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

Printing History: June 2004 Online only Version 4.0 (Release 14)

Contents

1 Getting Started

2 Selecting and Defining Templates

3 Managing Data Dictionary

Referenced Tables

A

Index

1

Getting Started

Introduction

The Real-Time Workshop® Embedded Coder generates C code for a Simulink® or Stateflow® model. This document discusses the module packaging features (MPF) of Real-Time Workshop Embedded Coder.

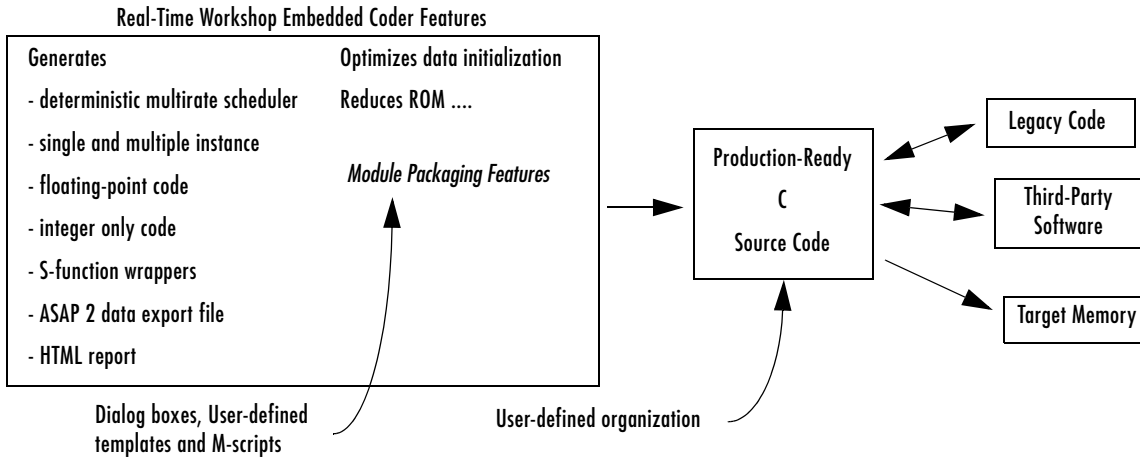
With MPF, you can

- Package the generated code into the desired number of `.c` and `.h` files.
- Control the *internal* organization of each of the generated files. For example, for readability, your company may have software standards that define where to place comments and sections of code within files.
- Control whether or not the generated files contain definitions for a model's global identifiers. 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 to meeting such packaging needs, MPF allows you to implement these and other features to meet your needs:

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

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



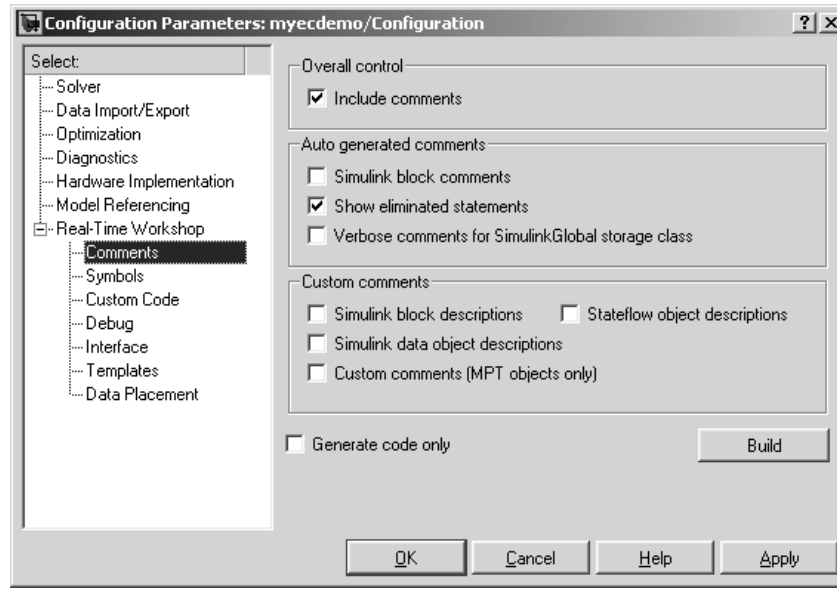
Module Packaging Features in Code-Generation Process

Note 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 model that is a part of a multimodel system. Using MPF, one user generates code for one model at a time. The term "packaging" refers to the ability to organize files.

Note When this document refers to a variable, it follows the distinction made in 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` file that references that variable. A variable is defined in a `.c` file.

Opening Model with Desired Settings

Each model you open has certain settings associated with it that are indicated on several panes of the **Configuration Parameters** dialog box, like that shown below.



Note While the model and **Configuration Parameters** dialog box are open, you can save the **Configuration Parameters** dialog box settings by saving the model in Simulink. When you reopen that model, it will have these settings.

Select from the following to open the **Configuration Parameters** dialog box with the desired settings:

- "Opening an Empty Model with Initial MPF Settings" on page 1-5
- "Opening an Empty Model That Uses Settings from Another Model" on page 1-6
- "Opening an Existing Model That Uses Saved MPF Settings" on page 1-7

Opening an Empty Model with Initial MPF Settings

This procedure opens an empty model and explains how to configure settings on the **Configuration Parameters** dialog box so that all module packaging features are available:

1. Start MATLAB.
2. Click the **Simulink** button on the toolbar. The **Simulink Library Browser** opens.
3. From the **File** menu, open an empty model using the **New** command. In this case, the settings on the **Configuration Parameters** dialog box are the default settings. To make all of the module packaging features available, you must change some of these default settings, by following the steps below. Most users should do this.
4. From the **Simulation** menu, click **Configuration Parameters**., or use the short cut **Ctrl+E**.
5. On the left pane, select **Optimization**. The **Optimization** pane appears on the right.
6. Ensure that the **Inline parameters** check box is selected, and then click the **Apply** button if it is available.
7. Click **Solver** on the left pane.
8. In the **Type** field, ensure that `Fixed-step` is selected.
9. Click **General** under **Real-Time Workshop** on the left pane. The **General** pane appears on the right.
10. In the **RTW system target file** field, select the appropriate `ert.tlc`., and click **OK**.
11. Ensure that the **Ignore custom storage classes** check box is cleared.
12. Click **Comments** under **Real-Time Workshop** on the left pane. The **Comments** pane appears on the right.

13 Ensure that the **Include comments** check box is selected, and then click the **Apply** button if it is available.

Now you can start a new procedure by going to "Selecting the Desired MPF Procedure" on page 1-8, or return to a procedure.

Opening an Empty Model That Uses Settings from Another Model

This procedure opens the initial **Configuration Parameters** dialog box for an empty model. The settings are those transferred from another model:

1 Start MATLAB.

2 Click the **Simulink** button on the toolbar. The **Simulink Library Browser** opens.

3 On the **File** menu, open an empty model using the **New** command.

4 On the **File** menu, open an existing model that has the MPF settings you want to transfer to the empty model.

5 From the **Tools** menu, click **Model Explorer**. Both model names appear in the left pane.

6 Expand the name of the existing model in the left pane.

7 Select **Configuration**, and drag and drop it onto the name of the new model in the left pane.

8 Expand the name of the new model in the left pane. Notice there are two configurations. The one you just copied is on the bottom.

9 Right click the **Configuration** on the bottom and select `Active`. The new model now has the settings of the existing model.

10 If desired, change the name of the new **Configuration** on the right pane in the **Name** field.

11 Accept or edit the remaining settings on the right pane as desired.

12 From the **File** menu, click **Save As** to save the empty model and its MPF settings with the desired filename and directory path.

Proceed to "Selecting the Desired MPF Procedure" on page 1-8, or return to a procedure.

Opening an Existing Model That Uses Saved MPF Settings

This procedure opens an existing model. The settings on the **Configuration Parameters** dialog box are those made in the previous session for this model, even if MATLAB was closed:

1 Start MATLAB.

2 On the **File** menu, open the desired existing model using the **Open** command.

Selecting the Desired MPF Procedure

The following chapters document the MPF procedures. Each procedure has an explanation, followed by the steps to implement it.Follow the desired procedure:

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

Note Some MPF settings are interdependent. These are identified in Chapter 5, “Managing File Placement of Data Definitions and Declarations.” That chapter explains how these interdependent MPF settings manage file placement of data definitions and declarations, their priorities and frequencies of use.

2

Selecting and Defining Templates

Introduction

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`.

Table 2-1: Generated Files and Templates That Organize Them

Generated File	example_ banner. cgt	code_c_ template. cgt	code_h_ template. cgt	data_c_ template._ cgt	data_h_ template. cgt	example_ file_ process_ template. 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_date.h` file	x				x	x

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

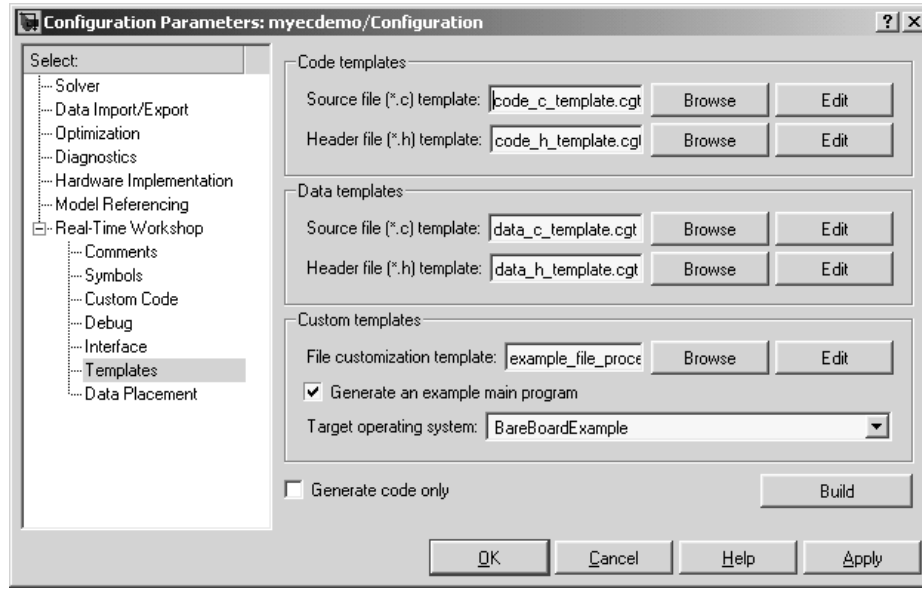
A **Code template** organizes all of the generated files that, primarily, contain functions but not identifiers. The *source* code template organizes C code files. These include, for example, the main `.c` or any of the `.c` files that contain functions that Real-Time Workshop Embedded Coder generates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

one 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 2-9 describes how to create your own code or data templates.

