a Template and Its Generated File” on page 2-9:

1. Open the Configuration Parameters dialog box and select **Templates** on the left pane. The **Templates** pane now appears on the right, like that shown in the section “Overview of Templates” on page 2-2.

   Each Stateflow or Simulink model can have up to five types of templates from which `.c` or `.h` files are generated. These templates are accessible on this pane. Generated Files and Templates That Organize Them on page 2-2, identifies all the files that Real-Time Workshop Embedded Coder generates and the supplied templates that organize each file. MPF Elements on Configuration Parameters Panes on page A-2, describes the supplied code templates and data templates.

2. To edit a code or data template, first type its filename in the desired template field on the **Templates** pane, or select it using the **Browse** button. Then click **Edit**. The file opens in an editor.

   The location of a template symbol in one of the MPF template files identified in Generated Files and Templates That Organize Them on page 2-2 determines where the items associated with the symbols are located in the generated file, according to certain rules.

3. Modify (edit) the template file as desired, while consulting the following:

   - Template Symbol Groups on page A-11
   - Template Symbols on page A-13
   - “Rules for Modifying or Creating a Template” on page A-17

4 Perform a **Save** or **Save As** operation, naming the template file as desired. Performing a **Save** operation on an existing template file will replace the original. This is desirable if your intent is to update an existing user-defined template. If you are modifying a supplied template, perform a **Save As** operation, not a **Save**.

5 Follow "Selecting Preexisting Templates" on page 2-5, selecting the template you just defined.

6 Click **Generate Code**.

7 Inspect the generated file or files to see how the template organized them.

8 Repeat this procedure only if the organization of the generated file or files is not acceptable.

Note Practice is the best way to learn how a user-defined template affects the organization of a generated file. Create a template. Generate code. Compare the two. Repeat this process to see the results that changes on the template have on its respective generated file or files.

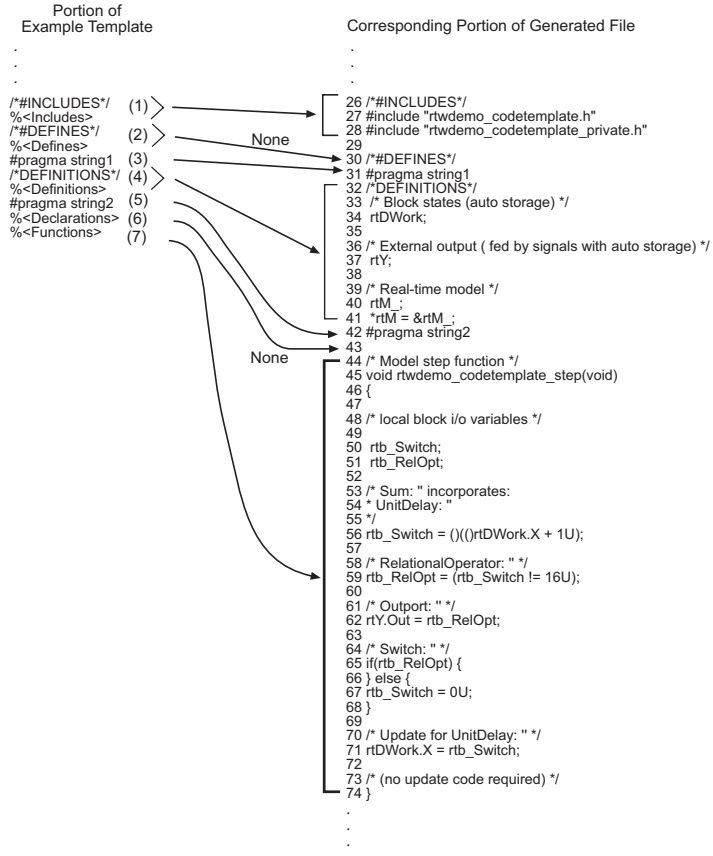
Comparison of a Template and Its Generated File

The next figure shows part of a user-modified MPF template and the resulting code generated by the Real-Time Workshop Embedded Coder. This figure illustrates how you can use a template to

- Define what code the Real-Time Workshop Embedded Coder should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

Notice `%<Includes>`, for example, on the template. The term `Includes` is a symbol name. A percent sign and brackets (`%< >`) must enclose every symbol name. You can add the desired symbol name (within the `%< >` delimiter) at a particular location in the template. This is how you control where the code generator places an item in the generated file.

Template and Generated File



How the Template Affects Code Generation

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(1)	`/*#INCLUDES*/` `%<Includes>`	26–28	An `/*#INCLUDES*/` comment, followed by `#include` statements	The code generator adds the C/C++ comment as a header, and then interprets the `%<Includes>` template symbol to list all the necessary `#include` statements in the file. This code is first in this section of the file because the template entries are first.
(2)	`/*DEFINES*/` `%<Defines>`	30	A `*/DEFINES*/` comment, but no `#define` statements	Next, the code generator places the comment as a header for `#define` statements, but the file does not need `#define`. No code is added.
(3)	`#pragma string1`	31	`#pragma` statements	While the code generator requires `%<>` delimiters for template symbols, it can also interpret C/C++ statements in the template without delimiters. In this case, the generator adds the specified statements to the code, following the order in which the statements appear in the template.
(5)	`#pragma string2`	42		
(4)	`/#DEFINITIONS*/` `%<Definitions>`	32–41	`/*#DEFINITIONS*/` comment, followed by definitions	The code generator places the comment and definitions needed in the file between the `#pragma` statements, according to the order in the template. It also inserts comments (lines 33 and 36) that are preset in the model's Configuration Parameters dialog box.

How the Template Affects Code Generation (Continued)

This part of the template...		Generates in the file...		Explanation
		Line	Description	
(6)	`%<Declarations>`	43	No declarations	The file needs no declarations, so the code generator does not generate any for this file. The template has no comment to provide a header. Line 43 is left blank.
(7)	`%<Functions>`	44–74	Functions	Finally, the code generator adds functions from the model, plus comments that are preset in the Configuration Parameters dialog box. But it adds no comments as a header for the functions, because the template does not have one. This code is last because the template entry is last.

For a list of template symbols and the rules for using them, see Template Symbol Groups on page A-11, Template Symbols on page A-13, and "Rules for Modifying or Creating a Template" on page A-17. To set comment options, from the Simulink menu, select **Configuration Parameters > Real-Time Workshop > Comments**. For details, see "Configuring Real-Time Workshop Code Generation Parameters" in the Real-Time Workshop documentation.

3

Managing the Data Dictionary

Overview of the Data Dictionary (p. 3-3)	Describes the data dictionary created for Simulink and Stateflow models (the "code generation data dictionary").
Creating Simulink and mpt Data Objects (p. 3-5)	Explains how to add Simulink and mpt data objects to the code generation data dictionary.
Saving and Loading Data Objects (p. 3-19)	Explains how to save the set of data objects (and their properties) that you have created so that you can load them for subsequent use.
Applying Naming Rules to Identifiers Globally (p. 3-20)	Explains how to change the case or spelling of all identifier names according to the same rule, when code generation occurs.
Creating User Data Types (p. 3-25)	Explains how to register user-defined data types so they can be associated with the corresponding MathWorks C/C++ data types.
Selecting User Data Types for Signals and Parameters (p. 3-31)	Explains how to select registered, user-defined data types for signals and parameters.

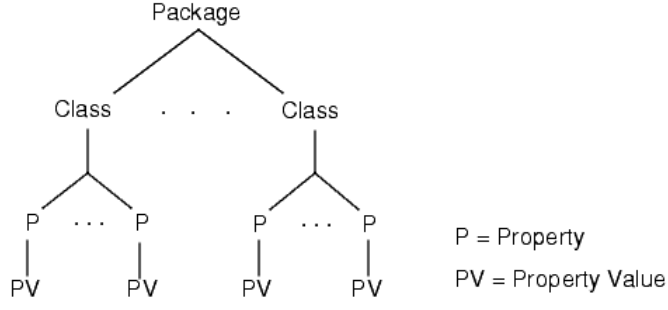
Registering User Object Types (p. 3-38)	Explains how to register one or more sets of user-defined properties and property values that can be applied automatically to user data objects as desired.
Replacing Built-In Data Type Names in Generated Code (p. 3-44)	Explains how to replace built-in data type names with user-defined replacement data type names in generated code.

Overview of the Data Dictionary

A data dictionary contains all of the parameters and signals that the source code uses, and a description of their properties. The data dictionary that is created for Simulink and Stateflow models is called the code generation data dictionary. (You can use the data dictionary for simulation. This does not require that you have a Real-Time Workshop Embedded Coder license.) The dictionary is the total number of data objects that appear in the middle pane of the Model Explorer. These data objects also appear in the MATLAB workspace. The procedure described in this chapter allows you to create or edit the dictionary. The procedure allows you to control property values for each data object. This, in turn, determines how each parameter and signal is defined and declared in the automatically generated code.

The values of data object properties can affect where the code generator places a parameter or signal in the generated file. This is because some property values are associated with different template symbols. The location of a symbol in a template determines where the associated parameter or signal is located in the generated file. For details about templates and symbols, see Chapter 2, “Selecting and Defining Templates”.

It is helpful to define terms you will see when managing the dictionary, especially when you view them using the Model Explorer. In Simulink, there is a hierarchy of terms that are drawn from object-oriented programming. For details, see “Working with Data Objects” in the Simulink documentation. The sketch below summarizes this hierarchy.



`Simulink` or `mpt` is the package. Parameter and Signal are two classes in each of these packages. Each class has a number of properties associated with it. Sometimes properties are called *attributes*. Data objects (the parameters and signals) are the instances of a `package.class` that make up the data dictionary. All parameter data objects have a set of properties. All signal data objects have a different set of properties than that for parameters. For each data object, each property in the set has its own property *value* that must be specified in the dictionary.

Note In this document, "signal" refers to a named wire on a Simulink model, a discrete state, or a data store.

Creating Simulink and mpt Data Objects

There are different ways of creating Simulink and `mpt` data objects for a data dictionary.

- One-by-one, either using the MATLAB command line or using the Model Explorer **Add** menu and selecting **Simulink Parameter**, **Simulink Signal**, **MPT Parameter**, or **MPT Signal**. For more information, see "Working with Data Objects" in the Simulink documentation.
- All at once, invoking the Data Object Wizard for an existing model. For more information and examples, see Data Object Wizard in the Simulink documentation and "Creating Data Objects with Data Object Wizard" on page 3-5.
- Creating data objects based on an external data dictionary. You can do this manually item by item, or all at once automatically using a script. For more information, see "Creating Data Objects Based on an External Data Dictionary" on page 3-17.

The following sections illustrate how to create Simulink and `mpt` data objects and compares their properties as data types.

Creating Data Objects with Data Object Wizard

You can use the Data Object Wizard to create data objects for your model (see Data Object Wizard in the Simulink documentation).

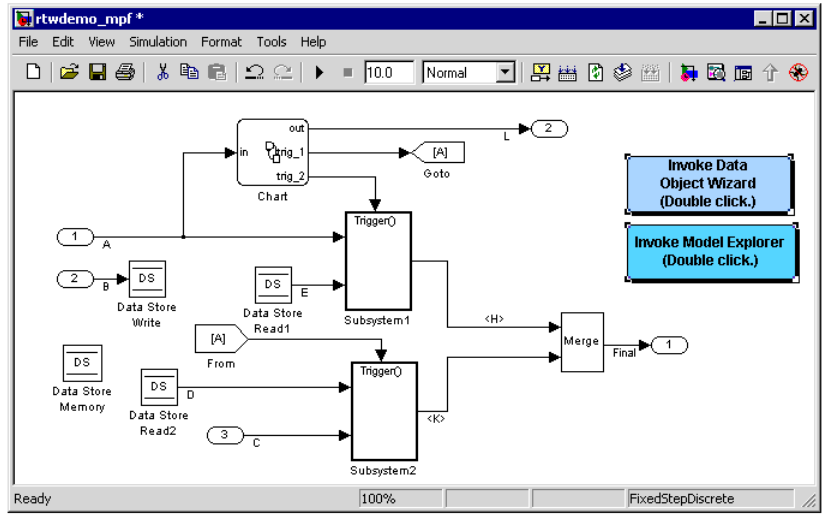
The Data Object Wizard is especially useful for creating multiple data objects for

- Existing models that do not currently use data objects.
- Existing models to which you have added signals or parameters and therefore you need to create more data objects.

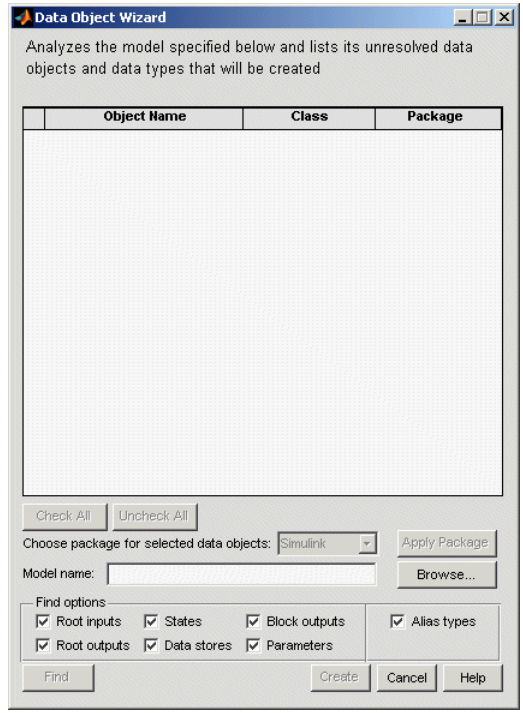
Creating Simulink Data Objects

This procedure creates Simulink data objects using the **Data Object Wizard**.

1 Open the model whose data objects you want to be in the data dictionary. For example, open `rtwdemo_mpf.mdl` (which is located in `toolbox/rtw/rtwdemos`). This model appears as shown below.



2 Open the Data Object Wizard by entering `dataobjectwizard` at the MATLAB command line or by selecting **Data Object Wizard** from the **Tools** menu of your model. The Data Object Wizard dialog box appears, as shown below.



3 In the **Model name** field, type the name of the model you opened in step 1 and press the **Enter** key, or navigate to it using the **Browse** button. The **Find** button becomes available. Notice the check boxes in the **Find options** pane.

4 In the **Find options** pane, select the desired check boxes. For descriptions of each check box, see Data Object Wizard in the Simulink documentation.

Be sure to check the **Alias types** option. This finds all user-registered data types in the `custom_user_type_registration.m` file plus all data type replacements specified for the model in the **Data Type Replacement** pane

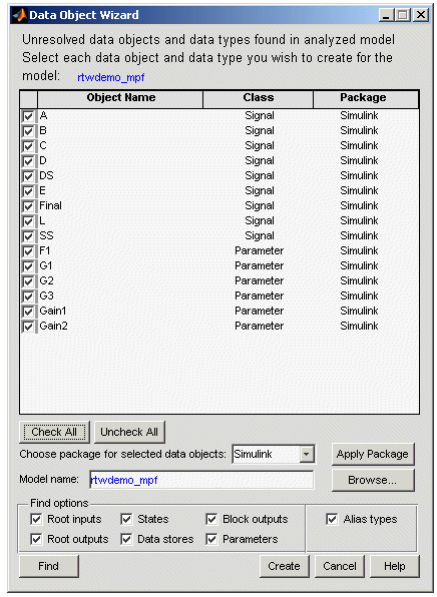
of the Configuration Parameters dialog box. The Data Object Wizard can create `Simulink.AliasType` objects from these.

5 Click the **Find** button. After a moment, a list of all of the model's potential data objects appear that are not yet in the code generation data dictionary, as shown below. This includes all of the model's signals (root inputs, root outputs, and block outputs), discrete states, data stores, and parameters, depending on

- The check boxes you selected in the previous step
- The constraint mentioned in the note above

The Data Object Wizard finds only those signals, parameters, data stores, and states whose storage class is set to `Auto`. The Wizard lists each data store and discrete state that it finds as a signal class.

6 Click **Check All** to select all data objects. Notice in the **Choose package for selected data objects** field that `Simulink`, the default, is selected. So all of the data objects are associated with the `Simulink` package, as shown below.



7 Click **Create**. The data objects are added to the MATLAB workspace, and they disappear from the Data Object Wizard.

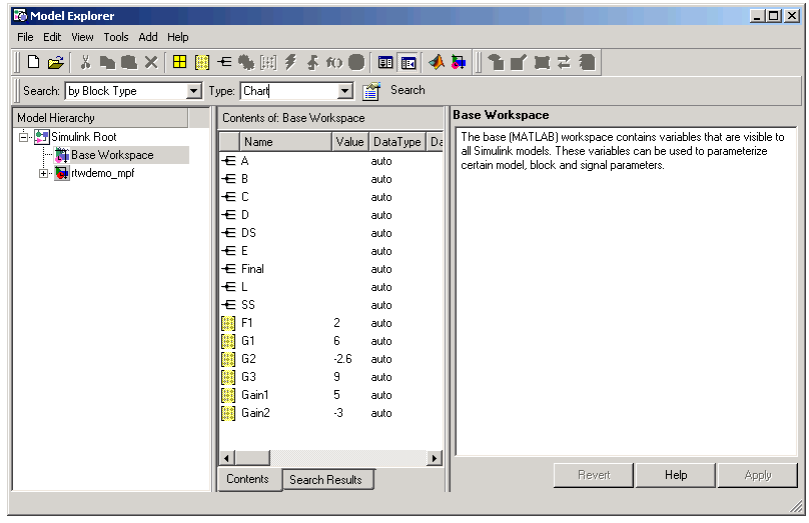
8 Click **Cancel**. The Data Object Wizard disappears.

Now you can set property values for the data objects.

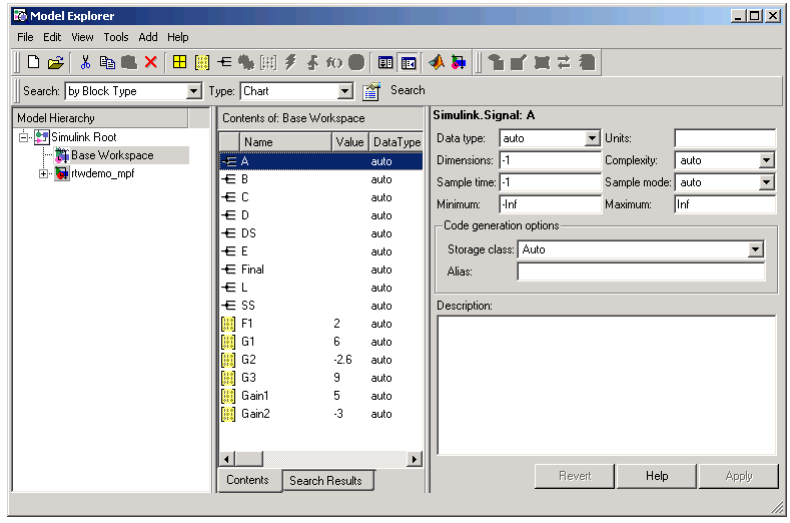
Setting Property Values for the Simulink Data Objects. Most of the property values of data objects are supplied by defaults. A few are from the model. Note that for Simulink data objects, the default storage class is `Auto`.

1 Type `daexplr` on the MATLAB command line, and press **Enter**. The Model Explorer appears.

2 In the Model Hierarchy (left) pane, select **Base Workspace**. All of the Simulink data objects in the code generation data dictionary appear in the **Contents of** (middle) pane, as shown below.