For details describing the custom template, see the discussion of Custom File Processing Templates in the Real-Time Workshop Embedded Coder documentation.

Selecting Preexisting Templates

Fields on the **Templates** pane allow you to specify template files that Real-Time Workshop Embedded Coder will use to organize all of the generated `.c` or `.h` files:

1. From the **Simulation** menu, click **Configuration Parameters**.
2. Click **Templates** under **Real-Time Workshop** on the left pane. The templates pane now appears on the right. For an explanation of fields on this pane, see Table A-1, MPF Elements on Configuration Parameters Panes, 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\mpt\mpt`. 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. The supplied MPF source file template is `code_c_template.cgt`. Real-Time Workshop Embedded Coder will use this file to organize the `.c` file or files that contain the source code for the model's functions.
4. In the **Header file (*.h) template** field of the **Code templates** pane, select the desired filename. The supplied MPF header file template is `code_h_template.cgt`. Real-Time Workshop Embedded Coder will use this file to organize the `.h` header file that contains the model's function prototypes.
5. In the **Source file (*.c) template** field of the **Data templates** pane, select the desired filename. The supplied MPF source file template is `data_c_template.cgt`. Real-Time Workshop Embedded Coder will use this file to organize the `.c` file that contains the definitions of variables of global scope.
6. In the **Header file (*.h) template** field of the **Data templates** pane, select the desired filename. The supplied MPF header file template is

`data_h_template.cgt`. Real-Time Workshop Embedded Coder will use this file to organize the `.h` file that contains declaration statements (`extern`, `typedef`, `#define`).

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** would save the choices but close the pane.)

Now you can generate 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. In 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.

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, 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**.

Note The generate code process generates the `.c` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see "Steps in the 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 2-9.

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`. A default path to templates is created during installation in the MATLAB folder: `toolbox\rtw\targets\mpt\mpt`. 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 "Example Template and Its Generated File" on page 2-10:

1. If the **Templates** pane of the **Configuration Parameters** dialog box is not open, open it by selecting **Configuration Parameters** on the **Simulation** menu, and then selecting **Templates** on the left pane. The **Templates** pane now appears on the right, like that shown in "Introduction" 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. Table 2-1, 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. Table A-1, MPF Elements on Configuration Parameters Panes, on page A-2, describes the supplied code 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 Table 2-1, 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 tables:

- Table A-2, Template Symbols, on page A-13
- Table A-3, Parent-Child Relationships of Template Symbols, on page A-19
- Table A-4, Rules for Modifying or Creating a Template, on page A-21

Note In the next step, 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**.

4. Perform a **Save** or **Save As** operation, naming the template file as desired.
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. The paragraphs below provide helpful guidelines.

Example Template and Its Generated File

"Section of Template File" on page 2-11 below shows a portion of an example `.c` code template file. "Corresponding Section of Generated File" on page 2-12 shows the corresponding portion of the `.c` file Real-Time Workshop Embedded Coder generated using this template.

Notice `%<FileName>` on the template. This illustrates how a symbol name is placed on a template. The term `FileName` is a symbol name. Every symbol name must be enclosed by a percent sign and brackets: `%< >`. You can add the desired symbol name (within the `%< >` delimiter) at a particular location on the template. This is how you control where an item will be located on the

generated source file. For example, notice the filename `control_logic.c` in the `.c` file (Generated File). Notice that this is located in the `.c` file where `%<FileName>` is located on the template (the first figure).

Table A-2, Template Symbols, on page A-13, identifies all provided symbols. Table A-3, Parent-Child Relationships of Template Symbols, on page A-19, shows all symbol names according to their symbol group and classifies the symbol names into their obvious groupings. Table A-4, Rules for Modifying or Creating a Template, on page A-21, explains the rules you must follow when placing symbols on a template.

Section of Template File

The listing below is part of an example `.c` code template file. Compare this with `control_logic.c` in "Corresponding Section of Generated File" on page 2-12.

```
/**
**   FILE INFORMATION:
**   Filename:           %<FileName>
**   File Creation Date: %<Date>
**
**   ABSTRACT
**   %<Abstract>v
**
**   NOTES:
**   %<Notes>
**
**   MODEL INFORMATION:
**   Model Name:         %<ModelName>
**   Model Descripiton   %<Description>
**   Model Version:      %<ModelVersion>
**   Model Author        %<Creator> - %<Created>
**
**   MODIFICATION HISTORY:
**   Model at Code Generation: %<ModifiedBy> - %<ModifiedDate>
.
.
.
```

Corresponding Section of Generated File

The listing below is part of a file named `control_logic.c`. Compare with the `.c` code template file shown in "Section of Template File" on page 2-11.

```
/**
**   FILE INFORMATION:
**   Filename:           control_logic.c
**   File Creation Date: 7-Jul-2004
**
**   ABSTRACT
**   This is the abstract for the model.
**
**   NOTES:
**   This is a note from Simulink.
**
**   MODEL INFORMATION:
**   Model Name:         control_logic
**   Model Descripiton
**   Model Version:      1.195
**   Model Author        Smith - Fri Jul 2 13:29:52 2004
**
**   MODIFICATION HISTORY:
**   Model at Code Generation: Jones - Wed Jul 07 11:12:42 2004
.
.
.
.
```

3

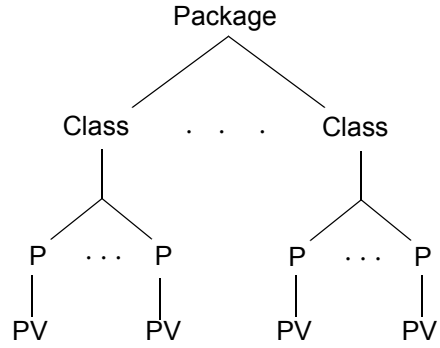
Managing Data Dictionary

Introduction

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 the code generator creates for Simulink and Stateflow models is called the code generation data dictionary. It is the total number of data objects that appear in the middle pane of the **Model Explorer** dialog box. These data objects also appear in the MATLAB workspace. This procedure 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.

Property value settings also can affect where the code generator places a parameter or signal in the generated file. This is because property values are associated with different template symbols. The location of a symbol on 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 the **Model Explorer** dialog box. In Simulink, there is a hierarchy of terms that are drawn from object-oriented programming. For details, see "Working with Data Objects" in the Using Simulink documentation. The sketch below summarizes this hierarchy.



P = Property

PV = Property Value

In our context, `mpt` is the package. There are only two classes in this package: Parameter and Signal. Each class has a number of properties associated with it (parameter properties in the parameter class and signal properties in the signal class). Sometimes properties are called *attributes*. Simulink data objects (the parameters and signals) are the instances of a `"package.class"` that make up the data dictionary. All parameter data objects share a single set of properties. All signal data objects share a different single set of properties. (In our case, the two sets are almost the same.) 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 Simulink signal or Stateflow data.

There are five main subprocedures in the Managing Data Dictionary process. Do these in the order listed below:

- "Registering User Data Types" on page 3-4
- "Registering User Object Types" on page 3-10
- "Adding Simulink Data Objects to the Dictionary" on page 3-11
- "Applying Naming Rules to Identifiers Globally" on page 3-19
- "Inspecting Code and Editing the Dictionary" on page 3-27

Registering User Data Types

You must either accept the MathWorks default data types or register user-defined data types. If you have user-defined data types, you must register these so that the code generator can associate them with the corresponding MathWorks data types. Then, the code generator will use your user data types in the generated source code instead of the MathWorks data types.

This procedure registers user data types by placing function calls in the `custom_user_type_registration.m` file, whose arguments specify information about the user data type and its associated MathWorks data type:

1. Look at the MathWorks data types listed in Table 3-1, Equivalent Data Types (User To Fill In), on page 3-7. 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 3-10. Otherwise, go to the next step. (For explanations of primary and secondary user data types, see Table 3-2, Arguments To Place in utype_register Function Call, on page 3-8.)

2. 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.

3 Place the pointer on a new line after the line that reads `utype_establish;`. As instructed below, you must register every user data type to be added to the dictionary by a call to the M-function `utype_register`. The arguments in this function call must be provided in the prescribed left-to-right order and syntax. See the figure below for an example and Table 3-2, Arguments To Place in utype_register Function Call, on page 3-8 for an explanation of each argument. Further, as instructed in the next step, you must know the names of all of the user data types to be entered in the dictionary, and The MathWorks data types to which each is equivalent. See Table 3-1, Equivalent Data Types (User To Fill In), on page 3-7.

Example of call to the function that registers a user data type. See Table 3-2, Arguments To Place in utype_register Function Call, on page 3-8.

```
utype_establish;
utype_register('real32_T', 'userfloat', 'primary', '<userdata_types.h>');
```

Function name

MathWorks data type*

User data type*

primary: User data type is primary

secondary: User data type is secondary

#include .h File

Use the ANSI C angle-bracket or double-quote delimiter.

* Equivalent data types

4 Make a copy of Table 3-1, Equivalent Data Types (User To Fill In), on page 3-7 and, in the user-defined data types columns, list all of your data types. You can associate as many user-defined data types with a single MathWorks data type as you want. Be sure to place each in its proper row, beside the equivalent MathWorks data type.

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

---

5 Type a `utype_register` function call for every user data type that you listed in the table mentioned in the previous step.

6 Ensure that the `.h` file that contains the `typedef` statements for each user data type is in the appropriate directory. (That is, appropriate to the chosen 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 in a user-specified folder that is on the MATLAB path. This path must be above `toolbox\rtw\targets\mpt\user_specific` in the MATLAB search path. The user data types are registered. Now these user data types are available for you to select in the **Data type** field of the **Model Explorer** dialog box for a signal. For details on how to apply user data types in the generated code, see "Applying User Data Types to Signal Objects in the Generated Code" on page 3-24.

Table 3-1: Equivalent Data Types (User To Fill In)

MathWorks Data Type	Primary User-Defined Data Type	Secondary User-Defined Data Types
`boolean_T`		
`int8_T`		
`int16_T`		
`int32_T`		
`real32_T`		
`real_T`		
`uint8_T`		
`uint16_T`		
`uint32_T`		

Table 3-2: Arguments To Place in utype_register Function Call

Argument (Left to Right) To Place in utype_register Function Calls	Description
MathWorks Data Type	The MathWorks data type, listed in Table 3-1, Equivalent Data Types (User To Fill In), on page 3-7, that is equivalent to the specified user data type.
User Data Type	The user-defined data type that is equivalent to a specified MathWorks data type specified in the function call. The name must conform to ANSI C rules. The name may contain any combination of uppercase or lowercase characters, decimal digits 0 to 9, or the underscore character. The identifier must start with a letter or underscore character. You cannot use an ANSI C reserved word. For readability, we recommend that the chosen characters of the identifier suggest what it is.

Table 3-2: Arguments To Place in utype_register Function Call (Continued)

Argument (Left to Right) To Place in utype_register Function Calls	Description
`primary or secondary`	Specifies whether the user data type indicated in the function call is primary or secondary. There can be one primary user data type for the equivalent MathWorks data type, and as many secondary user data types as you want. If primary, during code generation Real-Time Workshop Embedded Coder will use this data type instead of the MathWorks data type • For data type conversion (that is, for a cast operation specified in the code) • For data definition and declaration A secondary is another user data type that maps to the same MathWorks data type as the primary does. It allows you to have multiple aliases for the same MathWorks data type. A user data type specified as secondary is used in definition and declaration statements but not in a cast operation. If there is only one user data type for a MathWorks data type, you must specify `primary` and include one `utype_register` function call. If a user data type also has one or more secondary names, you must specify each as `secondary` in as many separate `utype_register` function calls. The terms `primary` and `secondary` are case insensitive.
`#include .h` File	Specifies the user's prewritten `.h` file that includes the `typedef` statement for the indicated user data type. Place between the ANSI C angle-brackets or double-quotes delimiter. The directory that each delimiter indicates varies by compiler and is explained in the compiler documentation. Usually, however (but not always), the angle brackets indicate that the compiler will find the `.h` file in the default directory where the standard C library is located (the "source" folder).

Registering User Object Types

This procedure registers one or more fixed sets of user-defined property values that can be applied to user 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 "Setting Property Values" on page 3-15 for each of these data objects.

You register the set of property values by placing function calls in a file named `custom_user_object_type_info.m`. You can register as many object types in the file as you want. Their unique names will be selectable in the **User object type** field on the **Model Explorer** dialog box. 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** dialog box when you select that name:

1. Using a text editor, create a file named `custom_user_object_type_info.m`.
2. In the file, register a unique data object name and the desired set of properties and property values you want associated with this name. (An example `custom_user_object_type_info.m` file, which contains instructions, is in the `matlab/toolbox/rtw/targets/mpt/mptdemos` directory.)
3. Save the file on the MATLAB path.
4. Repeat steps 1 through 3 for any remaining user object types you want to register.

Proceed to "Adding Simulink Data Objects to the Dictionary" on page 3-11.

Adding Simulink Data Objects to the Dictionary

Now you can add data objects to the code generation data dictionary. "Data objects" refers to the model's parameters and signals. All of the model's data objects must be in the dictionary. (If you have no user-defined data objects to import into the dictionary, start with "Adding Simulink Data Objects with Data Object Wizard" on page 3-13.)

Importing External Data Objects

This procedure imports into the code generation data dictionary any user-defined data objects (and their property values) that are in

- A `.mat` file from a previous Simulink session
- An external data dictionary (such as an Excel file)

Follow either of these two procedures below, as applicable.

Loading Data Objects from .mat File

On the MATLAB command line, type `load filename`, where `filename` is the name of the `.mat` file. This file must be on the MATLAB path or in the current directory. The data objects from the file are loaded into the MATLAB workspace. (The data objects in the workspace constitute the code generation data dictionary.)

Proceed to "Adding Simulink Data Objects with Data Object Wizard" on page 3-13.

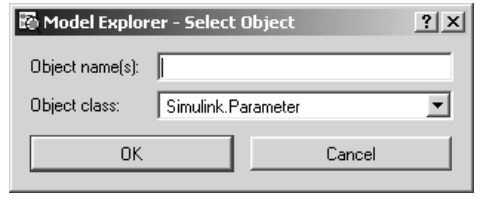
Creating Data Objects Based on External Data Dictionary

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

2. Determine all of the data objects in this file that correspond to the parameters and signals in the model. (See "Introduction" on page 3-2 for an explanation of a model's parameters and signals.) 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.

Note The steps below use a dialog box to create data objects. Instead, you can type `name = mpt.Signal` or `name = mpt.Parameter` on the MATLAB command line. The `name` is the name of each data object in your file that is in the `mpt.Signal` and `mpt.Parameter` class, respectively.

3. On the MATLAB command line, type `daexplr` and press **Enter**. The **Model Explorer** dialog box appears.
4. On the **Model Hierarchy** (left) pane, expand **Simulink Root**, and select **Base Workspace**.
5. On the **Add** menu, select **Custom**. The **Select Object** dialog box appears**,** like that shown below.



6. In the **Object class** field, click the down arrow and select `mpt.Parameter`.
7. In the **Object name(s)** field, type the names of all of the data objects in your file that are in the `mpt.Parameter` class, separating them by commas.
8. Click **OK**. The data objects for the parameter class appear in the middle pane of the **Model Explorer** dialog box, and therefore have been created in the code generation data dictionary.
9. Repeat steps 6 through 8 for all of the data objects in your file that are in the `mpt.Signal` class.

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

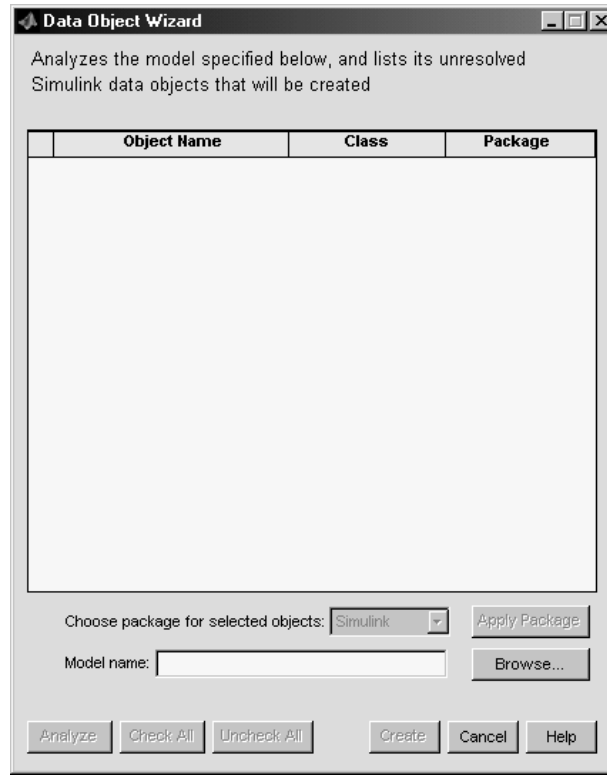
Now you need to add missing Simulink data objects.

Adding Simulink Data Objects with Data Object Wizard

This procedure places all missing data objects (that you select) in the code generation data dictionary. If you imported data objects from an external dictionary, this procedure adds the *remaining* data objects required for code generation. If you did not import data objects from an external dictionary, this procedure adds all of the selected data objects. Most of the property values of data objects are supplied by defaults. A few are from the model:

1. If the model whose data objects you want to add to the data dictionary is not open, follow “Opening Model with Desired Settings” on page 1-4.

2. On the MATLAB command line, type `dataobjectwizard`. 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** button, or navigate to it using the **Browse** button. The **Analyze** button becomes available.

4 Click the **Analyze** button. After a moment, a list of all of the model's data objects appear that are not yet in the code generation data dictionary. This includes all of the model's signals, parameters, data stores, and discrete states. Each data store and discrete state is listed as a signal class. (The following Simulink blocks are supported: Discrete Filter, Discrete State-Space, Discrete-Time Integrator, Discrete Transfer Fcn, Discrete Zero-Pole, Memory, and Unit Delay.)

5 If you want to associate one or more data objects with the `mpt` package, select each of those data objects. (Clicking **Check All** selects all data objects.) Otherwise, go to step 8.

6 In the **Choose package for selected objects** field, select `mpt`.

7 Click **Apply Package**. All selected data objects are associated with the `mpt` package, as indicated in the **Package** column.

8 If you want to associate one or more data objects with the `Simulink` package, repeat steps 5 through 7, except select `Simulink` in the **Choose package for selected objects** field.

Note After doing the next step, a **Data Object Wizard** message could appear for a parameter data object. If you click the **Do not ask again** button on that message, it will not appear again during the current MATLAB session.

---

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

10 Repeat steps 3 through 9 for any other model or models.

11 Click **Cancel**. The **Data Object Wizard** disappears. Now you must set property values for the data objects.

Setting Property Values

All of the model's selected data objects are in the dictionary. Default property values are supplied automatically. You can either accept the default value or change a property value for each property of each data object.

Note The **Alias** property is related to "Applying Naming Rules to Identifiers Globally" on page 3-19. As explained in that procedure, for an mpt data object (identifier), selecting the **Alias overrides naming rule** check box causes the name you specify in the **Alias** field to override the naming rule selection you make on the **Configuration Parameters** dialog

box. But for a Simulink data object, selecting other than `None` in a naming rule field overrides **Alias** on the **Model Explorer** regardless of whether or not you specify an **Alias** name. (The **Alias overrides naming rule** check box is not available for a Simulink data object.)

---

1 If the **Model Explorer** dialog box is not open, open it by typing `daexplr` on the MATLAB command line and pressing **Enter**.

Note At the top of the **Model Explorer** dialog box, ensure that the **Search** button is available. If it is not, click the **Done** button in the middle pane to make the **Search** button available. Otherwise, the data objects required in the following steps may not appear in the middle pane.

In the **Model Hierarchy** (left) pane, select **MATLAB Workspace**. All data objects in the code generation data dictionary appear in the **Contents of** (middle) pane.

2 Click a data object in the middle pane. The properties for this data object appear in the right pane. The name of this pane is either **mpt.Parameter** or **mpt.Signal**, depending on the data object you selected. The figure below shows properties for `mpt.Parameters`. The properties for `mpt.Signals` are almost the same. Table A-5, Parameter and Signal Property Values, on page A-24 defines properties for both selections.