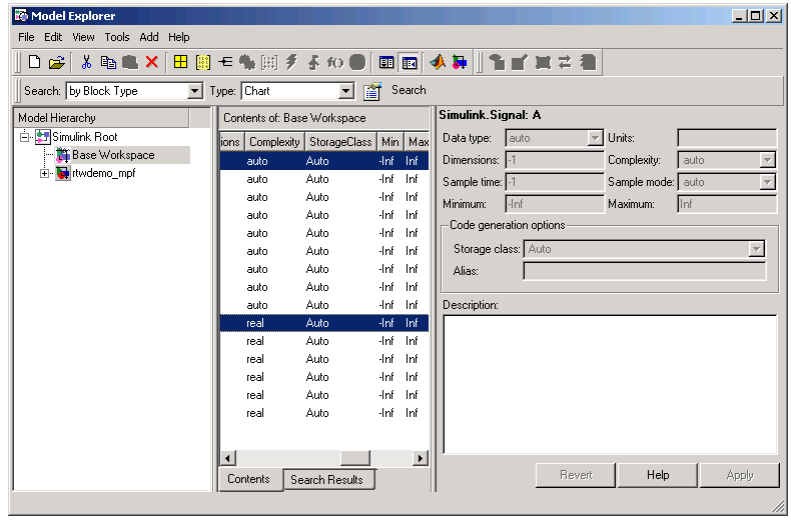


3 To see the properties of a Simulink data object, select a data object in the middle pane. The right pane displays the property names, as shown below. (For descriptions of the properties, see Parameter and Signal Property Values on page A-20.) These property names also appear as column headings in the middle pane. You have control over the values specified for these properties.

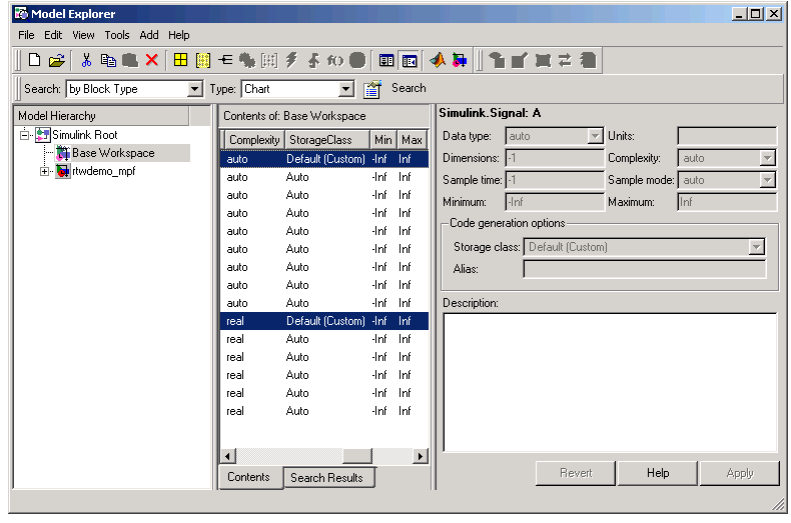


4 For this example, while pressing the **Ctrl** key, select signal data object `A` and parameter data object `F1` in the middle pane.

5 In the middle pane, move the scroll bar so that you can see the **StorageClass** column, as shown below.



6 For this example, click one of these rows and select `Default (Custom)`. The **StorageClass** property value for both of these Simulink data objects changes from the default `Auto` to `Default (Custom)`, as shown below.



Generating and Inspecting Code. All data objects for the model are in the code generation data dictionary. You have specified property values for each data object's properties as needed. Now you generate and inspect the source code, to see if it needs correction or modification. If it does, you can change property values and regenerate the code until it is what you want.

1 In the Configuration Parameters dialog box, click **Real-Time Workshop** in the left pane.

2 In the **Documentation** pane, select the **Generate HTML report** check box.

Note When you select the **Generate HTML report** check box, Real-Time Workshop Embedded Coder automatically selects the two check boxes under it: **Include hyperlinks to model** and **Launch report after code generation completes**. For large models, you may find that HTML report generation takes longer than you want, after performing step 4 below. In this case, especially for large models, consider clearing the **Include hyperlinks to model** check box. The report will be generated faster.

---

3 In the Configuration Parameters dialog box, select the **Generate code only** check box. The **Build** button changes to **Generate code**.

Note The generate code process generates the `.c/.cpp` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see "Build Process" in the Real-Time Workshop documentation.

---

4 Click the **Generate code** button. After a moment, the HTML report appears, listing the generated files on the left pane (under Generated Source Files).

5 Select and review files in the HTML report.

Creating mpt Data Objects, Setting Property Values, and Generating Code

Create `mpt` Data Objects using the **Data Object Wizard** the same way you did for Simulink data objects, as explained in "Creating Simulink Data Objects" on page 3-6, except select `mpt` as the package instead of `Simulink`. The `mpt` data objects contain all the properties of Simulink data objects plus additional properties.

Set the property values for the `mpt` data objects the same way you set them for Simulink data objects, as explained in "Setting Property Values for the Simulink Data Objects" on page 3-9, except note these exceptions:

- Accept the default custom storage class for `mpt` data objects, `Global(Custom)`
- For data objects `A` and `F1`, type `mydefinitionfile` in the **Definition file** field on the Model Explorer.

Then generate and inspect the code.

Note The **Alias** field is related to "Applying Naming Rules to Identifiers Globally" on page 3-20.

Comparing Simulink and mpt Data Objects

You can deduce the added control available for `mpt` data objects compared with that for Simulink data objects, by comparing the

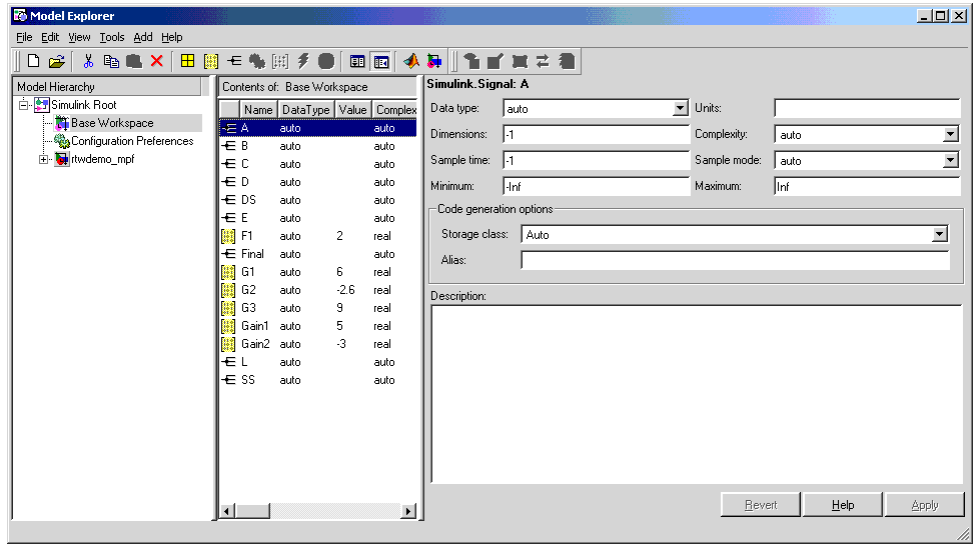
- Properties that appear on the Model Explorer
- Elements that appear on the Configuration Parameters dialog box
- Generated code

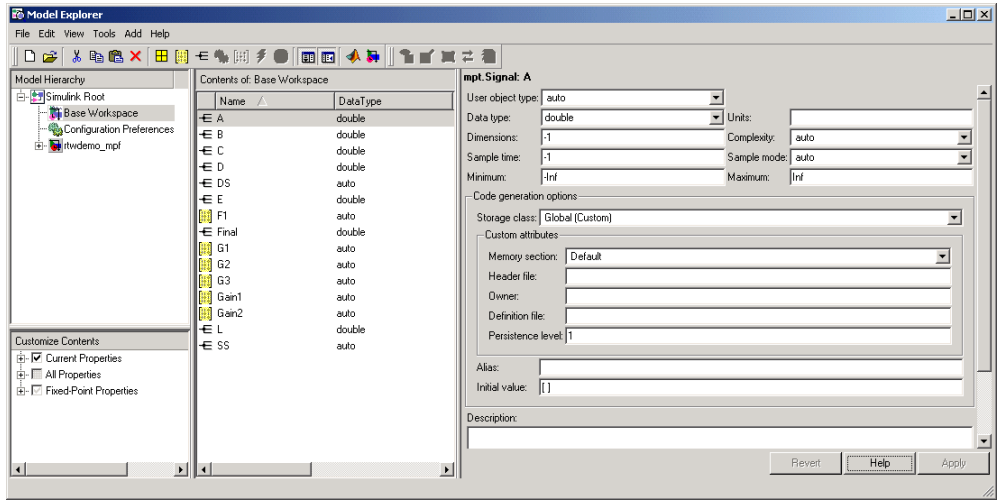
In summary, the different custom storage classes on the Model Explorer for `mpt` data objects provide control over the appearance of the generated code. The additional custom attributes (owner, definition file, persistence level, memory section) provide more control over data packaging of the generated code. For signals, the initial value property allows you to initialize the generated code. In the Configuration Parameters dialog box for `mpt` data

objects, the custom comments feature allows you to add a comment just above a signal or parameter's identifier in the generated code. With signal display level, you can specify whether or not the code generator declares a signal data object as global data. The parameter tune level allows you to specify whether or not the code generator declares a parameter data object as tunable global data. The source of initial values feature allows you to select whether the model or the data object initializes each of the model's signals in the generated code.

Properties on Model Explorer

The properties that appear on the Model Explorer when `mpt` is the package include all the properties that appear when Simulink is the package plus additional properties. Notice this by comparing the next two figures. (For descriptions of all properties on the Model Explorer, see Parameter and Signal Property Values on page A-20.)





Configuration Parameters Elements

The following elements on the Configuration Parameters dialog box reflect the Real-Time Workshop Embedded Coder module packaging features, when the selected system target is `ert.tlc` (or a system target file derived from an `ert.tlc`):

- **Custom comment (MPT objects only)** on the **Real-Time Workshop Comments** pane
- **Global data placement (MPT data objects only)** on the **Real-Time Workshop Data Placement** pane:
  - **Module naming**
  - **Signal display level**
  - **Parameter tune level**
  - **Source of initial values**

Generated Code

In the example used in "Setting Property Values for the Simulink Data Objects" on page 3-9, you selected `Default (Custom)` in the **Storage class** field for signal `A` and parameter `F1`. You selected the default `Auto` in the

Storage class field for the remaining data objects. But for the `mpt` data objects you used the default `Global (Custom)` in the **Storage class** field for all data objects. When you generated code, these selections resulted in the definitions and declarations shown in the table below.

Simulink Data Object with Auto Storage Class	Simulink Data Object with Default (Custom) Storage Class	mpt Data Object with Global (Custom) Storage Class and Definition File Named mydefinitionfile

In `rtwdemo_mpf.c`:
 	
`/* For signal A */`
`ExternalInputs rtU;`
 	
`/* For parameter F1 */`
`if(rtU.A * 2.0 > 10.0) {...`
 	
In `rtwdemo_mpf.h`:
 	
`/* For signal A */`
`typedef struct {`
`  real_T A;`
`} ExternalInputs;`
 	
`extern ExternalInputs rtU;`
	In `global.c`:
	
	`real_T A;`
	`real_T F1 = 2.0;`
	
	In `global.h`:
	
	`extern real_T A;`
	`extern real_T F1;`
		In `mydefinitionfile.c`:
		
		`real_T A;`
		`real_T F1 = 2.0;`
		
		In `global.h`:
		
		`extern real_T A;`
		`extern real_T F1;`

The results shown in the second and third columns of the preceding table require the following configuration parameter adjustments on the **Real-Time Workshop > Data Placement** pane:

- Set **Data definition** to `Data defined in single separate source file`.
- Set **Data definition filename** to `global.c`
- Set **Data declaration** to `Data declared in single separate source file`.
- Set **Data definition filename** to `global.h`

See the left column of the table, which shows generated code for Simulink signal and parameter data objects, whose **Storage class** field is `Auto`. The input A is defined as part of the structure `rtU` as shown above. In the case of the Simulink parameter data object `F1`, since the **StorageClass** was set to `auto`, the code generator chose to include the literal value of `F1` in the generated code. `F1` is a constant in the Stateflow diagram whose value is initialized as `2.0`:

```
if(rtU.A * 2.0 > 10.0) { ...
```

For more details, see "Introduction to Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation and "Summary of Signal Storage Class Options" in the Real-Time Workshop documentation.

See the middle column of the table. The Simulink data objects whose **Storage class** is not `Auto` are defined in a definition statement in the global source file (`global.c`) and declared in a declaration statement in the global header file (`global.h`).

In the right column, Simulink data objects whose **Storage class** is not `Auto` are defined in `mydefinitionfile`, as you specified. The declarations for those objects are in the global header file.

Creating Data Objects Based on an External Data Dictionary

This procedure creates data objects based on an external data dictionary (such as an Excel file). You can do this manually (that is, one-by-one) or automatically (all at once).

Manually Creating Objects to Represent External Data

You can create data objects (and their properties) one-by-one, based on an external data dictionary, as follows:

1. Open the external file that contains the data (such as a spreadsheet or database file).

2. Determine all of the data in this file that correspond to the parameters and signals in the model. In the code generation data dictionary, parameters

in the external file belong to the Simulink parameter class and signals belong to the Simulink signal class.

3 On the MATLAB command line, type `daexplr` and press **Enter**. The Model Explorer appears.

4 On the **Model Hierarchy** (left) pane, expand **Simulink Root**, and select **Base Workspace**.

5 On the **Add** menu, select **MPT Parameter** or **Simulink Parameter**. The default name `Param` appears in the **Contents of** (middle) pane.

6 Double-click `Param` and rename this data object as desired.

7 Repeat steps 5 and 6 for each additional data item in the external file that belongs to the `mpt.Parameter` class or `Simulink.Parameter` class.

Now you will add data items in the external file that belong to the `mpt.Signal` class or `Simulink.Signal` class.

8 On the **Add** menu, select **MPT Signal** or **Simulink Signal**. The default name `Sig` appears in the **Contents of** pane.

9 Double-click `Sig` and rename the data object as desired.

10 Repeat steps 8 and 9 for each additional data item in the external file that belongs to the `mpt.Signal` class or `Simulink.Signal` class.

All external data items for the `mpt.Parameter` or `Simulink.Parameter` class, and the `mpt.Signal` or `Simulink.Signal` class now appear in the **Contents of** pane and in the MATLAB workspace. Therefore, they have been created in the code generation data dictionary.

Note The property *values* for these data objects are supplied by default.

Automatically Creating Objects to Represent External Data

You can create data objects (and their properties) all at once, based on an external data dictionary by creating and running a `.m` file. This file contains the same MATLAB commands you could use for creating data objects

one-by-one on the command line, as explained in "Working with Data Objects" in the Simulink documentation. But instead of using the command line, you place the MATLAB commands in the `.m` file for *all* of the desired data in the external file:

1. Create a new `.m` file.
2. Place information in the file that describes all of the data in the external file that you want to be data objects. For example, the following information creates two `mpt` data objects with the indicated properties. The first is for a parameter and the second is for a signal:

```
% Parameters
mptParCon = mpt.Parameter;
mptParCon.RTWInfo.CustomStorageClass ='Const';
mptParCon.value = 3;
% Signals
mptSigGlb = mpt.Signal;
mptSigGlb.DataType = 'int8';
```

3. Run the `.m` file. The data objects appear in the MATLAB workspace.

Note If you want to import data from an external data dictionary, you can write functions that read the information, convert these to data objects, and load them into the MATLAB workspace. Among available MATLAB functions that you can use for this process are `xmlread`, `xmlwrite`, `xlsread`, `xlswrite`, `csvread`, `csvwrite`, `dlmread`, and `dlmwrite`.

Saving and Loading Data Objects

In a `.mat` file, you can save the set of data objects (and their properties) that you have created and load this information for later use or exchange it with another user. You can save some of the data objects in the workspace or all of them. See Opening, Loading, Saving Files in the MATLAB documentation.

Applying Naming Rules to Identifiers Globally

Note This feature applies both to Simulink and `mpt` data objects.

Signal and parameter names appear on the model. The same names appear as data objects on the Model Explorer. By default, these names are replicated exactly in the generated code. For example, `"Speed"` on the model (and workspace) appears as the identifier `"Speed"`in the code, by default. But you can change how they appear in the code. For example, if desired, you can change `"Speed"` to `SPEED` or `speed`. Or, you can choose to use a different name altogether in the generated code, like `MPH`. The only restriction is that you follow ANSI C/C++ rules for naming identifiers.

There are two ways of changing how a signal name or parameter name is represented in the generated code. You can do this *globally*, by following this procedure. This procedure makes selections on the Configuration Parameters dialog box to change *all* of the names when code generation occurs, according to the same rule. Or, you can change the names *individually* by following "Setting Property Values for the Simulink Data Objects" on page 3-9. The relevant field in that procedure is **Alias** on the Model Explorer.

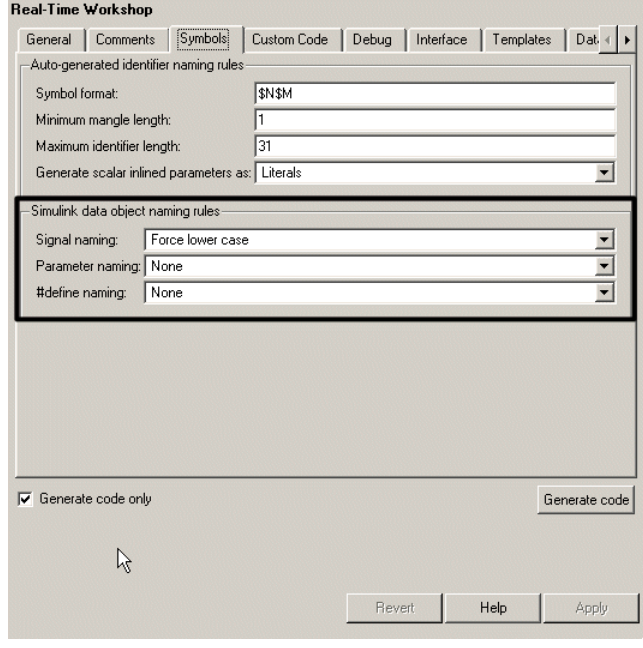
If the Alias field is empty, the naming rule that you select on the Configuration Parameters dialog box applies to all data objects. But if you do specify a name in the Alias field, this overrides the naming rule for that data object. The table below illustrates these cases. The table assumes that you selected `Force lower case` as the naming rule. But with the information provided, you can determine how any of the naming rules works for an `mpt` data object or a Simulink data object (`Force upper case`, `Force lower case`, or `Custom M-function`).

Naming Rules and Alias Override (Global Change of Force Lower Case Rule)

Name of Data Object in Model	Name in Alias Field	Package	Result in Generated Code
A		Simulink or mpt	a
A	D	Simulink or mpt	D

You specify data object naming rules on the **Real-Time Workshop > Symbols** pane of the Configuration Parameters dialog box. To access that pane,

1. Open your model.
2. Open the Configuration Parameters dialog box from the **Simulation** menu or Model Explorer.
3. Open the **Real-Time Workshop > Symbols** pane. See the section **Simulink data object naming rules**.



Notice the preconfigured settings on this pane. If all of these are acceptable as is, proceed to "Creating User Data Types" on page 3-25. Otherwise, follow the procedures below, as desired, to change parameter names, signal names, or parameter names you want to use in a `#define` preprocessor directive. "MPF Panes on the Configuration Parameters Dialog Box" on page A-2 describes all fields on this pane and their possible settings for these procedures.

- "Defining Rules That Change All #defines" on page 3-22
- "Defining Rules That Change All Parameter Names" on page 3-23
- "Defining Rules That Change All Signal Names" on page 3-24

Defining Rules That Change All #defines

This procedure allows you to change all of the model's parameter names whose storage class you selected as `Define` in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13, using the same rule. The new names will appear as identifiers in the generated code:

1. In the **#define naming** field, click the desired selection. ("MPF Panes on the Configuration Parameters Dialog Box" on page A-2, explains the possible selections, under the **Symbols** pane.) The default is `None`. If you select `Custom M-function`, go to the next step. Otherwise, click **Apply** and proceed to "Defining Rules That Change All Parameter Names" on page 3-23.

2. Write a function in M-code that changes all occurrences of the parameter name whose storage class you specified as `Define` in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13 so that it appears the way you want as an identifier in the generated code. (An example is shown below.)

3. Save the function as a `.m` file in any folder that is in the MATLAB path.

4. In the **M-function** field under **#define naming**, type the name of the file you saved in the previous step.

5. Click **Apply** and then define rules that change all parameter names.

Defining Rules That Change All Parameter Names

This procedure allows you to change all of the model's parameter names, using the same rule. The new names will appear as identifiers in the generated code:

1. In the **Parameter naming** field, click the desired selection. ("MPF Panes on the Configuration Parameters Dialog Box" on page A-2, explains the possible selections, under the **Symbols** pane.) The default is `None`. If you selected `Custom M-function`, go to the next step. Otherwise, click **Apply**, and proceed to "Defining Rules That Change All Signal Names" on page 3-24.