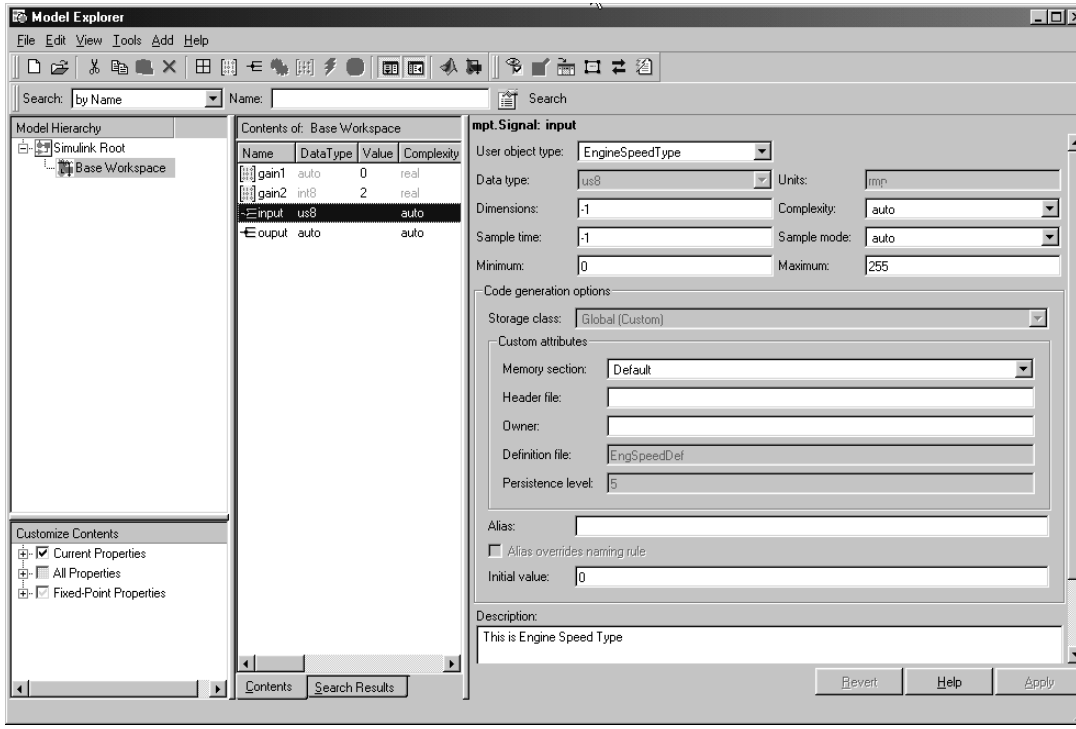
3 For each property (dialog box element) in the **mpt.Parameter** or **mpt.Signal** pane, either accept the default or specify the desired property value. Consult as reference, Table A-5, Parameter and Signal Property Values, on page A-24.

Note Some selections in the **Storage class** field have additional fields associated with them on the **Model Explorer** dialog box. To ensure that you see the exact fields that are associated with the chosen selection, make the selection in this field and then click the **Apply** button.

---

4 Click the **Apply** button.

5 Repeat steps 3 and 4 for the model's remaining data objects.


Model Explorer
File Edit View Tools Add Help
Search: by Name
Name:
Search
Model Hierarchy
Simulink Root
Base Workspace
Customize Contents
Current Properties
All Properties
Fixed-Point Properties
Contents of: Base Workspace
Name DataType Value Complexity
gain1 auto 0 real
gain2 int8 2 real
input us8 auto
ouput auto auto
Contents
Search Results
mpt.Signal: input
User object type: EngineSpeedType
Data type: us8
Units:
Dimensions: -1
Complexity: auto
Sample time: -1
Sample mode: auto
Minimum: 0
Maximum: 255
Code generation options
Storage class: Global (Custom)
Custom attributes
Memory section: Default
Header file:
Owner:
Definition file: EngSpeedDef
Persistence level: 5
Alias:
Alias overrides naming rule
Initial value: 0
Description:
This is Engine Speed Type
Revert
Help
Apply

Applying Naming Rules to Identifiers Globally

Signal names and parameter names appear on the model. The same names appear as data objects on the **Model Explorer** dialog box. These names can be spelled exactly the same way when they appear as identifiers in the generated source code. For example, `"Speed"` on the model (and workspace) can be spelled `"Speed"` as an identifier in the code. But you can change how they appear in the code. For example, if desired, you can change `"Speed"` to `SPEED`, `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 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 the spelling of *all* of the names when code generation occurs, according to the same rule. Or, you can change the spelling of names *individually* by following “Setting Property Values” on page 3-15. (The relevant property in that procedure is **Alias** on the **Model Explorer**.)

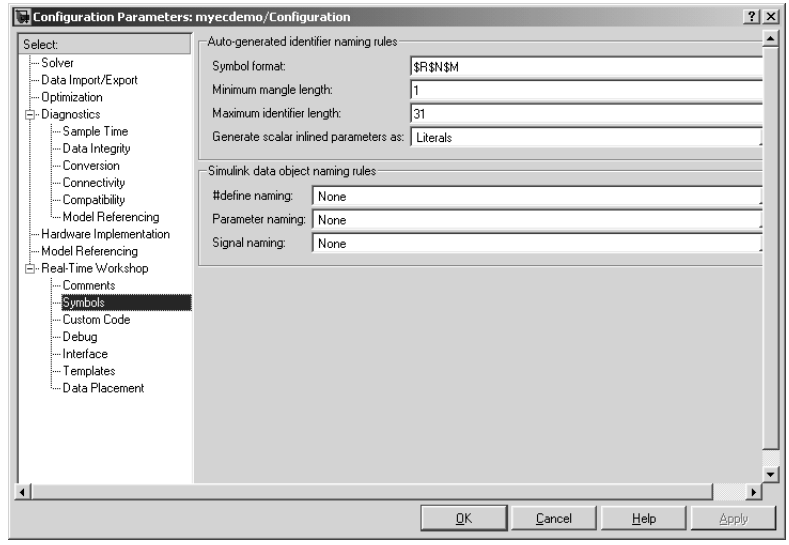
The naming rules versus **Alias** override works differently, depending on whether the data object is an mpt data object or a Simulink data object. For an mpt data object, clearing the **Alias overrides naming rule** check box on the **Model Explorer** causes the naming rule selection to be in effect, whether or not you have specified a name in the **Alias** field. But specifying an **Alias** name and selecting this check box causes the name to be in effect for that data object.

For a Simulink data object, selecting a naming rule (that is, selecting other than `None` in any of the naming rule fields), causes the naming rule to be in effect for that data object. This is true whether or not you have specified a name in the **Alias** field for that data object. (The **Alias overrides naming rule** check box is not available for a Simulink data object.)

Open the model and click **Configuration Parameters** on the **Simulation** menu. Click **Symbols** under **Real-Time Workshop** on the left pane. The **Simulink data object naming rules** pane appears, as shown in the next figure. Notice the preconfigured settings on this pane. If all of these are acceptable as is, proceed to “Inspecting Code and Editing the Dictionary” on page 3-27. 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. Table A-1, MPF Elements on Configuration Parameters Panes, 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 3-20
- "Defining Rules That Change All Parameter Names" on page 3-21
- "Defining Rules That Change All Signal Names" on page 3-22



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 "Setting Property Values" on page 3-15, 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. (Table A-1, MPF Elements on Configuration Parameters Panes, 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 3-21.

2. Write a function in M-code that changes all occurrences of the parameter name whose storage class you specified as `Define` in "Setting Property Values" on page 3-15 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. (Table A-1, MPF Elements on Configuration Parameters Panes, 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 3-22.

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. (Table A-1, 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 inspect proceed to "Inspecting Code and Editing the Dictionary" on page 3-27.

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 3-21.)

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 proceed to "Inspecting Code and Editing the Dictionary" on page 3-27.

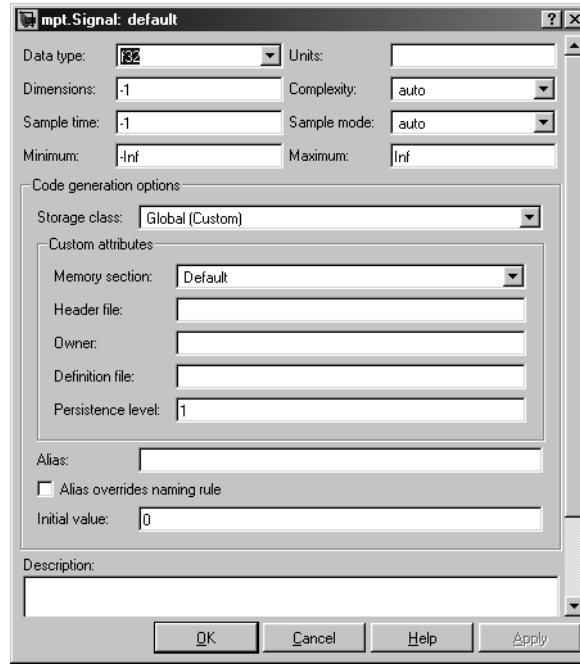
Applying User Data Types to Signal Objects in the Generated Code

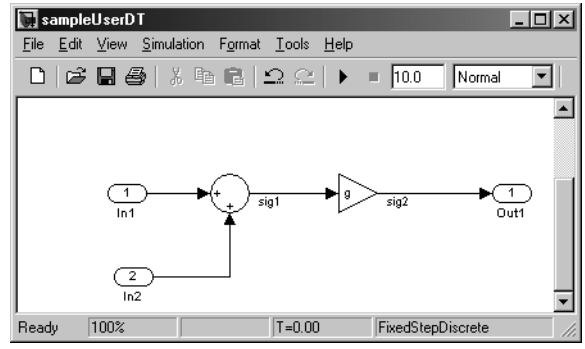
If you have registered user data types, as explained in "Registering User Data Types" on page 3-4, follow this procedure to ensure that they appear in the generated code. See the figures below:

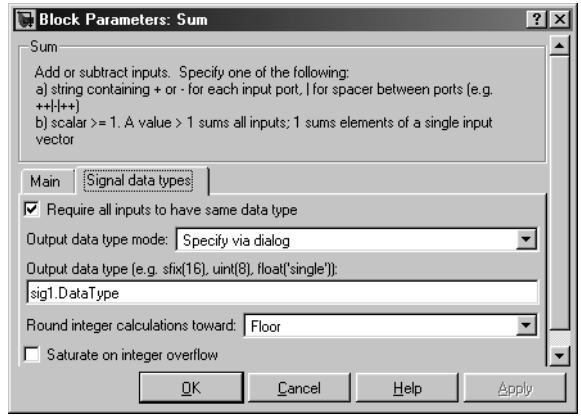
Note Step 1 creates all the `Simulink.AliasType` objects, that correspond to all registered data types, using a MATLAB command. As an alternative, you can create a `Simulink.AliasType` object one at a time using the **Simulink Alias Type** selection on the **Add** menu of the **Model Explorer** dialog box. 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. Alias Type" in the Simulink documentation.

---

1. Create all `Simulink.AliasType` objects by typing the MATLAB command, `ec_create_type_obj`. These `Simulink.AliasType` objects, which correspond to all registered user data types, appear in the MATLAB base workspace.
2. In your model, for a signal associated with a signal object, select the user data type in the **Data type** field on the **Model Explorer** dialog box.
3. Open the **Block Parameters** dialog box of the signal's source block.
4. Select the **Signal data types** tab.
5. In the **Output data type** field, type `object.DataType`, where object is the name of the signal object, and then click **Apply**.
6. Repeat steps 2 through 5 for each remaining signal associated with a signal object.
7. Save the model and save all of the data objects in the MATLAB base workspace in a `.mat` file.


mpt.Signal: default
Data type:
i32
Units:
Dimensions:
-1
Complexity:
auto
Sample time:
-1
Sample mode:
auto
Minimum:
-Inf
Maximum:
Inf
Code generation options
Storage class:
Global (Custom)
Custom attributes
Memory section:
Default
Header file:
Owner:
Definition file:
Persistence level:
1
Alias:
Alias overrides naming rule
Initial value:
0
Description:
OK
Cancel
Help
Apply


sampleUserDT
File Edit View Simulation Format Tools Help
10.0
Normal
1
In1
sig1
g
sig2
1
Out1
2
In2
Ready
100%
T=0.00
FixedStepDiscrete


Block Parameters: Sum
Sum
Add or subtract inputs. Specify one of the following:
a) string containing + or - for each input port, | for spacer between ports (e.g. ++|-|++)
b) scalar >= 1. A value > 1 sums all inputs; 1 sums elements of a single input vector
Main
Signal data types
Require all inputs to have same data type
Output data type mode:
Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
sig1.DataType
Round integer calculations toward:
Floor
Saturate on integer overflow
OK
Cancel
Help
Apply

Inspecting Code and Editing the Dictionary

All data objects for the model are in the code generation data dictionary. You have specified property values for each data object's properties. 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** on 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. On 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` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see "Steps in the 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. Click a `.c` file. The `.c` file opens in an editor.

6 Inspect the file to ensure that it is correct and is organized the way you want.

Note We advise that you practice making changes to property values, regenerate the code, and then notice how a `.c` file changed.

---

7 If desired, repeat steps 1 to 6 until the file contains the lines of code and their locations in the file that you want.

8 Repeat steps 1 through 7 for all remaining generated `.c` files and `.h` files listed in the HTML report.

4

Customizing with Additional Options

Ensuring Delimiter Is Specified for All #Includes

Understanding the purpose of this procedure requires understanding the `IncludeFile` property of a data object, described in Table A-5, Parameter and Signal Property Values, on page A-24, and performed in "Setting Property Values" on page 3-15. For a particular data object, you can specify as the `IncludeFile` 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 `IncludeFile` 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`, `"filename"`, or `<filename>`. The code generator finds every data object for which you specified a filename as its `IncludeFile` 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.

In the **#include file delimiter** field, select the `#include <header.h>` instead of the default `Use #include "header.h"`.

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. The selection is made in the **Source of initial values** field on the **Data Placement** pane of the **Configuration Parameters** dialog box. The code generator will place the appropriate initialization statement in the declaration section of the generated file. The identifiers that correspond to the model's signals then will be initialized to the desired values during the initialization phase when the program runs. There are two possible selections: `Model` and `Data object.` The default is `Model`:

- If you accept `Model`, the statement the code generator places in the file for each signal is `type identifier = zero;`, where `identifier` corresponds to the signal in the model, and `zero` is `0.0` or `0`, depending on the type. The `Model` selection always initializes the value to zero.
- If you select `Data object`, the statement the code generator places in the file is `type identifier = value;`, where `value` is the **Initial value** property value you entered for each signal data object in the **Model Explorer**. (See "Setting Property Values" on page 3-15.)

Note Follow "Managing Data Dictionary" on page 3-1 before proceeding with the steps below.

---

1. In the open model, click **Configuration Parameters** on the **Simulation** menu. The **Configuration Parameters** dialog box appears.
2. Click **Data Placement** under **Real-Time Workshop** on the left pane. The **Data Placement** pane appears on the right.
3. In the **Source of initial values** field, select `Model` or `Data object.`
4. Click the **Apply** button.

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 object 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 Table A-5, Parameter and Signal Property Values, on page A-24. This example comment contains some of the property values for the data object `MAP` as specified on the **Model Explorer** dialog box:

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

Note You can type text in the **Description** field on the **Model Explorer** dialog box 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 `custom_comments_example.m` is provided in the `matlab/toolbox/rtw/targets/mpt/mptdemos` directory. An example `.tlc`

file named `custom_comments_example.tlc` is provided in this directory. Each of these files contains instructions.

Note 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. See the instructions in the example `.m` or example `.tlc` file for details.

---

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. In the open model, select **Configuration Parameters** on the **Simulation** menu. The **Configuration Parameters** dialog box appears.
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 using the Simulink DocBlock so that this comment appears in the generated file where desired.

1. Drag the **DocBlock** from **Model-Wide Utilitites** in the Simulink library onto the model. See "DocBlock" in the Simulink documentation for details.

2. After double clicking the DocBlock and typing the desired comment in the editor, save and close the editor.

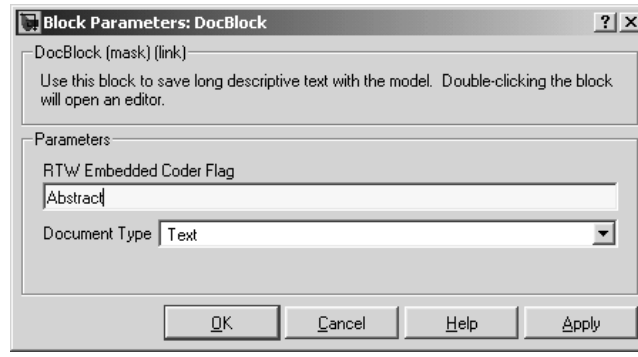
3. Right click the DocBlock and select **Mask Parameters**. The **Block Parameters** dialog box appears.

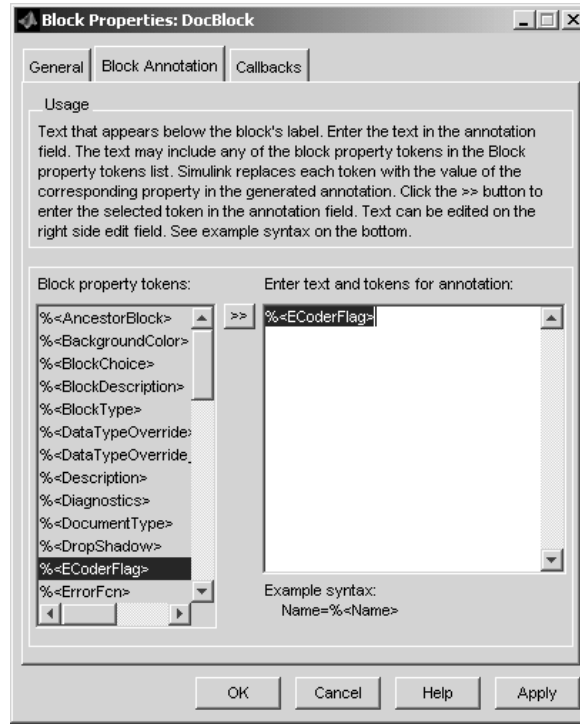
4. Type the desired Documentation child template symbol into the **RTW Embedded Coder Flag** field, as shown below, and then click **OK**. Note that symbol names are case sensitive. This is the symbol with which the comment will be placed in the generated file. To see a list of the supplied Documentation child template symbols, see Table A-3, Parent-Child Relationships of Template Symbols, on page A-19.

5. Right click the DocBlock and select **Block Properties**.

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 comment appears in the generated file in association with the symbol name.


Block Parameters: DocBlock
DocBlock (mask) (link)
Use this block to save long descriptive text with the model. Double-clicking the block will open an editor.
Parameters
RTW Embedded Coder Flag
Abstract
Document Type
Text
OK
Cancel
Help
Apply


Block Properties: DocBlock
General
Block Annotation
Callbacks
Usage
Text that appears below the block's label. Enter the text in the annotation field. The text may include any of the block property tokens in the Block property tokens list. Simulink replaces each token with the value of the corresponding property in the generated annotation. Click the >> button to enter the selected token in the annotation field. Text can be edited on the right side edit field. See example syntax on the bottom.
Block property tokens:
%<AncestorBlock>
%<BackgroundColor>
%<BlockChoice>
%<BlockDescription>
%<BlockType>
%<DataTypeOverride>
%<DataTypeOverride
%<Description>
%<Diagnostics>
%<DocumentType>
%<DropShadow>
%<ECoderFlag>
%<ErrorFcn>
>>
Enter text and tokens for annotation:
%<ECoderFlag>
Example syntax:
Name=%<Name>
OK
Cancel
Help
Apply

5

Managing File Placement of Data Definitions and Declarations

Introduction

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:

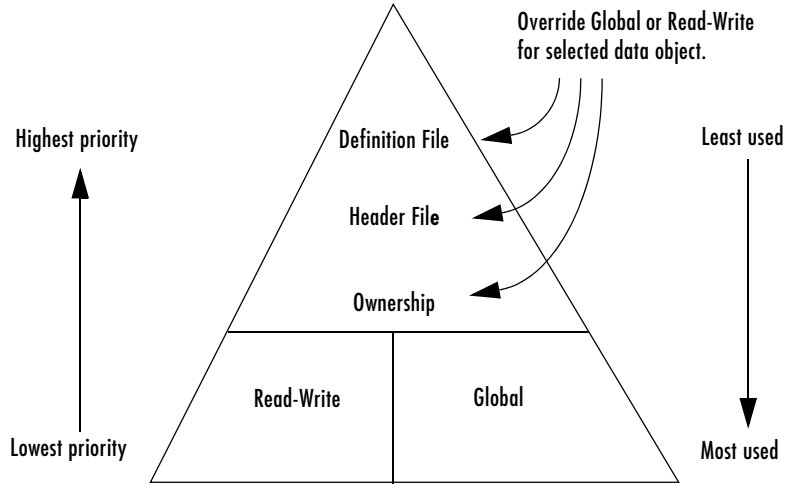
- The **Data definition** field on the **Data Placement** pane of the **Configuration Parameters** dialog box.
- The **Data declaration** on the **Data Placement** pane of the **Configuration Parameters** dialog box.
- The **Owner** field on the **Model Explorer** dialog box, and the **Module naming** 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 on the **Model Explorer** dialog box.
- The **Header file** field on the **Model Explorer** dialog box.
- The **Memory section** field on the **Model Explorer** dialog box.