2. Write a function in M-code that changes all occurrences of parameter names in the model to appear the way you want as identifiers in the generated code. For example, the code below changes all parameter names as necessary to make their first letter uppercase, and their remaining letters lowercase.

```
function
revisedName = initial_caps_only(name, object)
% INITIAL_CAPS_ONLY: User-defined naming rule causing each
% identifier in the generated code to have initial cap(s).
%
% name: name as spelled in model.
% object: the object of name; includes name's properties.
%
% revisedName: manipulated name returned to MPT for the
code.
%
%
:
revisedName = [upper(name(1)),lower(name(2:end))];
:
```

3. Save the function as a `.m` file in any folder that is in the MATLAB path.

4. In the **M-function** field under **Parameter naming**, type the name of the file you saved in the previous step.

5. Click **Apply** and then define rules that apply to all signal names.

Defining Rules That Change All Signal Names

This procedure allows you to change all of the model's signal names, using the same rule. The new names will appear as identifiers in the generated code:

1. On the **Signal naming** field, click the desired selection. (MPF Elements on Configuration Parameters Panes on page A-2, explains the possible selections, under **Symbols** pane.) The default is `None`. If you selected `Custom M-function`, go to the next step. Otherwise, click **Apply** and then generate and inspect code.

2. Write a function in M-code that changes all occurrences of signal names in the model to appear the way you want as identifiers in the generated code. (An example is shown in "Defining Rules That Change All Parameter Names" on page 3-23.)

3. Save the function as a `.m` file in any folder that is in the MATLAB path.

4. In the **M-function** field under **Signal naming**, type the name of the file you saved in the previous step.

5. Click **Apply** and then generate and inspect code.

Creating User Data Types

Note This feature applies both to Simulink and `mpt` data objects.

By default, MathWorks data types (such as `real32_T` and `uint8_T`) are used to define data in the generated code. Instead, if you prefer using your company-standard data types (such as `DBL` and `U8`), you can create user-defined data types. To use this feature, you must register your data types so that the code generator can associate them with the corresponding MathWorks C/C++ data types. Then, the code generator will use your user data types in the generated code instead of the MathWorks C/C++ data types.

You can register user data types by making calls to the appropriate API functions in the `custom_user_type_registration.m` file, whose arguments specify information about the user data type and its associated MathWorks C/C++ data type. This procedure explains how to register user data types:

Note Real-Time Workshop automatically associates the MathWorks C/C++ data types with the equivalent ANSI C/C++ data types.

1 Observe the default MathWorks C/C++ data types listed below:

- `boolean_T`
- `int8_T`
- `int16_T`
- `int32_T`
- `real32_T`
- `real_T`
- `uint8_T`
- `uint16_T`
- `uint32_T`

If you want to use only these, then you do not need to register your own data types. In this case, proceed to “Registering User Object Types” on page 3-38. Otherwise, go to the next step.

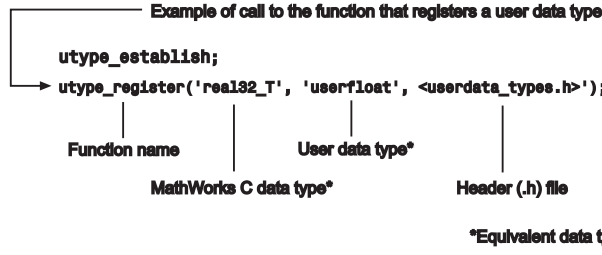
2 If desired, make a copy of Equivalent Data Types Worksheet on page 3-29 and, in the user-defined data types column, list all of your data types. This table serves as a convenient reference for performing the next steps. You can associate as many user-defined data types with a single MathWorks C/C++ data type as you want.

3 On the MATLAB command line, type

```
edit custom_user_type_registration
```

The preexisting file named `custom_user_type_registration.m` opens in the MATLAB editor. The code generator executes this file's code to register user-defined data types, only if these have been added to the file. Initially, this file contains no user-defined data type information.

In `custom_user_type_registration.m`, you must register every user data type to be added to the dictionary by a call to the M-function `utype_register`. You must provide the arguments in this function call in the prescribed left-to-right order and syntax. See the figure below for an example and Arguments to Place in utype_register Function Call on page 3-29 for an explanation of each argument.



4 Type a `utype_register` function call for every user data type that you listed in Equivalent Data Types Worksheet on page 3-29.

5 Ensure that the header file, referenced in the fourth argument of `utype_register`, contains a `typedef` statement for each user data type. Note that the base type of each user data type is required to match the base type of the MathWorks data type it is intended to replace. (For comparison, the `typedef` statements for the MathWorks data types can be found in the generated file `rtwtypes.h`.) For example, to define user data types `f32` and `U8` to replace MathWorks data types `real32_T` and `uinit8_T`, use:

```
typedef float f32;
typedef unsigned char U8;
```

Caution Failing to match the base type of each user data type to the base type of the corresponding MathWorks data type can lead to compiler warnings, errors, and incorrect numerical results.

---

6 Ensure that the header file is in the appropriate directory. (That is, appropriate to the chosen `#include` delimiter: angle-brackets or double-quotation marks.)

Caution During code generation, a consistency checker looks for the required filename `custom_user_type_registration.m`. So do not change the name of this file when doing the next step.

---

7 Save the `custom_user_type_registration.m` file outside of the MATLAB installation directory in a work folder on the MATLAB path. The path must be above `toolbox/rtw/targets/mpt/user_specific` in the MATLAB search path.

Note Step 8 automatically creates the `Simulink.AliasType` objects that correspond to the registered data types. Alternatively, you can create a `Simulink.AliasType` object one at a time using the **Simulink Alias Type** selection on the **Add** menu of the Model Explorer. Or, you can create a `Simulink.AliasType` object one at a time by typing `userdatatype = Simulink.AliasType` on the MATLAB command line, where `userdatatype` is a registered user data type. See `Simulink.AliasType` in the Simulink documentation. Note also that if there are data type replacements specified in the **Data type replacement** pane of the Configuration Parameters dialog box for the model, the next step also will create `Simulink.AliasType` objects from these. For details, see "Replacing Built-In Data Type Names in Generated Code" on page 3-44.

8 Create all `Simulink.AliasType` objects by typing the MATLAB command, `ec_create_type_obj`. These `Simulink.AliasType` objects, which correspond to registered user data types, appear in the MATLAB base workspace. Now they are registered and you can use them in a model.

Proceed to "Selecting User Data Types for Signals and Parameters" on page 3-31.

Equivalent Data Types Worksheet

MathWorks C/C++ Data Type	User-Defined Data Types
`boolean_T`	
`int8_T`	
`int16_T`	
`int32_T`	
`real32_T`	
`real_T`	
`uint8_T`	
`uint16_T`	
`uint32_T`	

Arguments to Place in utype_register Function Call

Argument (Left to Right) to Place in utype_register Function Calls	Description
MathWorks C/C++ Data Type	The MathWorks C/C++ data type, listed in Equivalent Data Types Worksheet on page 3-29, that is equivalent to the specified user data type.
User Data Type	The user-defined data type that is equivalent to the MathWorks C/C++ data type specified in the function call. The name must conform to ANSI C/C++ rules and the base type must match the base type of the corresponding MathWorks data type. You can add a fifth argument in the `utype_register` function call to specify that the user-defined data type can be used with a signal, a parameter, or both. To do so, after the header file delimiter argument, add `'Signal'`, `'Parameter'`, or `'Both'`. Omitting a fifth argument is interpreted as `'Both'`.
Header file	Specifies the user's existing header file. For the generated code to compile, you must account for this header file with an `#include` in the appropriate

Arguments to Place in utype_register Function Call (Continued)

Argument (Left to Right) to Place in utype_register Function Calls	Description
	generated file. Also, this header file must contain the required `typedef` statements for the registered user data types. Place the filename between the ANSI C/C++ angle-brackets or double-quotes delimiters. The directory that each delimiter indicates varies by compiler and is explained in the compiler documentation. Usually (but not always), the angle brackets indicate that the compiler will find the `.h` file in the default directory where the standard C/C++ library is located (the "source" folder).

Selecting User Data Types for Signals and Parameters

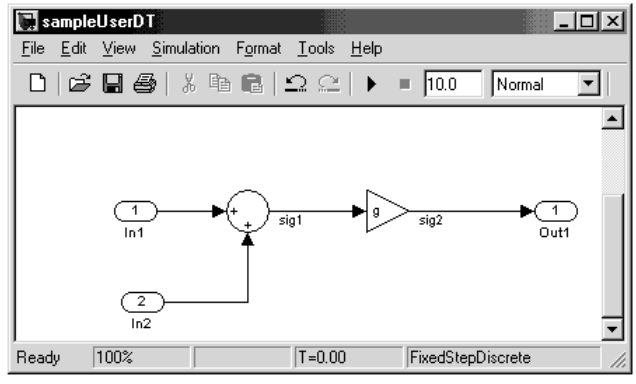
This procedure assumes you followed “Creating User Data Types” on page 3-25, in which you

- Registered user data type `f32`, associating it with MathWorks C/C++ data type `real32_T`
- Specified `<userdata_types.h>` as the `#include.h` file for `f32` in `custom_user_type_registration.m`.

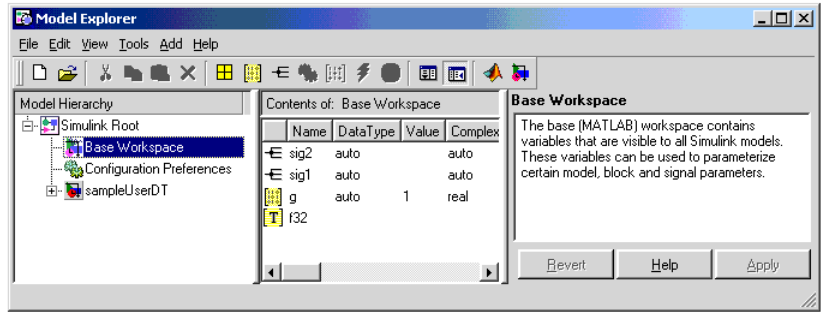
An example of the `typedef` statement you placed in the `#include` file, `userdata_types.h`, is

```
typedef float f32;
```

Open a model and create a data dictionary. (See “Creating Simulink and mpt Data Objects” on page 3-5.) The example model used in this procedure and its resulting Model Explorer, are shown below. The three data objects are signals `sig1` and `sig2`, and parameter `g`. The registered, user-defined data type, `f32`, appears in the middle pane. The "**T**" indicates `f32` is an alias data type.



The Model Explorer for the preceding model looks like this:



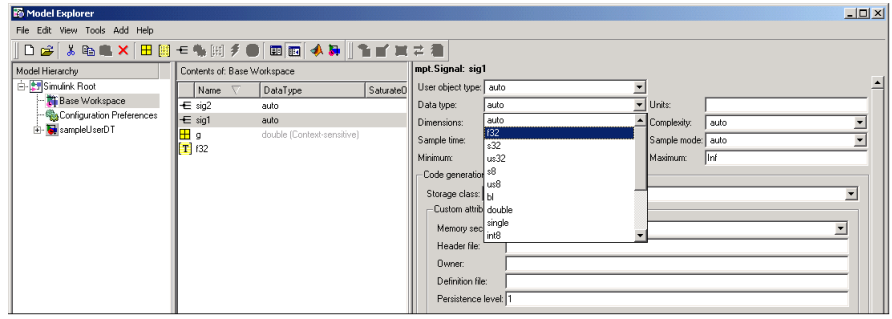
Proceed to the following, as applicable:

- “Selecting User Data Types for Simulink Signals” on page 3-32
- “Selecting User Data Types for Simulink Parameters” on page 3-35

Selecting User Data Types for Simulink Signals

This procedure explains how to use user-defined data types for Simulink signals and for their corresponding identifiers in the generated code. You can use user-defined data types with signals whether or not they have Simulink signal objects associated with them.

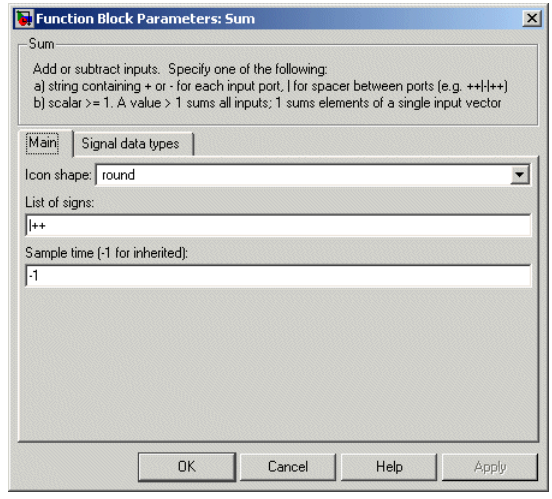
1 For an *mpt* signal object that is associated with a signal in your model, select the desired user data type in the **Data type** field. For example, select `f32`, for `sig1`, as shown below.



This selects `f32` for the `sig1` data object in the data dictionary, but does not select `f32` for the corresponding labeled signal in the model. So the two may be in conflict. If you tried to update the model, you could get an error message.

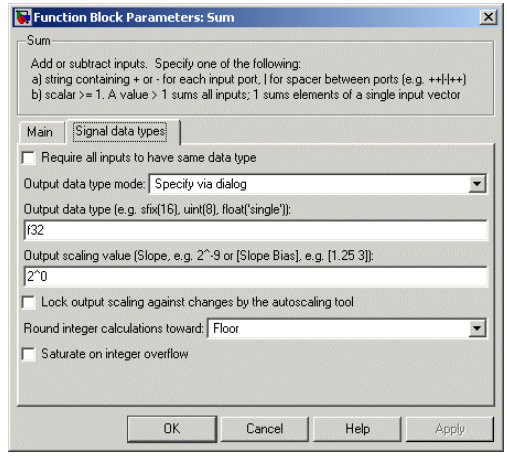
For a *Simulink* signal object, type `f32` into the **Data type** field.

2 Select the model and double-click the signal's source block. (The source block of a model signal controls the signal's data type.) For example, since the source block for `sig1` is the Sum block, double-click the Sum block. The Function Block Parameters dialog box appears.

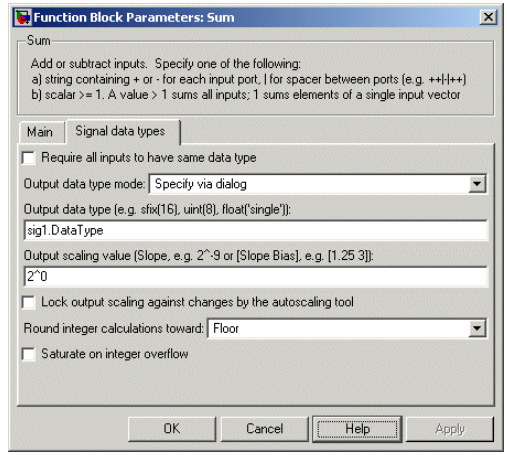


3 Select the **Signal data types** tab. In the **Output data type mode** field, select `Specify via dialog`. The dialog expands.

4 As shown below, in the **Output data type** field, type the same user data type name that you selected for the data object (step 1), and then click **Apply**. The user data type of the signal in the model and that of the signal object now are the same.



Instead of specifying a specific data type (step 4), you could do what is termed dictionary-driven data typing: In the **Output data type** field, specify `object.DataType`, where `object` is the case-sensitive object name. For example, `sig1.DataType`, as shown below.



The advantage of referencing like this is that subsequent changing of the user data type in the dictionary for this object automatically changes the user data type of the corresponding model signal.

5 Repeat steps 1 through 5 for each remaining model signal that has a corresponding signal object for which you selected a user data type.

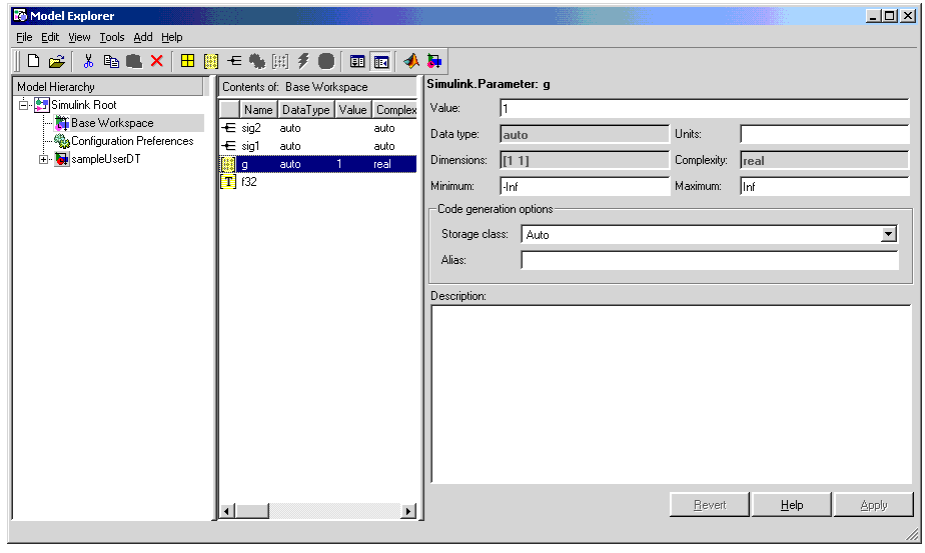
6 Save the model and save all of the data objects in the MATLAB base workspace in a `.mat` file for reuse later. After generating code, you would see the following code fragment for the example model `sampleUserDT.mdl` (that has the default MPF settings):

In `sampleUserDT.c`: `f32 sig1;`

In `sampleUserDT_types.h`: `#include <userdata_types.h>`

Selecting User Data Types for Simulink Parameters

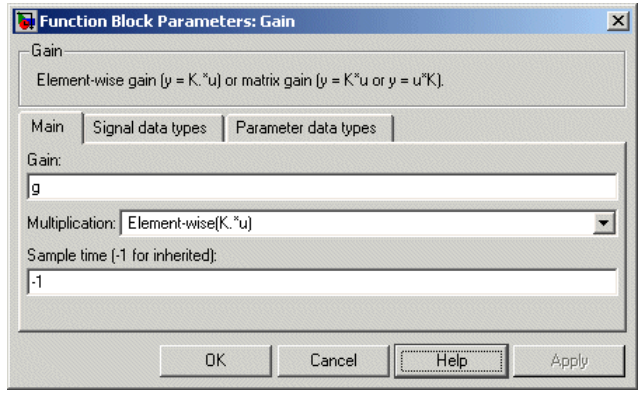
This procedure explains how to use a registered, user-defined data type as the data type for a Simulink parameter and for its corresponding identifier in the generated code. Unlike `mpt` signal objects, described in "Selecting User Data Types for Simulink Signals" on page 3-32, registered user data types cannot be used directly with Simulink parameter objects, as shown below.



Note that `f32` appears in the middle pane and thus is a registered user data type. But it is not available for selection in the right pane.) Instead, for some blocks (like Gain), you can specify the user data type by double-clicking the block and using the **Parameter data types** tab on the **Function Block Parameters** dialog box.

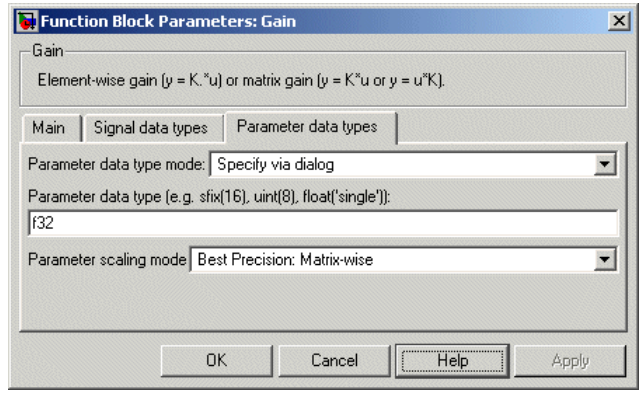
The steps below illustrate this method. However, for many blocks, the **Parameter data types** tab is not available. In these cases, the data type of the block's input or output signal determines the block's parameter data type.