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
- Global 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 "Adding Simulink Data Objects with Data Object Wizard" on page 3-13 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.

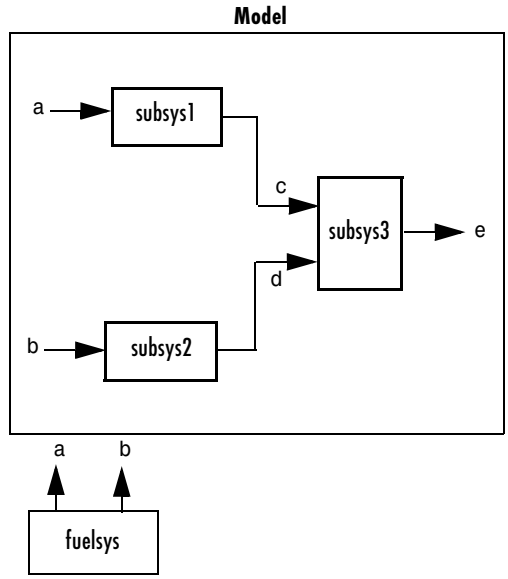
Note The priorities are in effect only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see "Designing Custom Storage Classes" in the Real-Time Workshop Embedded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

It will prove helpful to explain the lowest and most commonly used priorities first.

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 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.



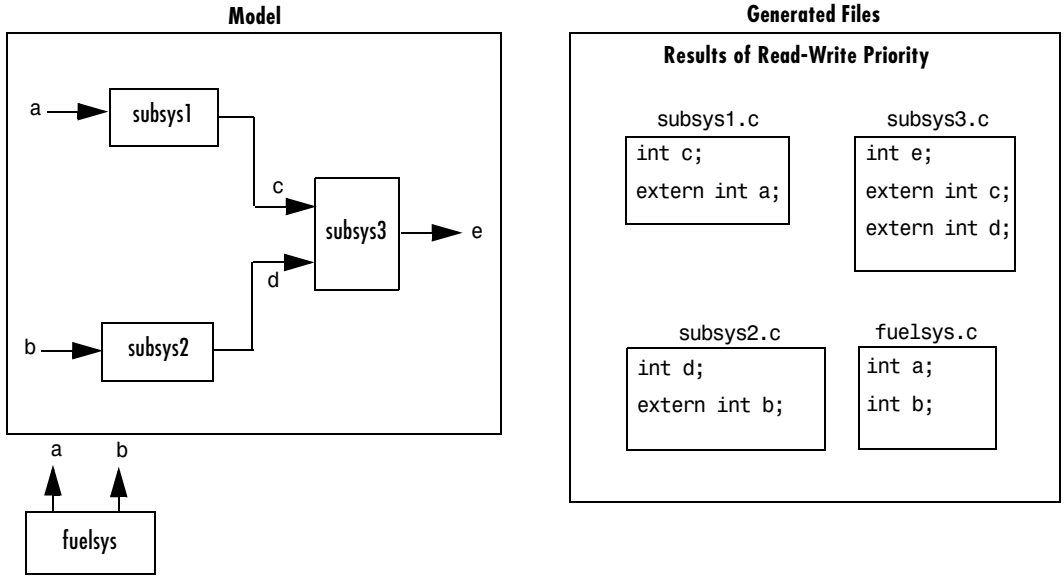
The Generated Files

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 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** dialog box 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** dialog box are blank.

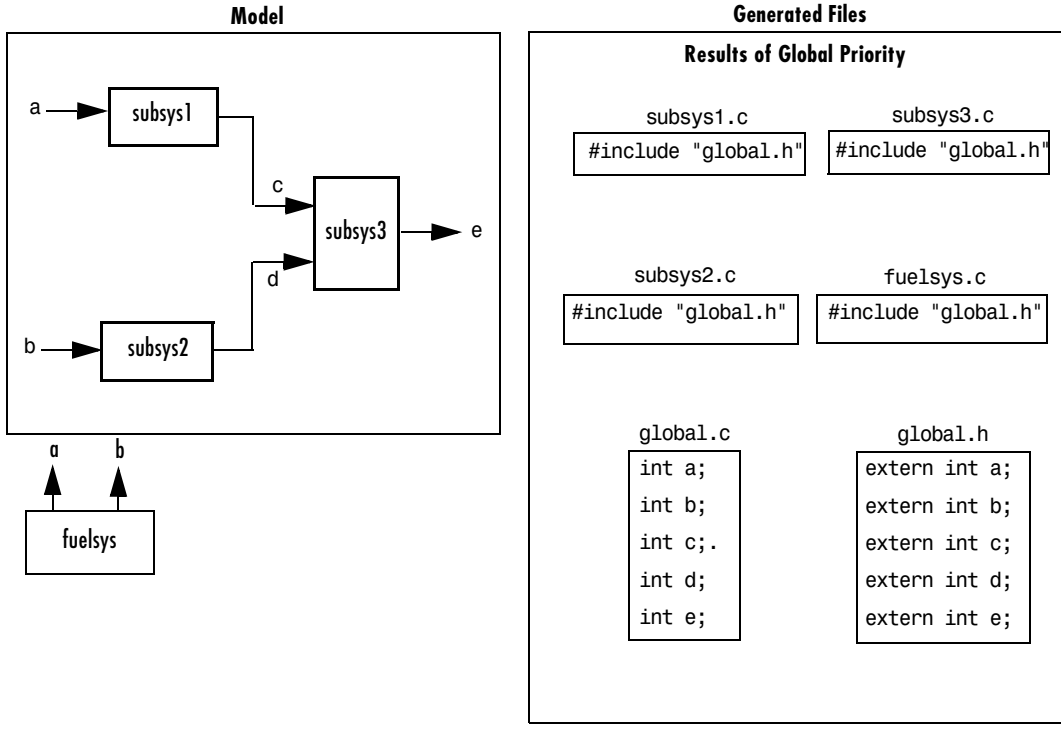
Global Priority

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** dialog box. 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 Table A-7, Effects of Ownership Settings, on page A-37.

The Memory Section Setting

Regarding **Memory section**, Table A-5, Parameter and Signal Property Values, on page A-24, 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.

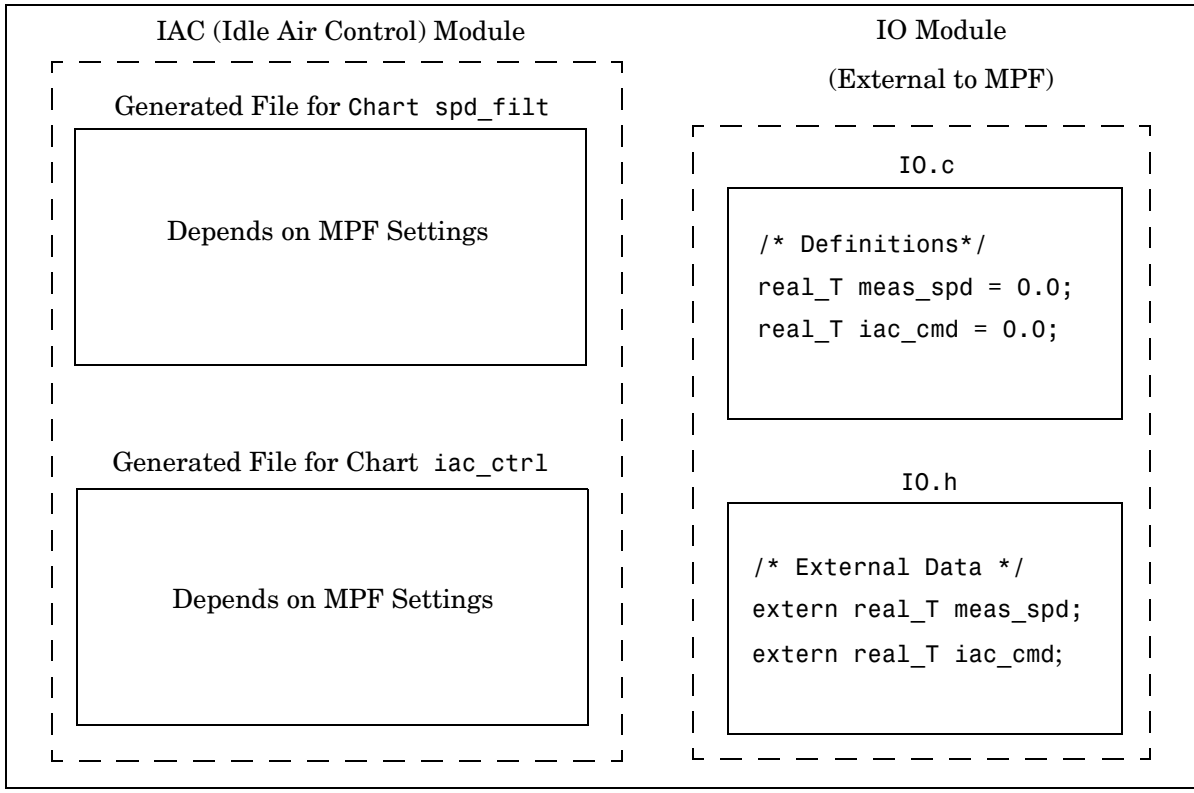
Note You can add more memory sections. See "Designing Custom Storage Classes" in the Real-Time Workshop Embedded Coder documentation.