1 Double-click the parameter's source block. For example, the source block for `g` is the Gain block. The Function Block Parameters dialog box appears.



2 Select the **Parameter data types** tab. In the **Parameter data type mode** field, select `Specify via dialog`. The dialog expands.

3 As shown below, in the **Parameter data type** field, type the desired, registered user data type name, and click **Apply**.



4 Repeat steps 1 through 4 for each remaining Simulink parameter for which you want to specify a registered user data type.

5 Save the model and save all of the data objects in a MATLAB base workspace in a `.mat` file for reuse later. After generating code, you would see the following code fragment for the example model `sampleUserDT.mdl`:

In `sampleUserDT.c`:

```
real32_T g = 1.0F;
```

In `sampleUserDT_types.h`:

```
#include <userdata_types.h>
```

Registering User Object Types

Note This feature applies only to `mpt` data objects.

This procedure registers one or more fixed sets of user-defined property values that can be applied to `mpt` data objects. Each set, called a "user object type," is given a unique name. Use this feature when you want to apply the same set of properties and their values to multiple data objects. You can apply the predefined set automatically to selected data objects, rather than having to follow “Creating mpt Data Objects, Setting Property Values, and Generating Code” on page 3-13 for each of these data objects.

You register the set of property values by placing code in a file named `custom_user_object_type_info.m`. The table below explains the content of this file.

Content of custom_user_object_type_info.m

Item	Description	Required (R) or Optional (O) Field in Structure
`custom_user_object_type_info`	A function that specifies user object type information	N/A
`objectType`	A cell array, each element of which is a structure for a user object type. The fields of the structure specify all of the desired information about this user object type.	N/A
Field names in each structure:	Corresponding element on the Model Explorer, if applicable:	N/A
`Name`	Name of object type	R

Content of custom_user_object_type_info.m (Continued)

Item	Description	Required (R) or Optional (O) Field in Structure
`Type`	Specifies that you can apply the data object type name (specified for `Name`, above) only to signals (`'Signal'`), only to a parameters (`'Parameter'`), or to both a signals and parameters (`'Both'`).	R
`DataType`	Data type of the `mpt` data object. This can be one of the user-defined data types or a MathWorks data type (`boolean, int8, int16, int32, single, double, uint8, uint16, uint32`). **Data type** on the Model Explorer for an `mpt.Signal` data object. **Value** on the Model Explorer for an `mpt.Parameter` data object.	R
`Units`	**Units**	O
`Owner`	**Owner**	O
`Level`	**Persistence level**	O
`InitialValue`	**Initial value**	O
`Value`	**Value**	O
`Max`	**Maximum**	O
`Min`	**Minimum**	O
`DefinitionFile`	**Definition**	O
`Description`	**Description**	O

Content of custom_user_object_type_info.m (Continued)

Item	Description	Required (R) or Optional (O) Field in Structure
HeaderFile	**Header file**	O
CustomStorageClass	**Storage class**	O

You can register as many user object types in the file as you want. For example, the following code registers three, `EngineSpeedType` (used only for signals), `CalType` (used only for parameters), and `AirFlowInSpeedType` (used for both):

```
function objectType = custom_user_object_type_info
objectType = '';
%
%Registering EngineSpeedType object type for signals
objectType{end+1}.Name = 'EngineSpeedType';
objectType{end}.Type = 'Signal';
objectType{end}.DataType = 'S32';
objectType{end}.Units = 'rmp';
objectType{end}.DefinitionFile = 'EngSpeedDef.c';
objectType{end}.Description = 'This is Engine Speed Type';
objectType{end}.CustomStorageClass = 'Global';
objectType{end}.Level = 5;
%
%Registering CalType object type for parameters
objectType{end+1}.Name = 'CalType';
objectType{end}.Type = 'Parameter';
objectType{end}.DataType = 'single';
objectType{end}.Units = ' ';
objectType{end}.CustomStorageClass = 'Const';
objectType{end}.HeaderFile = 'CalibrationInclude.h';
objectType{end}.Value = 5;
%
```

```
%Registering AirFlowInSpeedType for parameters and signals
objectType{end+1}.Name = 'AirFlowInSpeedType';
objectType{end}.Type = 'Both';
objectType{end}.DataType = 'int32';
objectType{end}.Units = 'mm^3/s';
objectType{end}.Min = -2^12;
objectType{end}.Max = 2^12;
%
```

When you register at least one user object type, the User object type field appears on the Model Explorer. (Otherwise, this field is unavailable.) The unique names of the registered user object types are selectable in the **User object type** field. The property values you specify in the file for a particular user object type name appear automatically in the corresponding fields on the Model Explorer when you select that name.

Never change the following two lines in the file:

```
function objectType = custom_user_object_type_info
objectType = {};
```

As indicated in Content of custom_user_object_type_info.m on page 3-38, when you register a user object type name, you must specify values for its `Name`, `Type`, and `DataType` structure fields. Specifying values for the remaining fields is optional. If you do specify values in any of these remaining fields, they automatically populate the corresponding elements on the Model Explorer. Otherwise, default values appear in these elements.

The `custom_user_object_type_info.m` file must be on the MATLAB path:

1. Using a text editor, create a file named `custom_user_object_type_info.m`. Note that an example `custom_user_object_type_info.m` file that you can modify is available in the `matlab/toolbox/rtw/targets/mpt/user_specific` directory.
2. In the file, register a unique user object type name and the desired set of properties and property values you want associated with this name.
3. Repeat the preceding step to register any additional user object type or types.

4 Save the file on the MATLAB path.

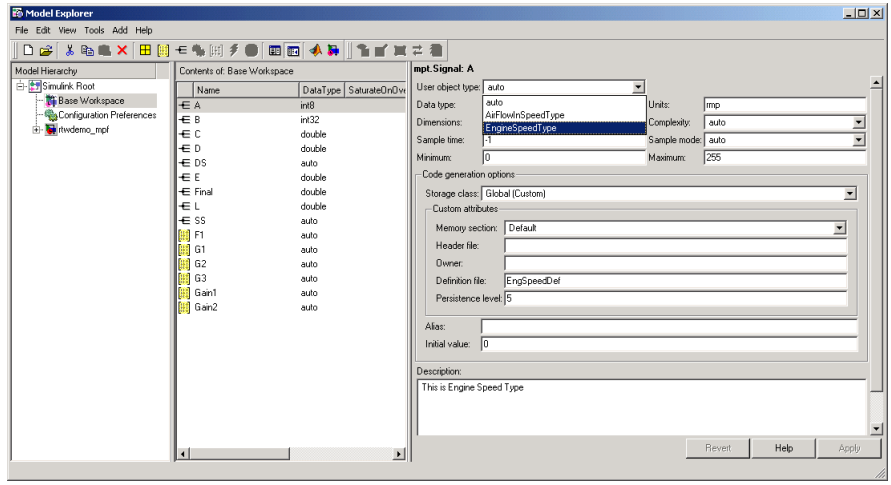
5 Restart MATLAB.

6 In an open model, create `mpt` data objects. See “Creating Simulink and mpt Data Objects” on page 3-5.

7 Open the Model Explorer, and select a signal data object in the middle pane.

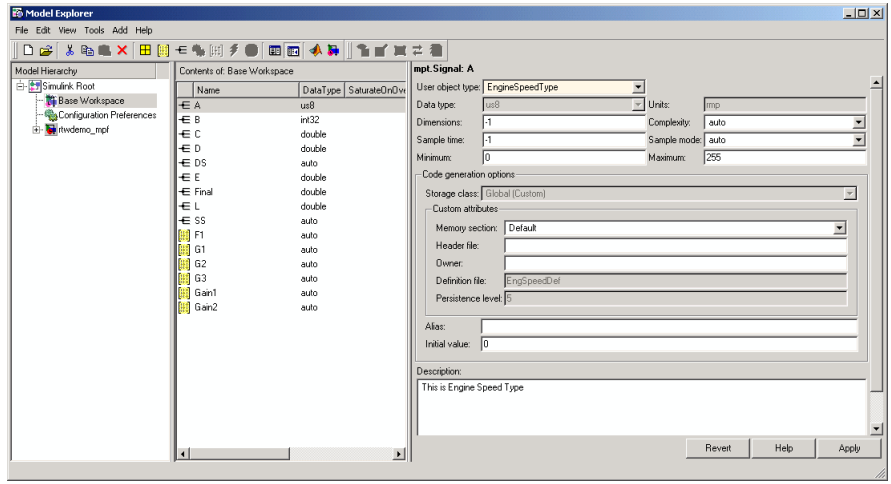
8 On the right pane, click the down arrow beside the **User object type** field.

Notice in the example below that the registered `AirFlowInSpeedType` and `EngineSpeedType` are selections, as expected. `EngineSpeedType` appears because you specified its `Type` as `'Signal'` in `custom_user_object_type_info.m`. `AirFlowInSpeedType` appears because you specified its `Type` as `'Both'`. (If you select a parameter data object in the middle pane, `AirFlowInSpeedType` and `CalType` appear instead.)



Note For any new selections to be available, you must restart MATLAB.

9 Select `EngineSpeedType`. The property values that you specified in `custom_user_object_type_info.m` for `EngineSpeedType` populate the right pane, as shown below.

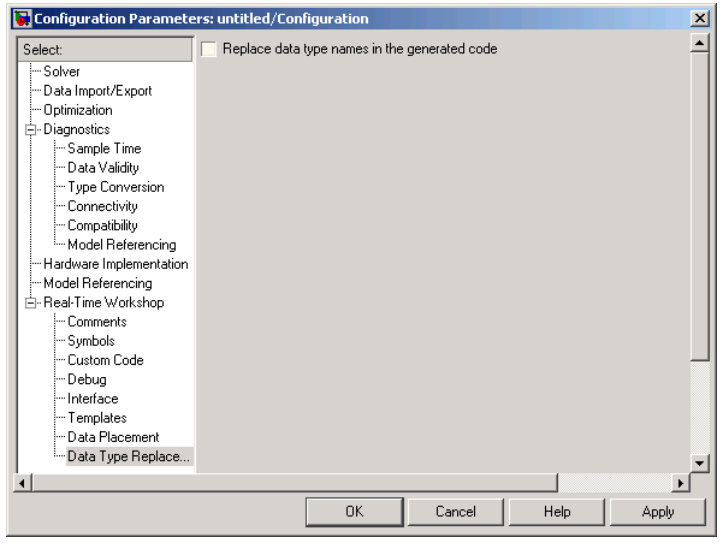


10 Click **Apply**, and generate code.

Note To apply the property values of a user object type to more than one data object simultaneously, you must use the middle pane: Select the data objects using the **Ctrl** or **Shift** key, scroll to the **UserObjectType** column, click any selected row and select the desired user object type.

Replacing Built-In Data Type Names in Generated Code

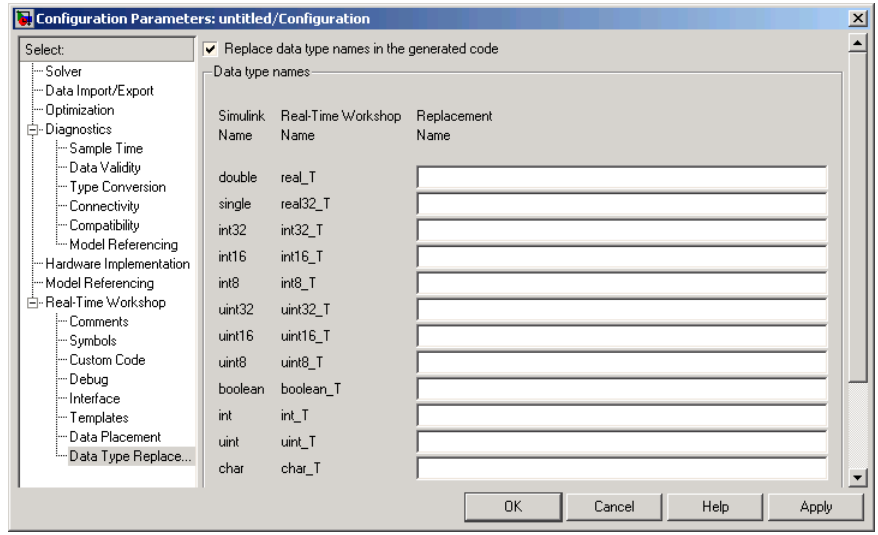
If your application requires you to replace built-in data type names with user-defined replacement data type names in the generated code, you can do so from the **Real-Time Workshop > Data Type Replacement** pane of the Configuration Parameters dialog shown below.



This pane is available for ERT target based Simulink models. In addition to providing a mechanism for mapping built-in data types to user-defined replacement data types, this feature:

- Performs consistency checks to ensure that your specified data type replacements are consistent with your model's data types.
- Allows many-to-one data type replacement, the ability to map multiple built-in data types to one replacement data type in generated C code. (Many-to-one data type replacement is not supported for C++ code generation.) For example, built-in data types `uint8` and `boolean` could both be replaced in your generated C code by a data type `U8` that you have previously defined.

If you select **Replace data type names in the generated code**, the **Data type names** table is displayed:



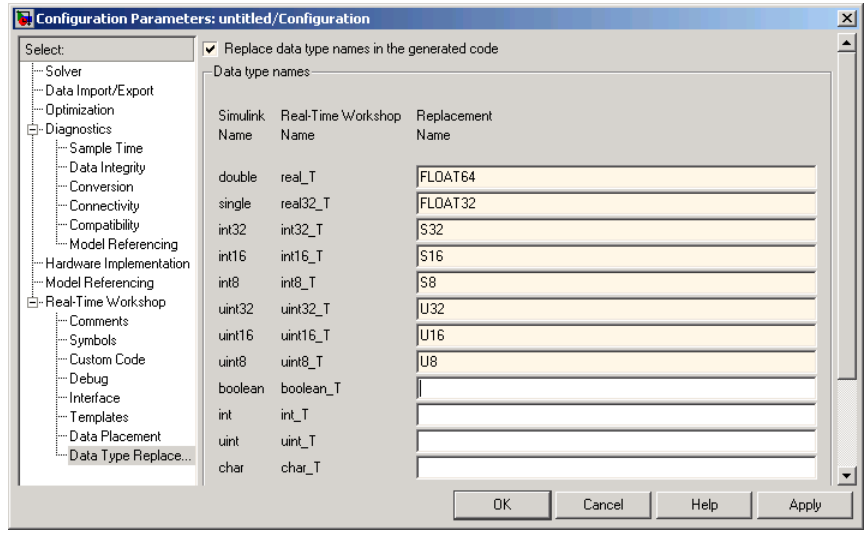
The table **Data type names** lists each Simulink built-in data type name along with its Real-Time Workshop data type name. Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

For each replacement data type entered, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces. For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, and `boolean`, the replacement data type's `BaseType` must match the built-in data type. For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for `int` or `char` on the **Hardware Implementation** pane of the Configuration Parameters dialog. An error occurs if a replacement data type specification is inconsistent.

For example, suppose that you have previously defined the following replacement data types, which exist as `Simulink.AliasType` objects in the base workspace:

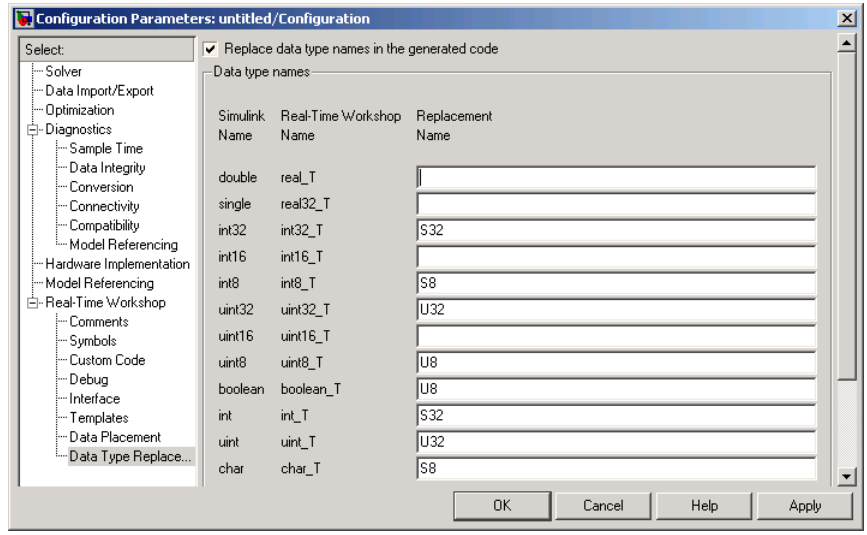
User-Defined Name	Description
FLOAT64	64-bit floating point
FLOAT32	32-bit floating point
S32	32-bit integer
S16	16-bit integer
S8	8-bit integer
U32	Unsigned 32-bit integer
U16	Unsigned 16-bit integer
U8	Unsigned 8-bit integer

You can fill in the **Data Type Replacement** pane with a one-to-one replacement mapping, as follows:



If you are generating C code, you can also apply a many-to-one data type replacement mapping. For example, in the following display:

- `int32` and `int` are replaced with user type `S32`
- `int8` and `char` are replaced with user type `S8`
- `uint32` and `uint` are replaced with user type `U32`
- `uint8` and `boolean` are replaced with user type `U8`



The user-defined replacement types you specify will appear in your model's generated code in place of the corresponding built-in data types. For example, if you specify user-defined data type `FLOAT64` to replace built-in data type `real_T` (`double`), then the original generated code shown in Generated Code with real_T Built-In Data Type on page 3-48 will become the modified generated code shown in Generated Code with FLOAT64 Replacement Data Type on page 3-48.

Generated Code with real_T Built-In Data Type

```
...
/* Model initialize function */
void sinwave_initialize(boolean_T firstTime)
{
 if (firstTime) {
...
    {real_T *dwork_ptr = (real_T *) &sinwave_DWork.lastSin;
...
}
...
```

Generated Code with FLOAT64 Replacement Data Type

```
...
/* Model initialize function */
void sinwave_initialize(boolean_T firstTime)
{
 if (firstTime) {
...
    {FLOAT64 *dwork_ptr = (FLOAT64 *) &sinwave_DWork.lastSin;
...
}
...
```

4

Customizing with Additional Options

This chapter describes the following module packaging features:

Ensuring Delimiter Is Specified for All #Includes (p. 4-2)	Explains how to instruct the code generator to use the angle-bracket delimiter for all data objects whose Header file property has no delimiter specified.
Selecting Source That Initializes Signals (p. 4-3)	Allows you to select the source that initializes each of the model's signals in the generated code.
Adding Custom Comments (p. 4-5)	Explains how to add the selected data object's property values as a comment in the general code above that data object's identifier.
Adding Global Comments (p. 4-7)	Explains how to add a comment to the model so that the comment appears in the generated files where desired.
Selecting Persistence Level for Signals and Parameters (p. 4-12)	Controls the persistence level of signal and parameter objects associated with a model.

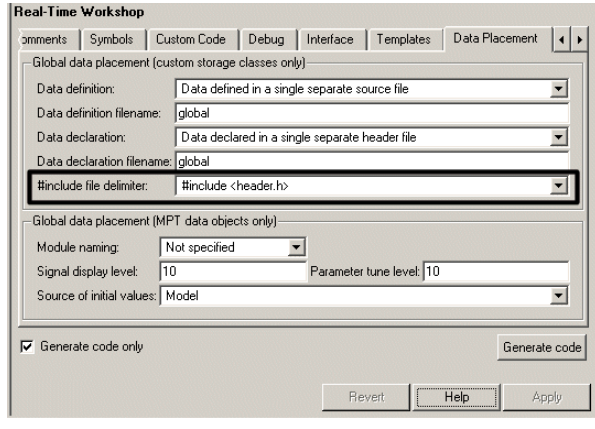
Ensuring Delimiter Is Specified for All #Includes

Understanding the purpose of this procedure requires understanding the `Header file` property of a data object, described in Parameter and Signal Property Values on page A-20, and applied in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13. For a particular data object, you can specify as the `Header file` property value a `.h` filename where that data object will be declared. Then, in the `IncludeFile` section of the generated file, this `.h` file is indicated in a `#include` preprocessor directive.

Further, when specifying the filename as the `Header file` property value, you may or may not place it within the double-quote or angle-bracket delimiter. That is, you can specify it as `filename.h`, `"filename.h"`, or `<filename.h>`. The code generator finds every data object for which you specified a filename as its `Header file` property value *without* a delimiter. By default, it assigns to each of these the double-quote delimiter.

This procedure allows you to specify the angle-bracket delimiter for these instead of the default double-quote delimiter. See the figure below.

1. In the **#include file delimiter** field on the **Data Placement** pane of the Configuration Parameters dialog box, select `#include <header.h>` instead of the default `#include "header.h"`.
2. Click **Apply**.



Selecting Source That Initializes Signals

This procedure allows you to select the source that initializes each of the model's signals in the generated code. This has no effect in model simulation. You make the selection in the **Source of initial values** field on the **Data Placement** pane of the Configuration Parameters dialog box. There are two possible selections: `Model` and `Data object`. The default is `Model`. If you accept `Model`, the Simulink default data initialization controls the initialization for each signal.

If you select `Data object`, the initial value of the data object determines the initialization statement. By default, `mpt` signal objects use dynamic initialization in the model initialization function of the generated code. Alternatively, you can choose static initialization in a data definition. See the relevant discussion of dynamic and static data initialization in "Designing Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation.

For dynamic or auto data initialization of custom storage class, the code generator places the appropriate initialization statement in the generated file. The identifiers that correspond to the model's signals then are initialized to the desired values during the initialization phase when the program runs.

For static data initialization of custom storage class, the code generator places the initialization in a definition statement of the generated file, using the following form:

```
datatype identifier[dimension] = {...};
```

Note that some blocks in a model can overwrite this initial value setting, like the output of a constant block, for example.

1. Open the model and the Configuration Parameters dialog box.
2. Open the **Real-Time Workshop > Data Placement** pane.
3. In the **Source of initial values** field, select `Model` or `Data object`.
4. Click **Apply**.

5 If you selected `Data object` in step 3, type the initial value for the desired signal or signals in the **Initial value** field. You must specify a numeric value of data type double. If you specify a non-double, a warning appears on the command line informing you that the non-double value will be converted to a double data type. Also, the value's dimension must be the same as that specified in the **Port dimension** field of the corresponding signal in the model.

Note For a block that stores states in a Dwork vector, such as Unit Delay or Data Store Memory, Real-Time Workshop Embedded Coder uses the initial value specified in the block parameter dialog box in the generated code instead of any initial value you might specify for an `mpt` signal, using the **Source of initial values** option.

---

6 Click **Real-Time Workshop** on the left pane of the Configuration Parameters dialog box, and click **Generate code**.

7 If an error message occurs indicating the dimension is not consistent with port width, return to step 5 and specify the correct dimension. Or, return to step 5 and specify `[]` for the value so that the Simulink default data initialization takes place.

Adding Custom Comments

This procedure allows you to add a comment just above a signal or parameter's identifier in the generated code. This is accomplished using

- A function that you write in M-code or TLC-code and save in a `.m` or `.tlc` file
- The **Custom comments (MPT objects only)** check box on the **Comments** pane of the Configuration Parameters dialog box
- Selecting the `.m` or `.tlc` file in the **Custom comments function** field on the **Comments** pane of the Configuration Parameters dialog box.

You may include at least some or all of the property values for the data object. Each Simulink data object (signal or parameter) has properties, as described in Parameter and Signal Property Values on page A-20. This example comment contains some of the property values for the data object `MAP` as specified on the Model Explorer:

```
/*     DocUnits:       PSI                                    */
/*     Owner:                                                 */
/*     DefinitionFile: specialDef                             */
real_T MAP = 0.0;
```

You can type text in the **Description** field on the Model Explorer for a signal or parameter data object. If you do, and if you select the **Simulink data object descriptions** check box on the **Comments** pane of the Configuration Parameters dialog box, this text will appear beside the signal's or parameter's identifier in the generated code as a comment. This is true whether or not you select the **Custom comments (MPT objects only)** check box discussed in this procedure. For example, typing `Manifold Absolute Pressure` in the **Description** field for the data object `MAP` always will result in the following in the generated code:

```
real_T MAP = 0.0;     /* Manifold Absolute Pressure */
```

1. Write a function in M-code or TLC-code that places comments in the generated files as desired. An example `.m` file named `rtwdemo_comments_mptfun.m` is provided in the `matlab/toolbox/rtw/rtwdemos` directory. This file contains instructions.

The M-code function must have three arguments that correspond to `objectName`, `modelName`, and `request`, respectively. The TLC-code must have three arguments that correspond to `objectRecord`, `modelName`, and `request`, respectively. Note also, in the case of the TLC file, you can use the library function `LibGetSLDataObjectInfo` to get every property value of the data object.

2. Save the function as a `.m` file or a `.tlc` file with the desired filename and place it in any folder in the MATLAB path.

3. Open the model and the Configuration Parameters dialog box.

4. Click **Comments** under **Real-Time Workshop** on the left pane. The **Comments** pane appears on the right.

5. Select the **Custom comments (MPT objects only)** check box.

6. In the **Custom comments function** field, either type the filename of the `.m` file or `.tlc` file you created, or select this filename using the **Browse** button.

7. Click the **Apply** button.

8. Click **Generate Code**.

9. Open the generated files and inspect their content to ensure the comments are what you want.

Adding Global Comments

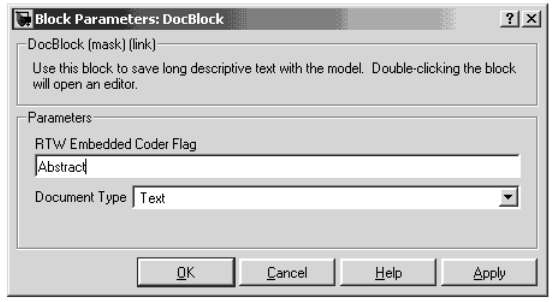
This procedure allows you to add a comment to the model so that the comment appears in the generated file or files where desired. This is accomplished by specifying a template symbol name *with* a Simulink DocBlock, a Simulink Annotation, or a Stateflow Note. For details on template symbols, see "Template Symbols and Rules" on page A-11.

Note Template symbol names `Description` and `ModifiedHistory`, referenced below, also are fields in the Model Properties dialog box. If you use one of these symbol names for global comment text, and its Model Properties field has text in it too, both will appear in the generated files.

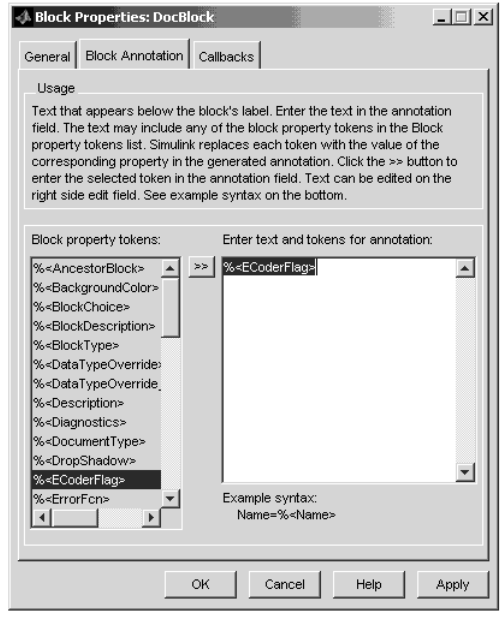
Using a Simulink DocBlock to Add the Comment

1. With the model open, select **Library Browser** from the **View** menu.
2. Drag the DocBlock from **Model-Wide Utilities** in the Simulink library onto the model.
3. After double-clicking the DocBlock and typing the desired comment in the editor, save and close the editor. See DocBlock in the Simulink documentation for details.
4. Right-click the DocBlock and select **Mask Parameters**. The Block Parameters dialog box appears.

5 Type one of the following Documentation child into the **RTW Embedded Coder Flag** field, illustrated below, and then click **OK**: `Abstract`, `Description`, `History`, `ModifiedHistory`, or `Notes`. Template symbol names are case sensitive.



6 In the Block Properties dialog box, select `%<ECoderFlag>` as shown in the figure below, and then click **OK**. The symbol name typed in the previous step now appears under the DocBlock on the model.



7 Save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

8 To add one or more other comments to the generated files, repeat steps 1 through 7 as desired.

Using a Simulink Annotation to Add the Comment

1 Double-click the unoccupied area on the model where you want to place the comment. See "Annotating Diagrams" in the Simulink documentation for details.

Note If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Using Sorted Notes to Add Comments" on page 4-10.

---

2 Type `<S:Symbol_name>` followed by the comment, where `Symbol_name` is one of the following Documentation child : `Abstract`, `Description`, `History`, `ModifiedHistory`, or `Notes`. For example, type `<S:Description>This is the description I want.` Template symbol names are case sensitive. (The `"S"` before the colon indicates "symbol.")

3 Click outside the rectangle and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

4 To add one or more other comments to the generated files, repeat steps 1 through 3 as desired.

Using a Stateflow Note to Add the Comment

1 Right-click the desired unoccupied area on the Stateflow chart where you want to place the comment. See “Using Notes to Extend Chart Diagrams” in the Stateflow documentation for details.

2 Select `Add Note` from the drop down menu.

Note If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with "Using Sorted Notes To Add Comments" below.

---

3 Type `<S:Symbol_name>` followed by the comment, where `Symbol_name` is one of the following Documentation child : `Abstract`, `Description`, `History`, `ModifiedHistory`, or `Notes`. For example, type `<S:Description>This is the description I want.` Template symbol names are case sensitive.

4 Click outside the note and save the model. After you generate code, the code generator places the comment in each generated file whose template has the symbol name you typed. The code generator places the comment in the generated file at the location that corresponds to where the symbol name is located in the template file.

5 To add one or more other comments to the generated files, repeat steps 1 through 4 as desired.

Using Sorted Notes to Add Comments

The sorted-notes capability allows you to add automatically sorted comments to the generated files. The code generator places these comments in each generated file at the location that corresponds to where the `Notes` symbol is located in the template file.

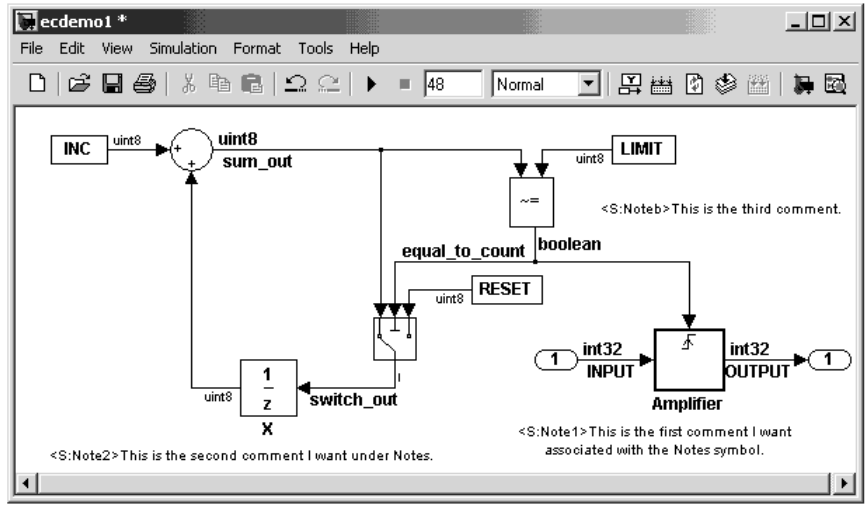
The sorting order the code generator uses is

- Numbers before letters
- Among numbers, 0 is first
- Among letters, uppercase are before lowercase.

You can use sorted notes either with a Simulink annotation or a Stateflow note, but not with a DocBlock:

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment, where `Y` is a number or letter.
- Repeat for as many additional comments you want, except replace `Y` with a subsequent number or letter.

The figure below illustrates sorted notes on a model, and where the code generator places each in a generated file.



Here is the relevant fragment from the generated file for the above model:

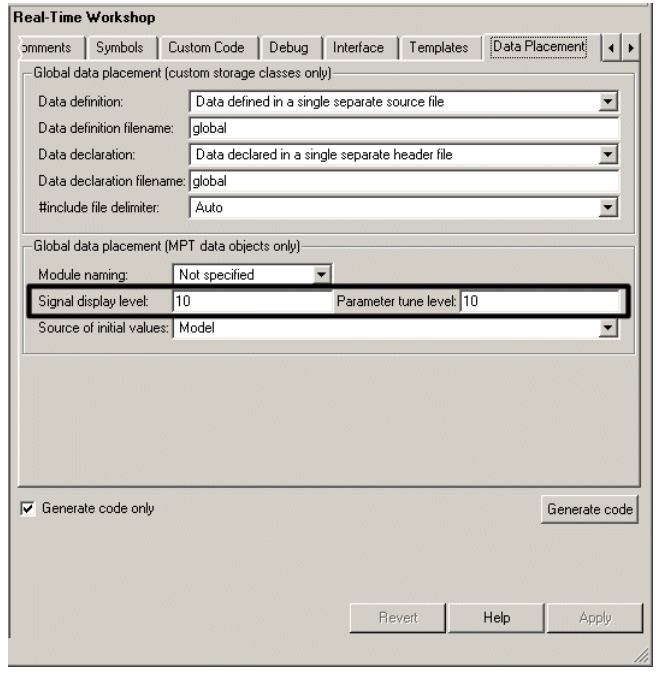
```
** NOTES

** Note1: This is the first comment I want
associated with the Notes symbol.
Note2: This is the second comment I want under Notes.
Noteb: This is the third comment.

**
```

Selecting Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Notice the **Persistence Level** field on the Model Explorer, as illustrated in the figure below. For descriptions of the properties on the Model Explorer, see Parameter and Signal Property Values on page A-20. Notice also the **Signal display level** and **Parameter tune level** fields on the Configuration Parameters dialog box, as illustrated in the next figure.



The **Signal display level** field allows you to specify whether or not the code generator defines a signal data object as global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Signal display level** number is for all `mpt` signal data objects in the model. The **Persistence level** number is for a

particular `mpt` signal data object. If the data object's **Persistence level** is equal to or less than the **Signal display level**, the signal appears in the generated code as global data with all of the properties (custom attributes) specified in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13. For example, this would occur if **Persistence level** is `2` and **Signal display level** is `5`.

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data and thus have none of the custom attributes you specified for that data object. Or, based on expression folding, the code generator could remove the data object so that it does not appear in the code. (See "Tips for Optimizing the Generated Code" in the Real-Time Workshop Embedded Coder documentation and "Optimizing a Model for Code Generation" in the Real-Time Workshop documentation for details on optimization.)

The **Parameter tune level** field allows you to specify whether or not the code generator declares a parameter data object as tunable global data in the generated code.

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for all `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object. If the data object's **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears in the generated code with all of the properties (custom attributes) specified in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13, and thus is tunable. For example, this would occur if **Persistence level** is `2` and **Parameter tune level** is `5`.

Otherwise, the parameter is inlined in the generated code, and Real-Time Workshop settings determine its exact form.

Note that, in the initial stages of development, you may be more concerned about debugging than code size. Or, you may want to ensure that one or more particular data objects appear in the code so that you can analyze intermediate calculations of an equation. In this case, you may want to specify the **Parameter tune level** (**Signal display level** for signals) to be

higher than **Persistence level** for some or all `mpt` parameter (or signal) data objects. This results in larger code size, because the code generator defines the parameter (or signal) data objects as global data, which have all the custom properties you specified. As you approach production code generation, however, you may have more concern about reducing the size of the code and less need for debugging or intermediate analyses. In this stage of the tradeoff, you could make the **Parameter tune level** (**Signal display level** for signals) greater than **Persistence level** for one or more data objects, generate code and observe the results. Repeat until satisfied.

1. With the model open, in the Configuration Parameters dialog box, click **Data Placement** under **Real-Time Workshop**.
2. Type the desired number in the **Signal display level** or **Parameter tune level** field, and click **Apply**.
3. In the Model Explorer, type the desired number in the **Persistence** field for the selected signal or parameter, and click **Apply**.
4. Save the model and generate code.

5

Managing File Placement of Data Definitions and Declarations

Overview of File Placement

This chapter focuses on interdependent selections. Their combined settings, along with the Simulink partitioning, determine what is termed "data placement." This term refers to

- The number of files generated.
- Whether or not the generated files contain definitions for a model's global identifiers. And, if a definition exists, the settings determine the files in which MPF places them.
- Where MPF places global data declarations (`extern`).

The following six MPF selections are distributed among the main procedures and form an important interdependency:

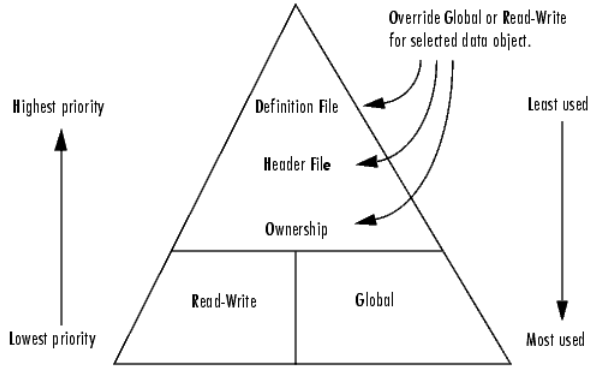
- The **Data definition** field on the **Data Placement** pane of the Configuration Parameters dialog box.
- The **Data declaration** field on the **Data Placement** pane of the Configuration Parameters dialog box.
- The **Owner** field of the data object on the Model Explorer, and the **Module naming** and **Module name** fields on the **Data Placement** pane of the Configuration Parameters dialog box. For discussion purposes, we use the term "Ownership" to refer to these three (**Owner**, **Module naming**, and **Module name**)
- The **Definition file** field of the data object on the Model Explorer.
- The **Header file** field of the data object on the Model Explorer.
- The **Memory section** field of the data object on the Model Explorer.

Priority and Usage

There is a priority among the interdependent selections. From highest to lowest priority, these are called

- Definition File priority
- Header File priority
- Ownership priority
- Read-Write priority or Global priority

But as to usage, the order is reversed. This distinction is illustrated below.



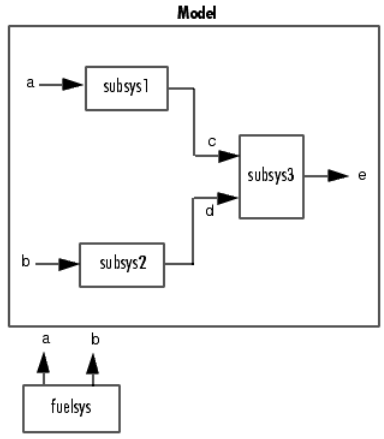
Unless they are overridden, the Read-Write and Global priorities place in the generated files *all* of the model's MPF-derived data objects that you selected using the Data Object Wizard. (See "Creating Data Objects with Data Object Wizard" on page 3-5 for details.) Before generating the files, you can use the higher priority Definition file, Header file, and Ownership, as desired, to override Read-Write or Global priorities for single data objects. Most users will employ Read-Write or Global, without an override. A few users, however, will want to do an override for certain data objects. We expect that those users whose applications include multiple modules will want to use the Ownership priority.

The priorities are in effect only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see "Designing Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation.) Otherwise, Real-Time Workshop determines the data placement.

Read-Write Priority

This is the lowest priority. Consider that a model consists of one or more Simulink blocks or Stateflow diagrams. There can be subsystems within these. For the purpose of illustration, think of a model with one top-level block called `fuelsys`. You double-clicked the block and now see three subsystems labeled `subsys1`, `subsys2` and `subsys3`, as shown in the next figure. Signals `a` and `b` are outputs from the top-level block (`fuelsys`). Signal `a` is an input to `subsys1` and `b` is input to `subsys2`. Signal `c` is an output from `subsys1`. Notice the other inputs and outputs (`d` and `e`). Signals `a` through `e` have corresponding data objects and are part of the code generation data dictionary.

As explained in Chapter 3, "Managing the Data Dictionary" MPF provides you with the means of selecting a data object that you want defined as an identifier in the generated code. MPF also allows you to specify property values for each data object. For this illustration, we choose to include all of the data objects to be in the dictionary.