Example Settings

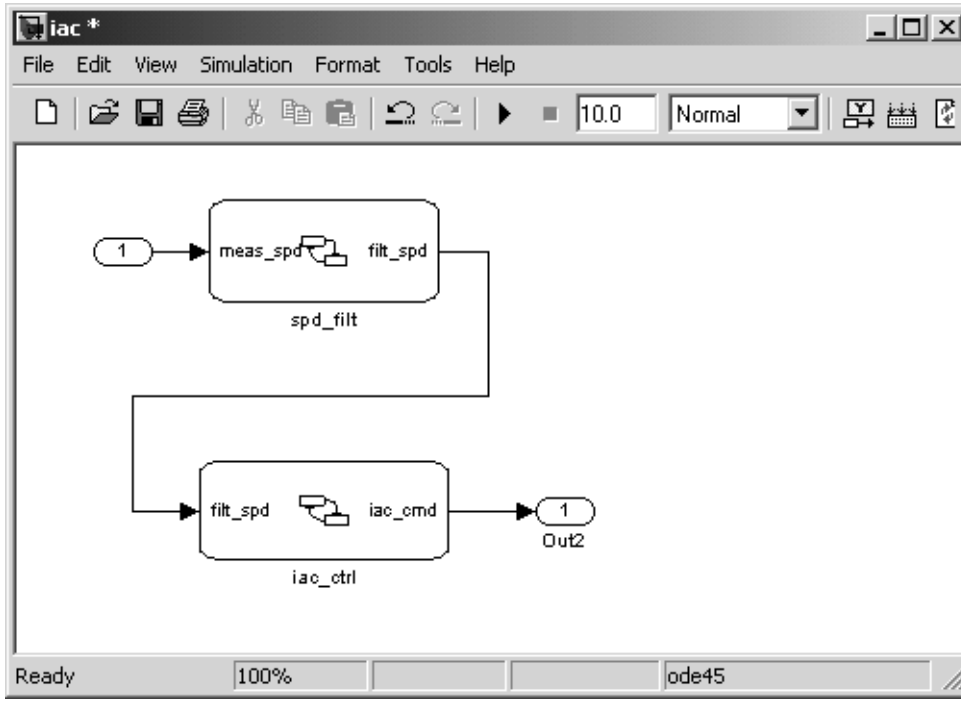
Table A-8, Example Settings and Resulting Generated Files, on page A-38, provides example settings for one data object of a model. Eight examples are listed so that you can see the generated files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. 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). 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 `.c` source file, called `IO.c`, and one `.h` 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.



Engine Idle Speed Control System



Proceed to the discussion of the desired example settings:

- “Read-Write Example” on page 5-13
- “Ownership Example” on page 5-14
- “Header File Example” on page 5-16
- “Definition File Example” on page 5-18

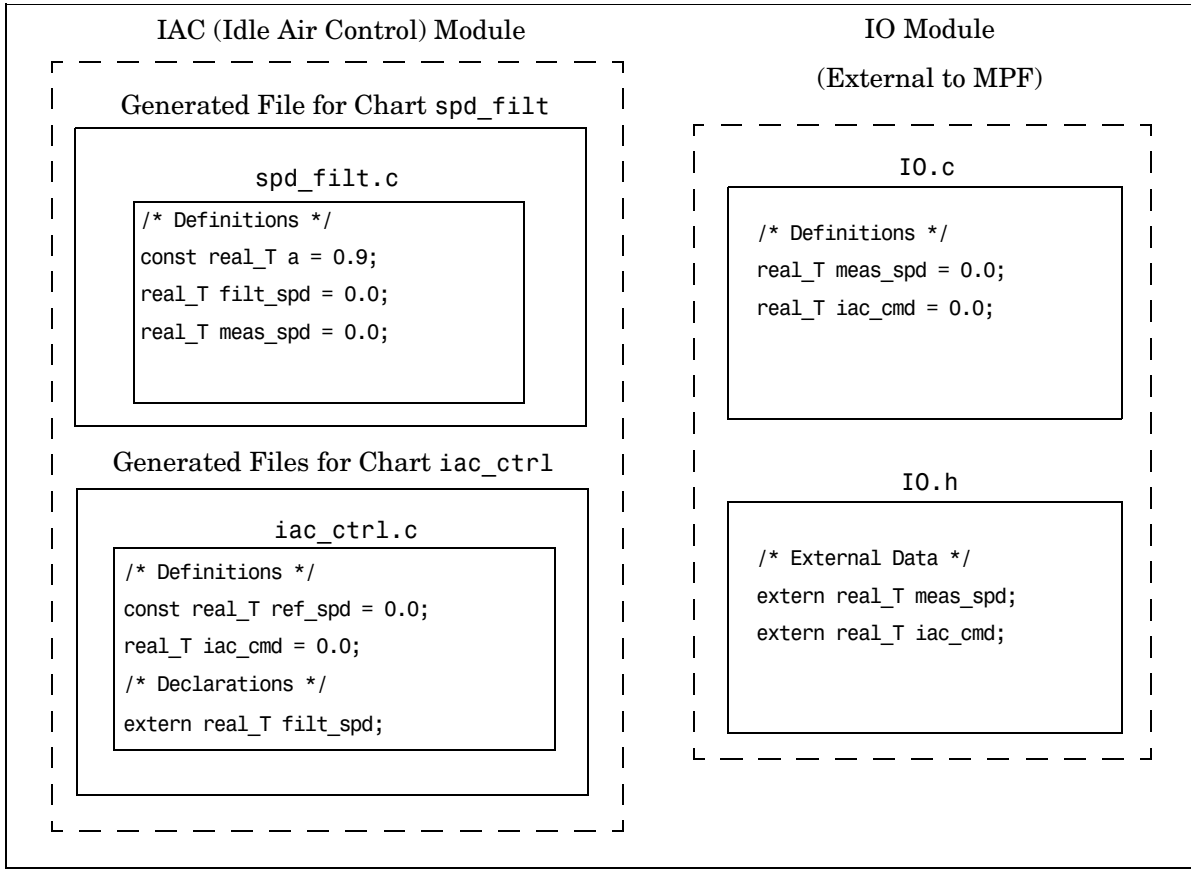
Read-Write Example

These settings and the generated files that result are shown as Example Settings 1 in Table A-8, Example Settings and Resulting Generated Files, on page A-38. 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** dialog box. 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.

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`.



Engine Idle Speed Control System (Read-Write Example)

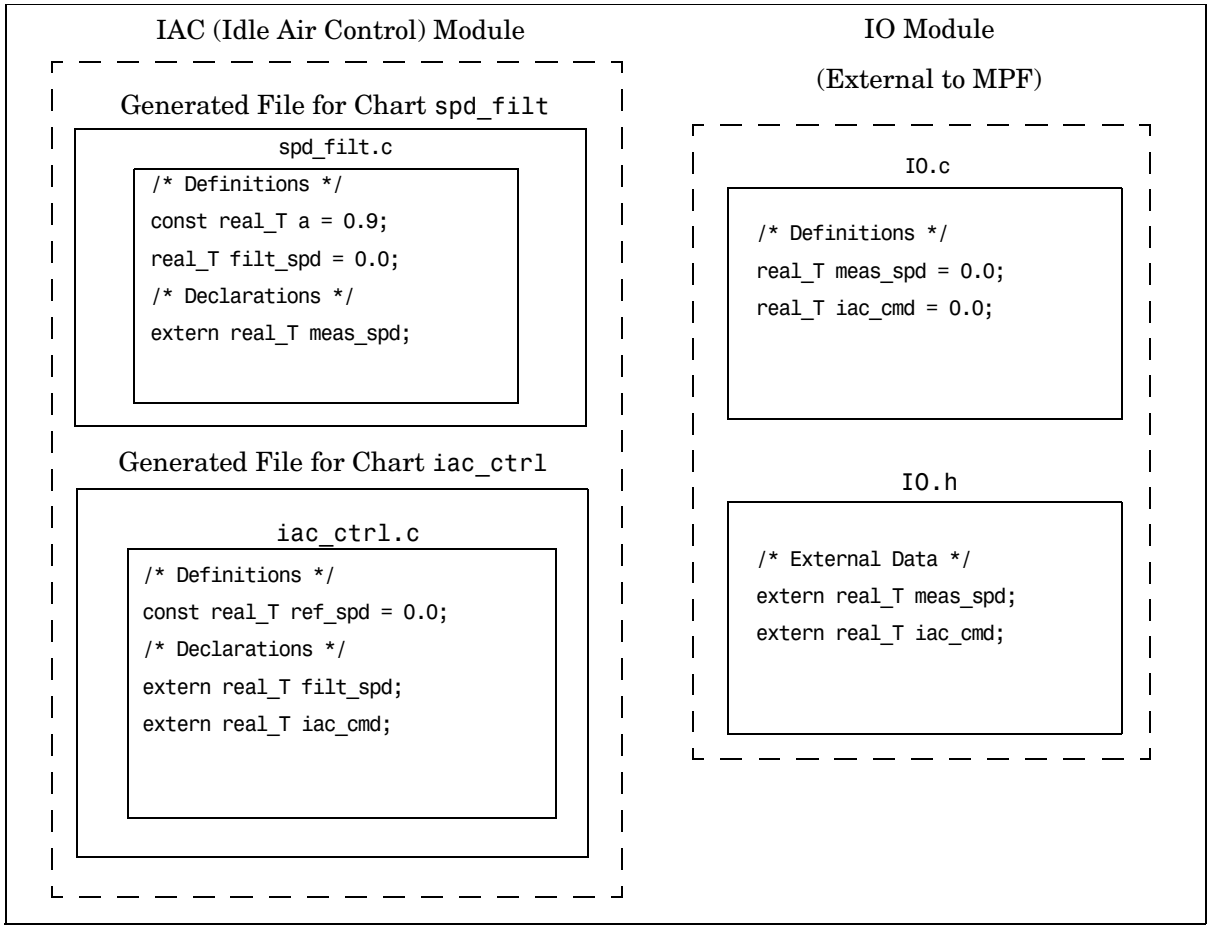
Ownership Example

See tables Table A-7, Effects of Ownership Settings, on page A-37, and Table A-8, Example Settings and Resulting Generated Files, on page A-38. In the "Read-Write Example" on page 5-13, 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 Table A-8, Example Settings and Resulting Generated Files, on page A-38. 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** dialog box (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