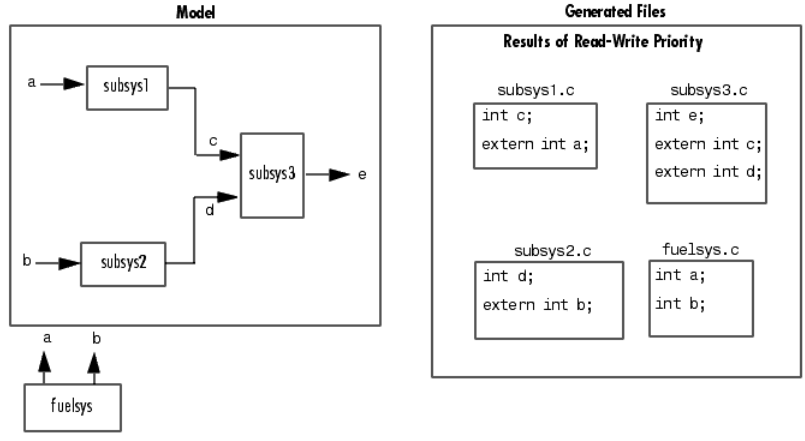


The Generated Files

We generate code for this model. As shown in the figure below, this results in a `.c` source file corresponding to each of the subsystems. (In actual applications, there could be more than one `.c` source file for a subsystem. This is based on the file partitioning previously selected in Simulink for the model. But for our illustration, we only need to show one for each subsystem.) Data objects `a` through `e` have corresponding identifiers in the generated files.

A `.c` source file has one or more functions in it, depending on the internal operations (functions) of its corresponding subsystem. An identifier in a generated .c file has local scope when it is used only in one function of that .c file. An identifier has file scope when more than one function in the same .c file uses it. An identifier has global scope when more than one of the generated files uses it.

A subsystem's source file always contains the definitions for all of that subsystem's data objects that have local scope or file scope. (These definitions are not shown in the figure.) But where are the definitions and declarations for data objects of global scope? These are shown in the next figure.



When the Read-Write priority is in effect, this source file contains the definitions for the subsystem's global data objects, if this is the file that first writes to the data object's address. Other files that read (use) that data object only include a reference to it. This is why this priority is called Read-Write. Since a read and a write of a file are analogous to input and output of a

model's block, respectively, there is another way of saying this. The definitions of a block's global data objects are located in the corresponding generated file, if that data object is an output from that block. The declarations (`extern`) of a block's global data objects are located in the corresponding generated file, if that data object is an input to that block.

Settings for Read-Write Priority

The generated files and what they include, as just described, occur when the Read-Write priority is in effect. For this to be the case, the other priorities are "turned off." That is

- The **Data definition** field on the **Data Placement** pane is set to `Data defined in source file`.
- The **Data declaration** field on the **Data Placement** pane is set to `Data declared in source file`.
- The **Owner** field on the Model Explorer is blank, and the **Module naming** field on the **Data Placement** pane is set to `Not specified`. (When `Not specified` is selected, the **Module name** field does not appear.)
- **Definition file** and **Header file** on the Model Explorer are blank.

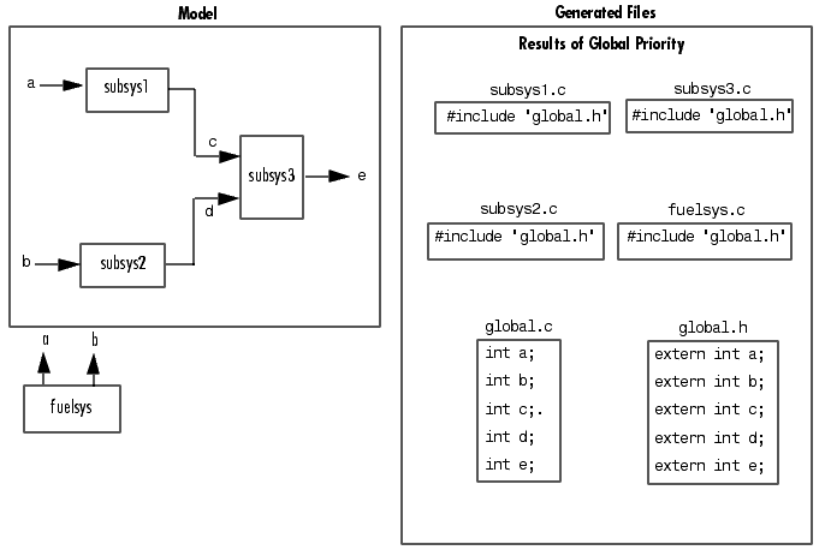
Global Priority

This has the same priority as Read-Write (the lowest) priority. The settings for this are the same as for Read-Write Priority, except

- The **Data definition** field on the **Data Placement** pane is set to `Data defined in single separate source file.`
- The **Data declaration** field on the **Data Placement** pane is set to `Data declared in single separate header file.`

The generated files that result are shown in the next figure. A subsystem's data objects of local or file scope are defined in the `.c` source file where the subsystem's functions are located (not shown). The data objects of global scope are defined in another `.c` file (called `global.c` in the figure). The declarations for the subsystem's data objects of global scope are placed in a `.h` file (called `global.h`).

For example, all data objects of local and file scope for `subsys1` are defined in `subsys1.c`. Signal `c` in the model is an output of `subsys1` and an input to `subsys2`. So `c` is used by more than one subsystem and thus is a global data object. Since global priority is in effect, the definition for `c` (`int c;`) is in `global.c`. The declaration for `c` (`extern int c;`) is in `global.h`. Since `subsys2` uses (reads) `c`, `#include "global.h"` is in `subsys2.c`.



Remaining Priorities

As mentioned previously, the Read-Write and Global priorities operate on *all* MPF-derived data objects that you want defined in the generated code. The remaining priorities allow you to override the Read-Write or Global priorities for one or more particular data objects. There is a high-to-low priority among these remaining priorities: Definition File, Header File, and Ownership, for a particular data object.

Ownership

As mentioned previously, Ownership refers to what you do or do not specify for the **Module naming** and **Module names** fields on the **Data Placement** pane of the Configuration Parameters dialog box, and the **Owner** field on the Model Explorer. These settings have no effect on what files are generated.

Their effects only have to do with definitions and `extern` statements. There are five possible configurations, as indicated in Effects of Ownership Settings on page A-30.

The Memory Section Setting

Regarding **Memory section**, Parameter and Signal Property Values on page A-20 explains that you can select `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile` as the **Memory section** selection. So, if you specify a filename for **Definition file**, and select either `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile` for **Memory section**, Real-Time Workshop Embedded Coder generates a `.c` file and a `.h` file. The `.c` file contains the definition for the data object with the `pragma` statement or qualifier associated with the **Memory section** selection. The `.h` file contains the declaration for the data object. Then the `.h` file, with the preprocessor directive `#include`, can be included in any file that needs to reference the data object. You can add more memory sections. See “Designing Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation.

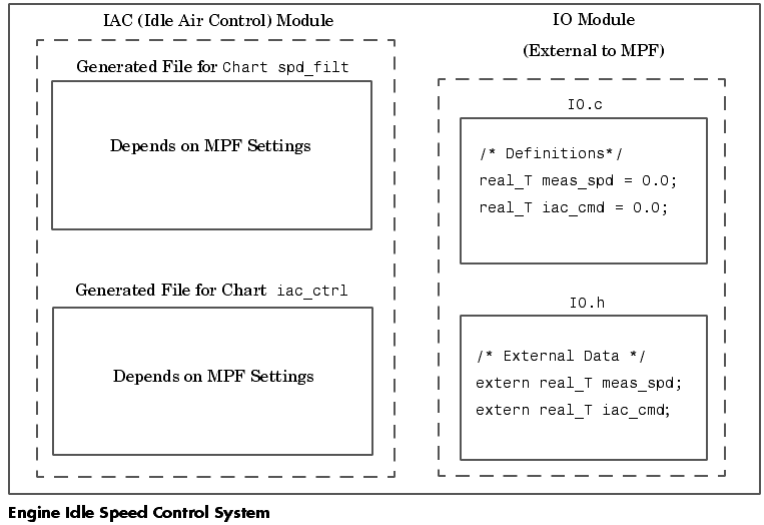
Data Placement Rules

For a complete set of data placement rules in convenient tabular form, based on the priorities discussed in this chapter, see “Data Placement Rules and Effects” on page A-30.

Example Settings

Example Settings and Resulting Generated Files on page A-31, provides example settings for one data object of a model. Eight examples are listed so that you can see the generated files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. These discussions provide adequate information for understanding the effects of any settings you might choose. For illustration purposes, the four examples assume that we are dealing with an overall system that controls engine idle speed.

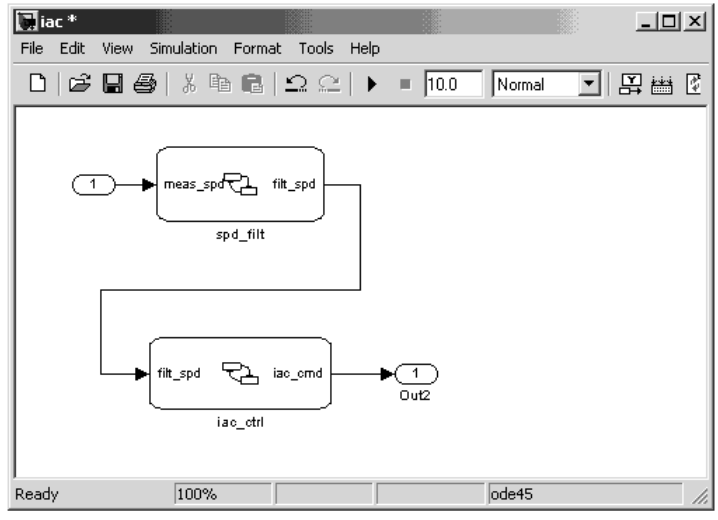
The next figure shows that the software component of this example system consists of two modules, IAC (Idle Air Control), and IO (Input-Output).



Engine Idle Speed Control System

The code in the IO module controls the system's IO hardware. Code is generated only for the IAC module. (Some other means produced the code for the IO module, such as hand-coding.) So the code in IO is external to MPF, and can illustrate legacy code. To simplify matters, the IO code contains one source file, called `IO.c`, and one header file, called `IO.h`.

The IAC module consists of two Stateflow charts, `spd_filt` and `iac_ctrl`. The `spd_filt` chart has two signals (`meas_spd`) and `filt_spd`), and one parameter (`a`). The `iac_ctrl` chart also has two signals (`filt_spd` and `iac_cmd`) and a parameter (`ref_spd`). (The parameters are not visible in the top-level charts.) One file for each chart is generated. This example system allows us to illustrate referencing from file to file within the MPF module, and model to external module. It also illustrates the case where there is no such referencing.



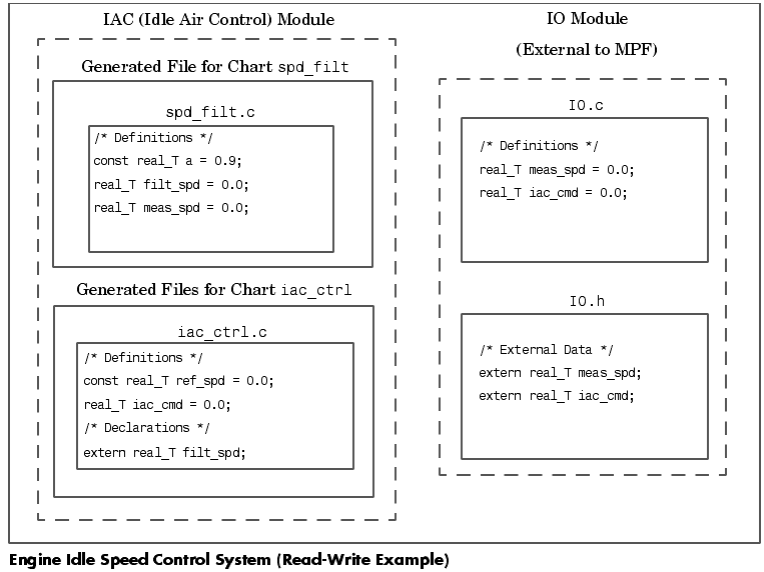
Proceed to the discussion of the desired example settings:

- "Read-Write Example" on page 5-11
- "Ownership Example" on page 5-12
- "Header File Example" on page 5-13
- "Definition File Example" on page 5-15

Read-Write Example

These settings and the generated files that result are shown as Example Settings 1 in Example Settings and Resulting Generated Files on page A-31. As you can see from the table, this example illustrates the case in which only one `.c` source file (for each chart) is generated.

So, for the IAC model, select the following settings. Accept the `Data defined in source file` in the **Data definition** field and the `Data declared in source file` in the **Data declaration** field on the **Data Placement** pane of the Configuration Parameters dialog box. Accept the default `Not specified` selection in the **Module naming** field. Accept the default blank settings for the **Owner**, **Definition file** and **Header file** fields on the Model Explorer. For **Memory section**, accept `Default`. Now the Read-Write priority is in effect. Generate code. The next figure shows the results in terms of definition and declaration statements.



Engine Idle Speed Control System (Read-Write Example)

The code generator generated a `spd_filt.c` for the `spd_filt` chart and `iac_ctrl.c` for the `iac_ctrl` chart. As you can see, MPF placed all definitions of data objects for the `spd_filt` chart in `spd_filt.c`. It placed all definitions of data objects for the `iac_ctrl` chart in `iac_ctrl.c`.

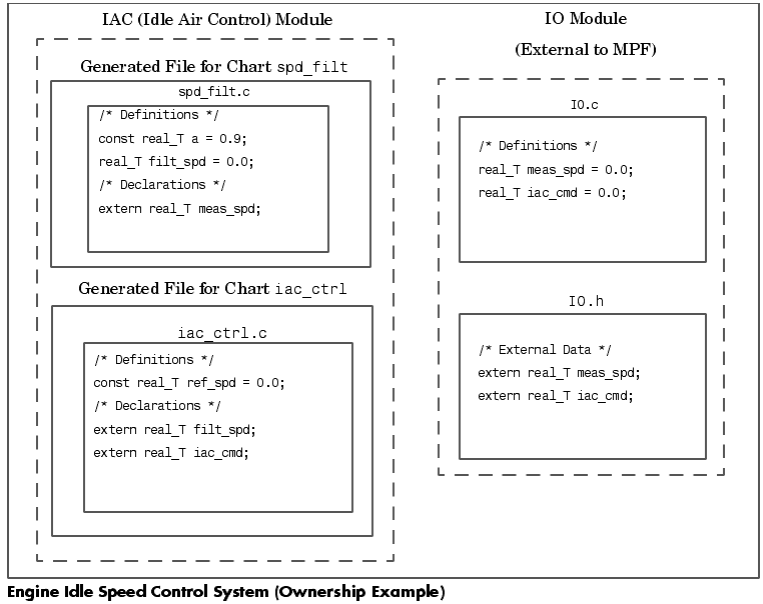
However, notice `real_T filt_spd`. This data object is defined in `spd_filt.c` and declared in `iac_ctrl.c`. That is, since the Read-Write priority is in effect, `filt_spd` is defined in the file that first writes to its address. And, it is declared in the file that reads (uses) it. Further, `real_T meas_spd` is defined in both `spd_filt.c` and the external `IO.c`. And, `real_T iac_cmd` is defined in both `iac_ctrl.c` and `IO.c`.

Ownership Example

See tables Effects of Ownership Settings on page A-30 and Example Settings and Resulting Generated Files on page A-31. In the "Read-Write Example" on page 5-11, there are several instances where the same data object is defined in more than one `.c` source file, and there is no declaration (`extern`) statement. This would result in compiler errors during link time. But in this example, we configure MPF Ownership rules so that adequate linking can take place. Notice the Example Settings 2 row in Example Settings and Resulting Generated Files on page A-31. Except for the Ownership settings, assume these are the settings you made for the model in the IAC module. Since this example has no **Definition file** or **Header file** specified, now Ownership takes priority. (If there *were* a **Definition file** or **Header file** specified, MPF would ignore the Ownership settings.)

On the **Data Placement** pane of the Configuration Parameters dialog box, select `User specified` in the **Module naming** field, and specify IAC in the **Module name** field (case sensitive). Open the Model Explorer (by issuing the MATLAB command `daexplr`) and, for all data objects except `meas_spd` and `iac_cmd`, type IAC in the **Owner** field (case sensitive). Then, only for the `meas_spd` and `iac_cmd` data objects, type IO as their **Owner** (case sensitive). Generate code.

The results are shown in the next figure. Notice the `extern real_T meas_spd` statement in `spd_filt.c`, and `extern real_T iac_cmd` in `iac_ctrl.c`. MPF placed these declaration statements in the correct files where these data objects are used. This allows the generated source files (`spd_filt.c` and `iac_ctrl.c`) to be compiled and linked with `IO.c` without errors.



Engine Idle Speed Control System (Ownership Example)

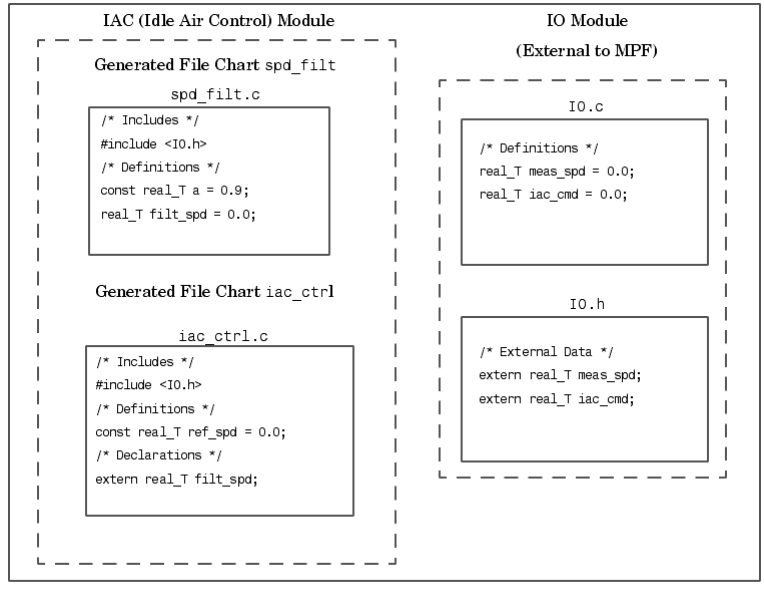
Header File Example

These settings and the generated files that result are shown as Example Settings 3 in Example Settings and Resulting Generated Files on page A-31. Since this example has no **Definition file** specified, it allows us to describe the effects of the **Header file** setting. (If there *were* a **Definition file**, MPF would ignore the **Header file** setting.) The focus of this example is to show how the **Header file** settings result in the linking of the two chart source files to the external IO files, shown in the next figure. (Also, Ownership settings will be used to link the two chart files with each other.)

As you can see in the figure, the `meas_spd` and `iac_cmd` identifiers are defined in `IO.c` and declared in `IO.h`. Both of these identifiers are external to the generated `.c` files. You open the Model Explorer and select both the `meas_spd`

and iac_cmd data objects. For each of these data objects, in the **Header file** field, specify IO.h, since this is where these two objects are declared. This setting ensures that the spd_filt.c source file will compile and link with the external IO.c file without errors.

Now we configure the Ownership settings. In the Model Explorer, select the filt_spd data object and set its **Owner** field to IAC. Then, on the **Data Placement** pane of the Configuration Parameters dialog box, select User specified in the **Module naming** field, and specify IAC in the **Module Name** field. This ensures that the spd_filt source file will link to the iac_ctrl source file. Generate code. See the figure below.



Engine Idle Speed Control System (Header File Example)

Since you specified the IO.h filename for the **Header file** field for the meas_spd and iac_ctrl objects, the code generator assumed correctly that their declarations are in IO.h. So the code generator placed #include IO.h in each source file: spd_filt.c and iac_ctrl.c. So these two files will link with the external IO files. Also, due to the Ownership settings that were specified, the code generator places the real_T filt_spd = 0.0; definition in

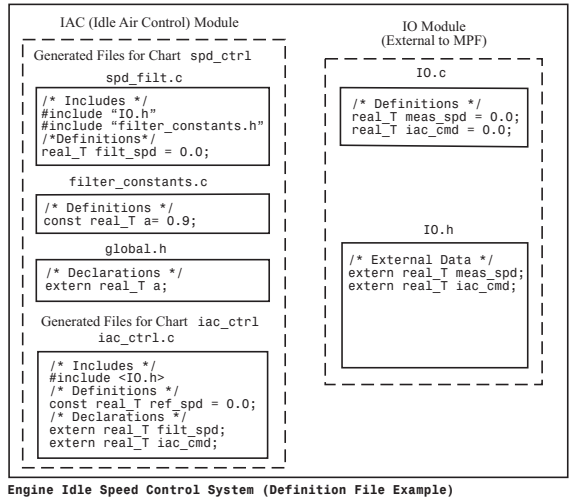
`spd_filt.c` and declares the `filt_spd` identifier in `iac_ctrl.c` with `extern real_T iac_cmd;`. Consequently, the two source files will link together.

Definition File Example

These settings and the generated files that result are shown as Example Settings 4 in Example Settings and Resulting Generated Files on page A-31. Notice that a definition filename is specified. The settings in the table only apply to the data object called `a`. You have decided that you do not want this object defined in `spd_filt.c`, the generated source file for the `spd_filt` chart. (There are many possible organizational reasons one might want an object declared in another file. It is not important for this example to specify the reason.)

For this example, assume the settings for all data objects are the same as those indicated in "Header File Example" on page 5-13, except for the data object `a`. The description below identifies only the differences that result from this.

Open the Model Explorer, and select data object `a`. In the **Definition file** field you specify any desired filename. Choose `filter_constants.c`. Generate code. The results are shown in the next figure.



Engine Idle Speed Control System (Definition File Example)

The code generator generates the same files as in the “Header File Example” on page 5-13, and adds a new file, `filter_constants.c`. Data object `a` now is defined in `filter_constants.c`, rather than in the source file `spd_filt.c`, as it is in the example. This data object is declared with an `extern` statement in `global.h`

A

Referenced Tables

MPF Panes on the Configuration Parameters Dialog Box (p. A-2)	Lists and describes all elements on MPF-related panes on the Configuration Parameters dialog box.
Template Symbols and Rules (p. A-11)	Lists and describes all MPF and template symbol rules.
Parameter and Signal Properties (p. A-19)	Lists and describes all mpt parameter and signal properties and property values, and illustrates how changing these affect the generated code.
Data Placement Rules and Effects (p. A-30)	Shows the effects that example changes to the interdependent MPF settings have on the generated code. Also, provides a complete set of data placement rules.

MPF Panes on the Configuration Parameters Dialog Box

The following tables define elements on each MPF-related pane on the Configuration Parameters dialog box. Elements that are not related to MPF are not described. Select **Real-Time Workshop** on the **Select** pane.

MPF Elements on Configuration Parameters Panes

Pane	Element	Description
General	Ignore custom storage classes	To make module packaging features available, this check box must be cleared.
Comments	Simulink data object descriptions	When this check box is selected, and you type text in the **Description** field of the Model Explorer, that text will appear beside the signal's or parameter's identifier in the generated code as a comment.
	Custom comments (MPT objects only)	When selected, this check box allows you to add a comment above a signal or parameter's identifier in the generated code. You control the content of the comment by writing a function in M-code (`.m` file) or TLC-code (`.tlc` file), and specifying its filename in the **Custom comments function** field.
	Custom comments function	In this field, you specify the `.m` filename or `.tlc` filename that contains the function mentioned just above. This field is available only when the **Custom comments (MPT objects only)** check box is selected.
Symbols	**#define naming**	This rule applies only to those parameters whose storage class you selected as `Define` in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13. Allows you to specify one rule by which all of these parameters change the same way. Then, they appear as identifiers in the generated code as you want.

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		For example, in "Creating mpt Data Objects, Setting Property Values, and Generating Code" on page 3-13, a parameter is named `parama`. For this parameter, you specified `Define (Custom)` in the **Storage class** field of the Model Explorer, and you specified its `Value` property as `"1."` So, in terms of ANSI-C/C++ syntax, you have said `#define parama 1;`. Now you select `Force upper case` in the **#define naming** field of the **Symbols** pane of the Configuration Parameters dialog box. The result of all of this is as follows. `"PARAMA"` appears in the generated code file every time this parameter name appears. In the compiled executable file, `"1"` appears every time `"PARAMA"` appears in the generated code file. In the **#define naming** field, select `Custom M-function` to write *your own* naming rule that changes all of these parameter names in the model to identifiers in the generated code, in the same way. Then you must write an M-function to accomplish this. For details on writing a MATLAB function, see Functions in the MATLAB documentation. Of course, there is a wide variety of possibilities. Some examples are • Remove all underscore characters in all signal names • Add underscores before a capital letter in all parameter names • Make all identifiers in the generated code uppercase

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		Then you save the function as a `.m` file, place it in any folder in the MATLAB path, and type its filename in the **M-function** field under the **#define naming** field. Select `Force upper case` *or* `Force lower case` to change case as desired. Select `None` to make no change to the `#define` names. With this selection, after code generation, all of them will appear as identifiers in the source code exactly as they appear in the model.
	M-function	If you selected `Custom M-function` in the **#define naming** field, place the name of the `.m` file here, with or without the `.m` extension. Otherwise, ignore this field.
	Parameter naming	Allows you to specify one rule by which all of the model's parameter names change the same way, so that they appear as identifiers in the generated code as you want. The selections in this field have the same functions as described above for `#defines`, except they apply to parameter names.
	M-function	If you selected `Custom M-function` in the **Parameter naming** field, place the name of the `.m` file here, with or without the `.m` extension. Otherwise, ignore this field.
	Signal naming	Allows you to specify one rule by which all of the model's signal names change the same way, so that they appear as identifiers in the generated code as you want. The selections in this field have the same functions as described above for `#defines`, except they apply to signal names.

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	M-function	If you selected `Custom M-function` in the **Signal naming** field, place the name of the `.m` file here, with or without the `.m` extension. Otherwise, ignore this field.
Templates	**Code templates**	A code template organizes all of the generated files that, primarily, contain functions but not identifiers.
	Source file (*.c) template	The source code template organizes C code files. These include, for example, the main `.c` or any of the `.c` files that contain functions that Real-Time Workshop Embedded Coder generates for the open model.
	Header file (*.h) template	The header code template organizes the `.h` file that includes the prototypes of these functions. (See **Source file (*.c) template** just above.)
	Data templates	A data template organizes all of the generated files that contain only identifiers (data), not functions (code).
	Source file (*.c) template	The source data template organizes the `.c` file that contains definitions of variables of global scope.
	Header file (*.h) template	The header data template organizes the `.h` file that can contain declarations of variables of global scope. (See **Source file (*.c) template** just above.)

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	Custom templates	A custom template has priority over the code and data templates in organizing the generated files. As its name suggests, this template is for advanced users who want to customize how the generated files are organized, by using this one template. For details, see "Custom File Processing" in the Real-Time Workshop Embedded Coder documentation.
Data Placement	**Data definition**	In this field, you select the `.c` file where the definitions of variables of global scope will be located. You can place these in a single `.c` file that is separate from the `.c` files where the model's functions are located, if desired. If you choose `Data defined in single separate source file`, the data source template specified in the **Source file (*.c) template** field of the **Templates** pane (for Data templates) will be used. This template file organizes the single separate source file. You must also specify the filename of this single separate source file itself in the **Data definition filename** field below. Or, you can place these definitions in the `.c` files where the functions *are* located. To do this you select `Data defined in source file`. In this case, the source template will not be used. There may be one function `.c` file or multiple function `.c` files, based on the file partitioning previously selected in Simulink for the model. If there are multiple files, and you select `Data defined in source file`, all of the definitions will be placed in their respective function files.

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		If you choose the default `Auto`, Real-Time Workshop Embedded Coder determines where the definitions will be located.
	Data definition filename	This field is available only if you selected `Data defined in single separate source file` in the **Data definition** field. Specify here the name of this source file.
	Data declaration	In this field, you select the file where declarations will be located (`extern`, `typedef` and `#define` statements). You can place these in a single `.h` file that is separate from the `.c` files where the model's functions are located, if desired. If you choose `Data declared in single separate header file`, the data header template specified in the **Header file (*.h) template** field of the **Templates** pane (for Data templates) will be used. This template file organizes the single separate header file. You must also specify the filename of this single separate header file itself in the **Data declaration filename** field below. Or, you can place these declarations in the `.c` files where the functions *are* located. To do this you select `Data declared in source file`. In this case, the data header template will not be used. As mentioned previously, there may be one function `.c` file or multiple function `.c` files, based on the file partitioning previously selected in Simulink for the model. If there are multiple files, and you select `Data declared in source file`, all of the declarations will be placed in their respective function files.

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		If you choose the default `Auto`, Real-Time Workshop Embedded Coder determines where the declarations will be located.
	Data declaration filename	This field is available only if you selected `Data declared in single separate header file` in the **Data declaration** field. Specify here the name of this header file.
	#include file delimiter	This field allows you to select the `#include` file delimiter used in those generated files that contain the `#include` preprocessor directive for `mpt` data objects. This applies the selected delimiter to all `mpt` data objects, except any whose delimiter is overridden by the **Header file** field on the Model Explorer. If you select `Auto`, Real-Time Workshop Embedded Coder determines the delimiter. If you select `#include "header.h"`, the double-quotation delimiter is used. If you select `#include <header.h>`, the angle-bracket delimiter is used.
	Module naming	In this field, you select whether or not to name the module. This is used in conjunction with the **Owner** field of a data object in the Model Explorer to constitute what is termed "ownership." For details, see “Ownership” on page 5-7 and Effects of Ownership Settings on page A-30. If you do want to specify the module name, you can select the convenient `Same as model`. This avoids having to type in a name in the **Module name** field described below.

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	Module name	This field is available only if you selected `User specified` in the **Module naming** field. Type the desired module name according to ANSI C/C++ conventions for naming identifiers.
	Signal display level	This field allows you to specify whether or not the code generator declares a signal data object as global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field on the Module Explorer dialog box. The **Signal display level** number is for all `mpt` signal data objects in the model. The **Persistence level** number is for a *particular* `mpt` signal data object.
	Parameter tune level	This field allows you to specify whether or not the code generator declares a parameter data object as tunable global data in the generated code. The number you specify in this field is relative to the number you specify in the **Persistence level** field on the Module Explorer dialog box. The **Parameter tune level** number is for all `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object.
	Source of initial values	This field allows you to select the source that initializes each of the model's signals in the generated code. You can select `Model` or `Data object`. `Model` is the default. If you accept `Model`, the Simulink default data initialization controls the initialization for each signal. If you select `Data object`, the initial value of the data object determines the initialization

MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		statement, according to the data initialization configuration of the custom storage class.

Template Symbols and Rules

The following tables describe all MPF template symbols and rules for using these. The location of a symbol in one of the MPF template files (`code_c_template.cgt`, `code_h_template.cgt`, `data_c_template.cgt`, or `data_h_template.cgt`) determines where the items associated with this symbol are located in the corresponding generated file. The first table identifies the symbol groups, starting with the parent ("Base") group, followed by the children in each parent.Template Symbols on page A-13 lists the symbols in alphabetical order. “Rules for Modifying or Creating a Template” on page A-17 lists the rules.

Template Symbol Groups

Symbol Group	Symbol Names in This Group

Base (Parents)	`Declarations`
	`Defines`
	`Definitions`
	`Documentation`
	`Enums`
	`Functions`
	`Includes`
	`Types`

Declarations	`ExternalCalibrationLookup1D`
	`ExternalCalibrationLookup2D`
	`ExternalCalibrationScalar`
	`ExternalVariableScalar`

Defines	`LocalDefines` `LocalMacros`

Template Symbol Groups (Continued)

Symbol Group	Symbol Names in This Group
Definitions	FilescopeCalibrationLookup1D
	FilescopeCalibrationLookup2D
	FilescopeCalibrationScalar
	FilescopeVariableScalar
	GlobalCalibrationLookup1D
	GlobalCalibrationLookup2D
	GlobalCalibrationScalar
	GlobalVariableScalar
Documentation	Abstract
	Banner
	Created
	Creator
	Date
	Description
	FileName
	History
	LastModificationDate
	LastModifiedBy
	ModelName
	ModelVersion
	ModifiedBy
	ModifiedComment
	ModifiedDate
	Modified History

Template Symbol Groups (Continued)

Symbol Group	Symbol Names in This Group
	`Notes` `ToolVersion`
Functions	`CFunctionCode`
Types	This parent has no children.

Template Symbols

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Abstract`	Documentation	N/A	User-supplied description of the model or file. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Banner`	Documentation	N/A	Comments located near top of the file. Contains information that includes versions of model and Real-Time Workshop, and date file was generated.
`CFunctionCode`	Functions	File	All of the C/C++ functions. Must be at the bottom of the template.
`Created`	Documentation	N/A	Date when model was created. From **Created on** field on Model Properties dialog box.
`Creator`	Documentation	N/A	User who created model. From **Created by** field on Model Properties dialog box.

Template Symbols (Continued)

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Date`	Documentation	N/A	Date file was generated. Taken from computer clock.
`Declarations`	Base		Data declaration of any signal or parameter. For example, `extern real_T globalvar;`.
`Defines`	Base	File	Any necessary `#defines` of `.h` files.
`Definitions`	Base	File	Data definition of any signal or parameter.
`Description`	Documentation	N/A	Description of model. From **Model description** field on Model Properties dialog box.**
`Documentation`	Base	N/A	Comments about how to interpret the generated files from Real-Time Workshop.
`Enums`	Base	File	Enumerated data type definitions.
`ExternalCalibrationLookup1D`	Declarations	External	***
`ExternalCalibrationLookup2D`	Declarations	External	***
`ExternalCalibrationScalar`	Declarations	External	***
`ExternalVariableScalar`	Declarations	External	***
`FileName`	Documentation	N/A	Name of the generated file.
`FilescopeCalibrationLookup1D`	Definitions	File	***
`FilescopeCalibrationLookup2D`	Definitions	File	***
`FilescopeCalibrationScalar`	Definitions	File	***
`FilescopeVariableScalar`	Definitions	File	***
`Functions`	Base	File	Generated function code.

Template Symbols (Continued)

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`GlobalCalibrationLookup1D`	Definitions	Global	***
`GlobalCalibrationLookup2D`	Definitions	Global	***
`GlobalCalibrationScalar`	Definitions	Global	***
`GlobalVariableScalar`	Definitions	Global	***
`History`	Documentation	N/A	User-supplied revision history of the generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`Includes`	Base	File	`#include` preprocessor directives.
`LastModificationDate`	Documentation	N/A	Date when model was last saved. From **Last saved on** field on Model Properties dialog box.
`LastModifiedBy`	Documentation	N/A	User who last saved model. From **Last saved by** field on Model Properties dialog box.
`LocalDefines`	Defines	File	`#define` preprocessor directives from code-generation data dictionary.
`LocalMacros`	Defines	File	C/C++ macros local to the file.
`ModelName`	Documentation	N/A	Name of the model.
`ModelVersion`	Documentation	N/A	Version number of the Simulink model.

Template Symbols (Continued)

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`ModifiedBy`	Documentation	N/A	Name of user who last modified the model. From **Model version** field on Model Properties dialog box.
`ModifiedComment`	Documentation	N/A	Comment user enters in the **Modified Comment** field on the Log Change dialog box. See "Creating a Model Change History" in the Simulink documentation.
`ModifiedDate`	Documentation	N/A	Date model was last modified before code was generated.
`ModifiedHistory`	Documentation	N/A	Text from **Modified history** field on Model Properties dialog box.**
`Notes`	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`ToolVersion`	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
`Types`	Base		Data types of generated code.

* All symbol names must be enclosed between `%< >`. For example, `%<Functions>`.

** This symbol can be used to add a comment to the generated files. See "Adding Global Comments" on page 4-7. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

*** The description can be deduced from the symbol name. For example, `GlobalCalibrationScalar` is a symbol that identifies a scalar. It contains data of global scope that you can calibrate .

Rules for Modifying or Creating a Template

The following are the rules for creating any MPF template. "Comparison of a Template and Its Generated File" on page 2-9 illustrates several of these rules.

1. Place a symbol on a template within the `%< >` delimiter. For example, the symbol named `Includes` should look like this on a template: `%<Includes>`. *Note that symbol names are case sensitive.*

2. Place a symbol on a template where desired. Its location on the template determines where the item associated with this symbol is located in the generated file. If no item is associated with it, the symbol is ignored.

3. Place a C/C++ statement outside of the `%< >` delimiter, and on a different line than a `%< >` delimiter, for that statement to appear in the generated file. For example, `#pragma message ("my text")` in the template results in `#pragma message ("my text")` at the corresponding location in the generated file. Note that the statement must be compatible with your C/C++ compiler.

4. Use the `.cgt` extension for every template filename. ("`cgt`" stands for *c*ode *g*eneration *t*emplate.)

5. Note that `%% $Revision: 1.1.4.10.4.1 $` appears at the top of the MathWorks supplied templates. This is for internal MathWorks use only. It does not need to be placed on a user-defined template and does not show in a generated file.

6 Place a comment on the template between `/* */` as in standard ANSI C. This results in `/*comment*/` on the generated file.

7 Each MPF template must have all of the Base group symbols, in predefined order. They are listed in Template Symbol Groups on page A-11. Each symbol in the Base group is a parent. For example, `Declarations` is a parent symbol.

8 Each symbol in a non-Base group is a child. For example, `LocalMacros` is a child.

9 Except for Documentation children, all children must be placed after their parent, before the next parent, and before the `Functions` symbol.

10 Documentation children can be located before or after their parent in any order anywhere in the template.

11 If a non-Documentation child is missing from the template, the code generator places the information associated with this child at its parent location in the generated file.

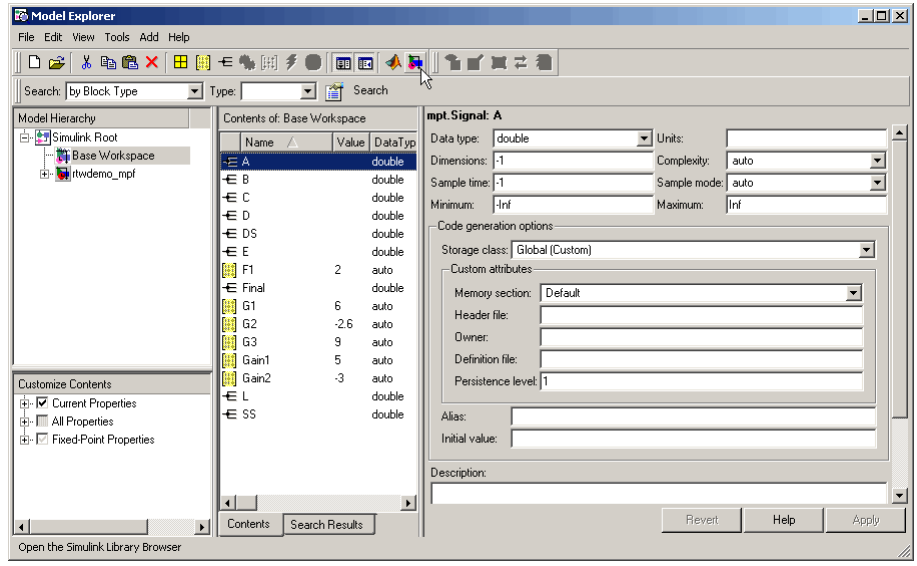
12 If a Documentation child is missing from the template, the code generator omits the information associated with that child from the generated file.

Parameter and Signal Properties

The following table describes the properties and property values for all `mpt.Parameter` and `mpt.Signal` data objects that appear on the Model Explorer.

The figure below shows an example of the Model Explorer. When you select an `mpt.Parameter` or `mpt.Signal` data object in the middle pane, its properties and property values display in the right-most pane.

In the Properties column, the table lists the properties in the order in which they appear on the Model Explorer. Another table describes the effects that example changes to property values have on the generated code.



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	User object type	*auto	Prenamed and predefined property sets that are registered in the custom_user_object_type_ info.m file. (See the procedure "Registering User Object Types" on page 3-38.) This field is unavailable if no user object type is registered. Select auto if this field is available but you do not want to apply the properties of a user object type to a selected data object. The fields on the Model Explorer are populated with default values.
		Any user object type name listed	Select a user object type name to apply the properties and values that you associated with this name in the custom_user_object_type_ info.m file. The fields on the Model Explorer are automatically populated with those values.
Parameter	Value	*0	The data type and numeric value of the data object. For example, int8(5). The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an mpt.Signal data object, but not for an mpt.Parameter data object. The data type for an mpt.Parameter data object is specified in the **Value** field above. See "Working with Data Types" in the Simulink documentation.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Units	*null	Units of measurement of the signal or parameter. (Enter text in this field.)
Both	Dimensions	*-1	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.
Both	Complexity	*auto real complex	Complexity specifies whether the signal or parameter is a real or complex number. Select auto for the code generator to decide. For a parameter, the complexity is derived from its value.
Signal	Sample time	*-1	Model or block execution rate.
Signal	Sample mode	*auto	Determines how the signal propagates through the model. Select auto for the code generator to decide.
		Sample based	The signal propagates through the model one sample at a time.
		Frame based	The signal propagates through the model in batches of samples.
Both	Minimum	*0.0	The minimum value to which the parameter or signal is expected to be bound.
		Any number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Maximum	*0.0	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		
	Storage class		Note that an auto selection for a storage class tells Real-Time Workshop to decide how to declare and store the selected parameter or signal.
Both	Default (Custom)		Real-Time Workshop Embedded Coder decides how to declare the data object.
Both	Global (Custom)	Global (Custom) is the default storage class for mpt data objects.	Ensures that the code generator places no qualifier in the data object's declaration.
Both	Memory section	*Default	**Memory section** allows you to specify storage directives for the data object. Default ensures that the code generator places no type qualifier and no pragma statement with the data object's declaration.
Parameter		MemConst	Places the const type qualifier in the declaration.
Both		MemVolatile	Places the volatile type qualifier in the declaration.
Parameter		MemConstVolatile	Places the const volatile type qualifier in the declaration.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Header file		Name of the file used to import or export the data object. This file contains the declaration (`extern`) to the data object. Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the `.h` extension. For example, `"object.h"` or `"object"` has the same effect. For the selected data object, this overrides the general delimiter selection in the **#include file delimiter** field on the Configuration Parameters dialog box.
Both	Owner	*Blank	The name of the module that owns this signal or parameter. This is used to help determine the ownership of a definition. For details, see “Ownership” on page 5-7 and Effects of Ownership Settings on page A-30.
Both	Definition file	*Blank	Name of the file that defines the data object.
		Any valid text string	

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Persistence level		The number you specify is relative to **Signal display level** or **Parameter tune level** on the **Data Placement** pane of the Configuration Parameters dialog box. For a signal, allows you to specify whether or not the code generator declares the data object as global data. For a parameter, allows you to specify whether or not the code generator declares the data object as tunable global data. See **Signal display level** and **Parameter tune level** in MPF Elements on Configuration Parameters Panes on page A-2.
Both	Bitfield (Custom)		Embeds Boolean data in a named bit field.
	Struct name		Name of the `struct` into which the object's data will be packed.
Parameter	Const (Custom)		Places the `const` type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Both	Volatile (Custom)		Places the `volatile` type qualifier in the declaration.
Both	Header file		See above.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Owner		See above.
Both	Definition file		See above.
Both	Persistence level		See above.
Parameter	ConstVolatile (Custom)		Places the `const volatile` type qualifier in declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Parameter	Define (Custom)		Represents parameters with a `#define` macro.
Parameter	Header file		See above.
Both	ExportToFile (Custom)		Generates global variable definition, and generates a user-specified header (`.h`) file that contains the declaration (`extern`) to that variable.
Both	Memory section		See above.
Both	Header file		See above.
Both	Definition file		See above.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	ImportFromFile (Custom)		Includes predefined header files containing global variable declarations, and places the `#include` in a corresponding file. Assumes external code defines (allocates memory) for the global variable.
Both	Data access	`*Direct`	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type (`Direct`) or stores the address of the data (a pointer).
Both		`Pointer`	If you select `Pointer`, the code generator places `*` before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named `struct` to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Alias	*null	As explained in detail in "Applying Naming Rules to Identifiers Globally" on page 3-20, for a Simulink or mpt data object (identifier), specifying a name in the **Alias** field overrides the global naming rule selection you make on the Configuration Parameters dialog.
		Any valid ANSI C/C++ variable name	
Signal	Initial value	[ ]	Numeric value used as an initial value in the generated code.
Both	Description	*null	Text description of the parameter or signal. Appears as a comment beside the signal or parameter's identifier in the generated code.
		Any text string	

Some Examples of the Effect of Property Value Changes on Generated Code

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 1: Parameter data objects can be declared or defined as constants. I know that the data object `GAIN` is a parameter. I want this to be declared or defined in the `.c` file as a variable. But I notice that `GAIN` is declared as a constant by the statement `const real_T GAIN = 5.0;`. Also, this statement is in the constant section of the file.	In the Model Explorer, I clicked the data object `GAIN`. I noticed that the property value for its **Memory section** property is set at `MemConst`. I changed this to `Default`.	I notice two differences. One is that now `GAIN` is declared as a variable with the statement `real_T GAIN = 5.0;`. The second difference is that the declaration now is located in the `MemConst` memory section in the .c or .cpp file.
Example 2: I notice again the declaration of `GAIN` in the `.c` file mentioned in Example 1. It appears as `real_T GAIN = 5.0;`. But I have changed my mind. I want data object `GAIN` to be `#define`.	I changed the **Storage class** selection to `Define (Custom)`.	`GAIN` is no longer declared in the `.c` file as a `MemConst` parameter. Rather, it now is defined as a `#define` macro by the code `#define GAIN 5.0`, and this is located near the top of the `.c` file with the other preprocessor directives.
Example 3: I changed my mind again after doing Example 2. I do want `GAIN` defined using the `#define` preprocessor directive. But I do not want to include the `#define` in this file. I know it exists in another file and I want to reference that file.	On the Model Explorer, I notice that the property value for the **Header file** property is blank. I changed this to `filename.h`. (I chose the ANSI C/C++ double quote mechanism for the `#include`, but could have chosen the angle bracket mechanism.) Also, it is necessary that I make the user-defined `filename.h` available to the	The `#define GAIN 5.0` is no longer in this `.c` file. Instead, the `#include filename.h` code appears as a preprocessor directive at the top of the file.

Some Examples of the Effect of Property Value Changes on Generated Code (Continued)

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
	compiler, placing it either in the system path or local directory.	
Example 4: I have one more change I want to make. Let us say that we have declared the data object `data_in`, and that its declaration statement in the `.c` file reads `real_T data_in = 0.0;` I want to replace this in all locations in the `.c` with an alias.	In the Model Explorer, I selected the data object `data_in`. I noticed that the **Alias** field is blank. I changed this to `data_in_alias`, which I know is a valid ANSI C/C++ variable name.	The identifier `data_in_alias` now appears in the `.c` file everywhere `data_in` appeared.

Data Placement Rules and Effects

The following tables show the effects that example changes to the interdependent MPF settings have on the generated code. See "Example Settings" on page 5-9. Data Placement Rules on page A-33 provides a complete set of data placement rules.

Effects of Ownership Settings

Row Number	Module Naming Setting	Owner Setting	Effect*
1	`Not specified`**	Blank**	There is a definition for the selected data object. The code generator places this definition in the `.c/.cpp` source file that uses it. There is also an `extern` declaration for this data object. The code generator places this `extern` declaration in one or more `.h` header files, as needed.
2	`Not specified`**	A name is specified.	Same effect as stated above.
3	Either `Same as model` or `User specified` is selected.	Blank**	Same as Row 1.
4	Either `Same as model` or `User specified is selected`, and this name is the same as that specified as the **Owner** property.	A name is specified and it is the same as that specified in the **Module naming** (**Module name**) field.	Same as Row 1.
5	Either `Same as model` or `User specified` is selected, and this	A name is specified but it is different from that specified in	There is no definition for the selected data object. However, there is an `extern` declaration for the object. The

Effects of Ownership Settings (Continued)

Row Number	Module Naming Setting	Owner Setting	Effect*
	name is different than that specified as the **Owner** property.	the **Module naming** (**Module name**) field.	`extern` declaration is placed in one or more header files, as needed.

* See also "Ownership" on page 5-7.

** Default

Example Settings and Resulting Generated Files

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	Source file	Source file	Blank	Blank	Blank	`.c`/`.cpp` source file
Example Settings 2 (Owner- ship Example)	Source file	Source file	Name of module specified	Blank	Blank	`.c`/`.cpp` source file
Example Settings 3 (Header File Example)	Source file	Source file	Blank	Blank	Desired include filename specified.	`.c`/`.cpp` source file `.h` definition file
Example Settings 4 (Def. File Example)	Source file	Source file	Blank	Desired definition filename specified.	Desired include filename specified.	`.c`/`.cpp` source file `.c`/`.cpp` definition file* `.h` definition file*

Example Settings and Resulting Generated Files (Continued)

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files
Example Settings 5	Single separate source file	Source file	Blank	Blank	Blank	`.c/.cpp` source file global `.c/.cpp`

Example Settings 6	Single separate source file	Single separate header file	Blank	Blank	Blank	`.c/.cpp` source file
						`global .c/.cpp`
						`global.h`

Example Settings 7	Single separate source file	Single separate header file	Name of module specified	Blank	Blank	`.c/.cpp` source file
						`global.c/.cpp`
						`global.h`

Example Settings 8	Single separate source file	Single separate header file	Blank	Blank	Desired include filename specified.	`.c/.cpp` source file
						`global.c/.cpp`
						`global.h`
						`.h` definition file

* "Blank" in ownership setting means `Not specified` is selected in the **Module naming** field on the **Data Placement** pane, and the **Owner** field on the Model Explorer is blank. "Name of module specified" can be a variety of ownership settings as defined in Effects of Ownership Settings on page A-30.

** The code generator generates a definition `.c/.cpp` file for every data object for which you specified a definition filename (unless you selected `#DEFINE` for the **Memory section** field). For example, if you specify the same definition filename for all data objects, only one definition `.c/.cpp` file is generated. The code generator places declarations in *`model`*`.h` by default, unless you specify `Data declared in single separate header file` for the **Data declaration** option on the **Real-Time Workshop > Data Placement** pane of the Configuration Parameter dialog box. If you select that data placement option, the code generator places declarations in `global.h`. If you specify a definition filename for each data object, the code generator generates one definition `.c/.cpp` file for each data object and places declarations in *`model`*`.h` by default, unless you specify `Data declared in single separate header`

`file` for **Data declaration**. If you select that data placement option, the code generator places declarations in`global.h`.

Note If you generate C++ rather than C code, the `.c` files listed in the following tables will be `.cpp` files.

Data Placement Rules

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
mpt or Simulink Noncustom Storage Classes:								
`auto`	N/A	N/A	N/A	N/A	N/A	Note 12	`model.h`	Note 1
`Exported- Global`	N/A	N/A	N/A	N/A	N/A	`model.c`	`model.h`	Note 1
`Imported- Extern`, `Imported- Extern- Pointer`	N/A	N/A	N/A	N/A	N/A	None. External	`model_ private.h`	Note 2
`Simulink- Global`	N/A	N/A	N/A	N/A	N/A	Note 13	`model.h`	Note 1
mpt or Simulink Custom Storage Class: Imported Data:								
`Imported- FromFile`	D/C	D/C	D/C	N/A	null	None	`model_ private.h`	Note 3
`Imported- FromFile`	D/C	D/C	D/C	N/A	`hdr.h`	None	`model_ private.h`	Note 4
Simulink Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	N/A	N/A	`#define, model.h`	Note 5
mpt Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	null	N/A	`#define, model.h`	Note 5

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
`Define`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	`#define`, `model.h`	Note 6
mpt or Simulink Custom Storage Class: Get Set:								
`GetSet`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	External `hdr.h`	Note 4
mpt or Simulink Custom Storage Class: Bitfield, Struct:								
`Bitfield, Struct`	D/C	D/C	N/A	N/A	N/A	`model.c`	`model.h`	Note 7
mpt Custom Storage Class: Global, Const, ConstVolatile, Volatile:								
`Global, Const, Const- Volatile, Volatile`	`auto`	`auto`	null	null or locally owned	null	`model.c`	`model.h`	Note 1
`Global, Const, Const- Volatile, Volatile`	`src`	`auto`	null	null or locally owned	null	`src.c`	`model.h`	Note 1
`Global, Const, Const- Volatile, Volatile`	`sep`	`auto`	null	null or locally owned	null	`glb.c`	`model.h`	Note 1
`Global, Const, Const- Volatile, Volatile`	`auto`	`src`	null	null or locally owned	null	`model.c`	`src.c`	Note 8
`Global, Const, Const- Volatile, Volatile`	`src`	`src`	null	null or locally owned	null	`src.c`	`src.c`	Note 8
`Global, Const, Const- Volatile, Volatile`	`sep`	`src`	null	null or locally owned	null	`glb.c`	`src.c`	Note 8

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
`Global, Const, Const- Volatile, Volatile`	`auto`	`sep`	null	null or locally owned	null	`model.c`	`glb.h`	Note 9
`Global, Const, Const- Volatile, Volatile`	`src`	`sep`	null	null or locally owned	null	`src.c`	`glb.h`	Note 9
`Global, Const, Const- Volatile, Volatile`	`sep`	`sep`	null	null or locally owned	null	`glb.c`	`glb.h`	Note 9
`Global, Const, Const- Volatile, Volatile`	D/C	D/C	data.c	D/C	null	`data.c`	See Note 10.	Note 10
`Global, Const, Const- Volatile, Volatile`	D/C	D/C	data.c	D/C	`hdr.h`	`data.c`	`hdr.h`	Note 11
`Global, Const, Const- Volatile, Volatile`	`auto`	D/C	null	null	`hdr.h`	`model.c`	`hdr.h`	Note 11
`Global, Const, Const- Volatile, Volatile`	`src`	D/C	null	null	`hdr.h`	`src.c`	`hdr.h`	Note 11
`Global, Const, Const- Volatile, Volatile`	`sep`	D/C	null	null	`hdr.h`	`glb.c`	`hdr.h`	Note 11

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
`Global, Const, Const-Volatile, Volatile`	D/C	`auto`	null	External owner	null	External user-supplied file	`model.h`	Note 1
`Global, Const, Const-Volatile, Volatile`	D/C	`src`	null	External owner	null	External user-supplied file	`src.c`	Note 8
`Global, Const, Const-Volatile, Volatile`	D/C	`sep`	null	External owner	null	External user-supplied file	`glb.h`	Note 9
`Global, Const, Const-Volatile, Volatile`	D/C	D/C	null	External owner	`header.h`	External user-supplied file	`hdr.h`	Note 11
`Global, Const, Const-Volatile, Volatile`	D/C	D/C	null	External owner	`header.h`	External user-supplied file	`hdr.h`	Note 11
mpt Custom Storage Class: Exported Data:								
`ExportTo-File`	`auto`	`auto`	null	null	null	`model.c`	`model.h`	Note 1
`ExportTo-File`	`src`	`auto`	null	null	null	`src.c`	`model.h`	Note 1
`ExportTo-File`	`sep`	`auto`	null	null	null	`glb.c`	`model.h`	Note 1

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
ExportTo-File	auto	src	null	null	null	model.c	src.c	Note 8
ExportTo-File	src	src	null	null	null	src.c	src.c	Note 8
ExportTo-File	sep	src	null	null	null	glb.c	src.c	Note 8
ExportTo-File	auto	sep	null	null	null	model.c	glb.h	Note 9
ExportTo-File	src	sep	null	null	null	src.c	glb.h	Note 9
ExportTo-File	sep	sep	null	null	null	glb.c	glb.h	Note 9
ExportTo-File	D/C	D/C	data.c	null	null	data.c	See Note 10.	Note 10
ExportTo-File	D/C	D/C	data.c	null	hdr.h	model.c	hdr.h	Note 11
ExportTo-File	auto	D/C	null	null	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	null	null	hdr.h	glb.c	hdr.h	Note 11
Simulink Custom Storage Class: Default, Const, ConstVolatile, Volatile:								
Default, Const, Const-Volatile, Volatile	auto	auto	N/A	N/A	N/A	model.c	model.h	Note 1

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
`Default, Const, Const-Volatile, Volatile`	`src`	`auto`	N/A	N/A	N/A	`src.c`	`model.h`	Note 1
`Default, Const, Const-Volatile, Volatile`	`sep`	`auto`	N/A	N/A	N/A	`glb.c`	`model.h`	Note 1
`Default, Const, Const-Volatile, Volatile`	`auto`	`src`	N/A	N/A	N/A	`model.c`	`src.c`	Note 8
`Default, Const, Const-Volatile, Volatile`	`src`	`src`	N/A	N/A	N/A	`src.c`	`src.c`	Note 8
`Default, Const, Const-Volatile, Volatile`	`sep`	`src`	N/A	N/A	N/A	`glb.c`	`src.c`	Note 8
`Default, Const, Const-Volatile, Volatile`	`auto`	`sep`	N/A	N/A	N/A	`model.c`	`glb.h`	Note 9
`Default, Const, Const-Volatile, Volatile`	`src`	`sep`	N/A	N/A	N/A	`src.c`	`glb.h`	Note 9

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion Approach**
`Default`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`sep`	N/A	N/A	N/A	`glb.c`	`glb.h`	Note 9
Simulink Custom Storage Class: Exported Data:								
`ExportTo-File`	`auto`	`auto`	N/A	N/A	null	`model.c`	`model.h`	Note 1
`ExportTo-File`	`src`	`auto`	N/A	N/A	null	`src.c`	`model.h`	Note 1
`ExportTo-File`	`sep`	`auto`	N/A	N/A	null	`glb.c`	`model.h`	Note 1
`ExportTo-File`	`auto`	`src`	N/A	N/A	null	`model.c`	`src.c`	Note 8
`ExportTo-File`	`src`	`src`	N/A	N/A	null	`src.c`	`src.c`	Note 8
`ExportTo-File`	`sep`	`src`	N/A	N/A	null	`glb.c`	`src.c`	Note 8
`ExportTo-File`	`auto`	`sep`	N/A	N/A	null	`model.c`	`glb.h`	Note 9
`ExportTo-File`	`src`	`sep`	N/A	N/A	null	`src.c`	`glb.h`	Note 9
`ExportTo-File`	`sep`	`sep`	N/A	N/A	null	`glb.c`	`glb.h`	Note 9
`ExportTo-File`	`auto`	D/C	N/A	N/A	`hdr.h`	`model.c`	`hdr.h`	Note 11

Data Placement Rules (Continued)

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	Data Def.	Data Dec.	Def. File	Owner	Header File	Where Data Def. Is	Where Data Dec. Is	Dec. Inclusion Approach
ExportTo-File	src	D/C	N/A	N/A	hdr.h	src.c	hdr.h	Note 11
ExportTo-File	sep	D/C	N/A	N/A	hdr.h	glb.c	hdr.h	Note 11

Notes

In the previous table:

- A Declaration Inclusion Approach is a file in which the header file that contains the data declarations is included.
- D/C stands for don't care.
- Dec stands for declaration.
- Def stands for definition.
- gbl stands for global.
- hdr stands for header.
- N/A stands for not applicable.
- null stands for field is blank.
- sep stands for separate.

Note 1: model.h is included directly in all source files.

Note 2: model_private.h is included directly in all source files.

Note 3: extern is included in model_private.h, which is in source.c.

Note 4: header.h is included in model_private.h, which is in source.c.

Note 5: `model.h` is included directly in all source files that use `#define`.

Note 6: `header.h` is included in `model.h`, which is in source files that use `#define`.

Note 7: `model.h` is included in all `source.c` files.

Note 8: `extern` is inlined in source files where data is used.

Note 9: `global.h` is included in `model.h`, which is in all source files.

Note 10: When you specify a definition filename for a data object, no header file is generated for that data object. The code generator declares the data object according to the data placement priorities.

Note 11: `header.h` is included in `model.h`, which is in all source files.

Note 12: Signal: Either not defined because it is expression folded, or local data, or defined in a structure in `model.c`, all depending on model's code generation settings. Parameter: Either inlined in the code, or defined in `model_data.c`.

Note 13: Signal: In a structure that is defined in `model.c`. Parameter: In a structure that is defined in `model_data.c`.

Index

U

V

W