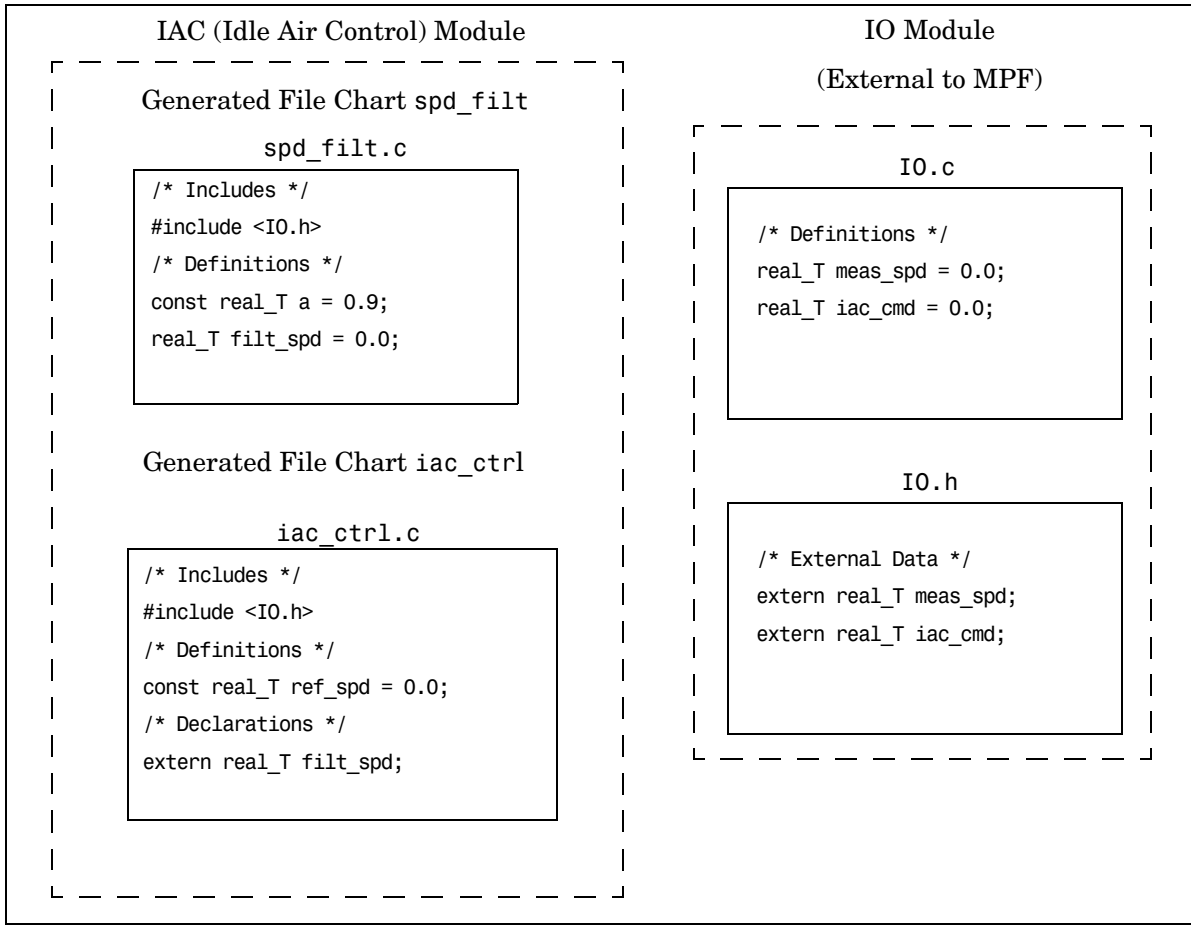
These settings and the generated files that result are shown as Example Settings 3 in Table A-8, Example Settings and Resulting Generated Files, on page A-38. Since this example has no **Definition file** specified, it allows us to describe the effects of the **Header file** setting. (If there *were* 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** dialog box 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** dialog box, 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. 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.



Engine Idle Speed Control System (Header File Example)

Definition File Example

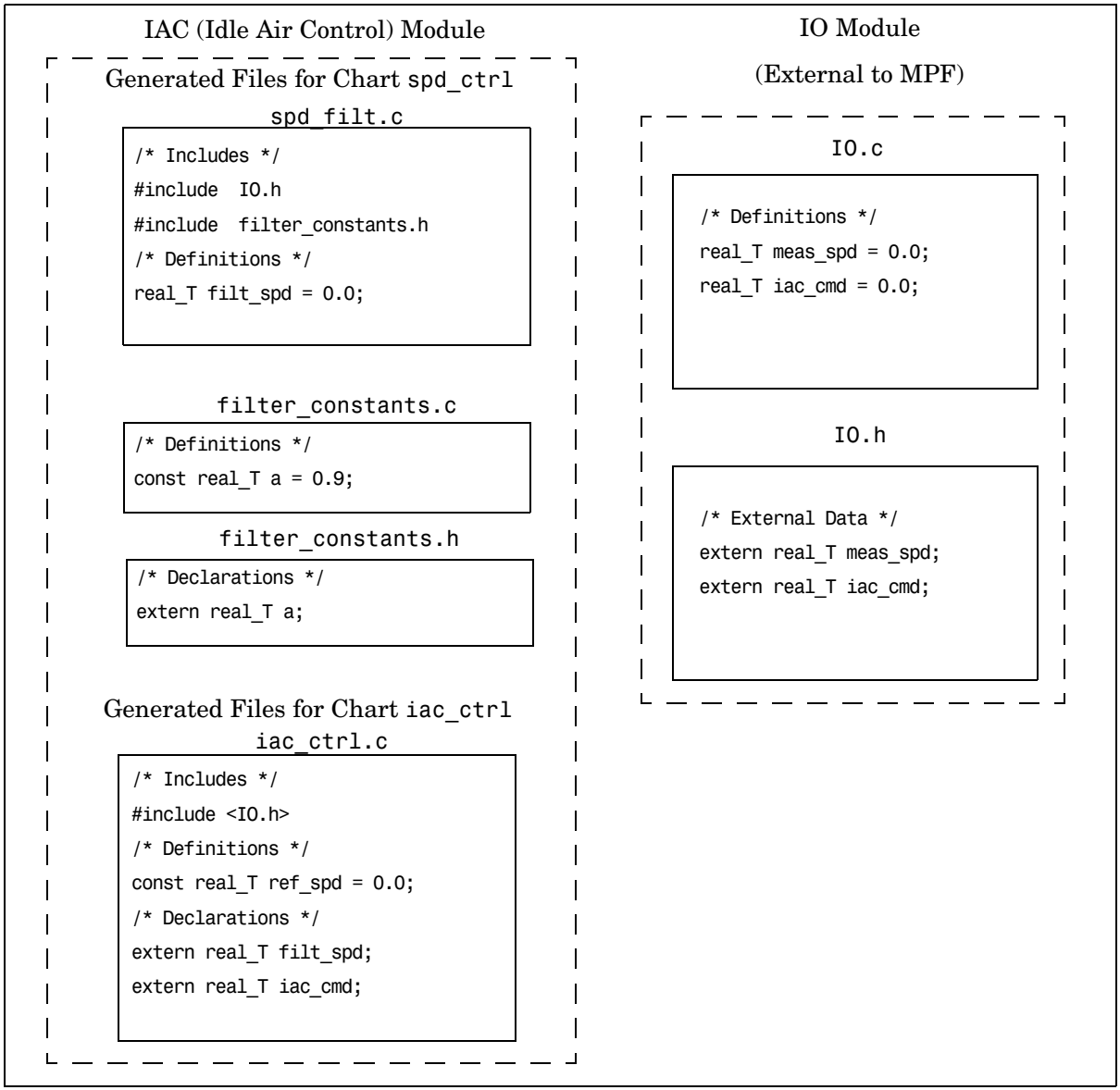
These settings and the generated files that result are shown as Example Settings 4 in Table A-8, Example Settings and Resulting Generated Files, on page A-38. Notice that a definition filename is specified. The settings in the table only apply to the data object called "a". You have decided that you 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.)

Note For this example, assume the settings for all data objects are the same as those indicated in "Header File Example" on page 5-16, except for the data object a. The description below identifies only the differences that result from this.

You open the **Model Explorer**, and select data object a. In the **Definition file** field you specify any desired filename. Choose control_constants.c. Generate code.

The results are shown in the next figure. The code generator generates the same files as in the "Header File Example" on page 5-16, and adds two new files, filter_constants.c and filter_constants.h. 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 in filter_constants.h with an extern statement. MPF accounted for this .h file by placing the #include filter_constants.h in the spd_filt.c source file spd_filt.c.



Engine Idle Speed Control System (Definition File Example)

A

Referenced Tables

MPF-Related Panes on Configuration 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-13)	Lists and describes all MPF template symbols and template symbol rules.
Parameter and Signal Properties (p. A-23)	Lists and describes all `mpt` parameter and signal properties and property values, and illustrates how changing these affect the generated code.
Interdependent Settings (p. A-37)	Shows the effects that example changes to the interdependent MPF settings have on the generated code.

MPF-Related Panes on Configuration 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.

Table A-1: 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** dialog box, 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 "Setting Property Values" on page 3-15. 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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		For example, in "Setting Property Values" on page 3-15, 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 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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		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 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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	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.
	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.)

Table A-1: 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.
		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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	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. 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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	#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. 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** property of a data object in the **Model Explorer** dialog box to constitute what is termed "ownership." For details, see "Ownership" on page 5-8 and Table A-7, Effects of Ownership Settings, on page A-37. 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.
	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 conventions for naming identifiers.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	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 in "Setting Property Values" on page 3-15. 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 **Persistence level** is equal to or less than the **Signal display level**, the signal will appear in the generated code as global data with all of the properties (custom attributes) specified in "Setting Property Values" on page 3-15. For example, this would occur if **Persistence level** is 2 and **Signal display level** is 5.
		Otherwise, the code generator automatically will determine how the particular signal data object will appear 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 Controlling and Optimizing the Generated Code in the Real-Time Workshop Embedded Coder documentation for details on optimization.)

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	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 in "Setting Property Values" on page 3-15. 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 **Persistence level** is equal to or less than the **Parameter tune level,** the parameter will appear in the generated code with all of the properties (custom attributes) specified in "Setting Property Values" on page 3-15, 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 by a `#define` preprocessor directive, and thus is not tunable.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
		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 will declare 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.

Table A-1: MPF Elements on Configuration Parameters Panes (Continued)

Pane	Element	Description
	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`. If you accept `Model`, the statement the code generator places in the file for each signal is `type identifier = zero;`, where `identifier` corresponds to the signal in the model, and `zero` is `0.0` or `0`, depending on the type. The `model` selection always initializes the value to zero. If you select `Data object`, the statement the code generator places in the file is `type identifier = value;`, where `value` is the **Initial value** property value you entered for each signal data object in the **Model Explorer**. (See "Setting Property Values" on page 3-15.)

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.

Table A-2: Template Symbols

Symbol Name* (Each must be enclosed within %< >)	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 functions. Must be at the bottom of the template.

Table A-2: Template Symbols (Continued)

Symbol Name* (Each must be enclosed within %< >)	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`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.
`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.

Table A-2: Template Symbols (Continued)

Symbol Name* (Each must be enclosed within %< >)	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`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.
`ExportAccessMethods`	Defines	Global	External `#define` definitions that correspond to the alternate names in the data objects.
`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	***

Table A-2: Template Symbols (Continued)

Symbol Name* (Each must be enclosed within %< >)	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`FilescopeVariableScalar`	Definitions	File	***
`Functions`	Base	File	Generated function code.
`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.

Table A-2: Template Symbols (Continued)

Symbol Name* (Each must be enclosed within %< >)	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`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 macros local to the file.
`ModelName`	Documentation	N/A	Name of the model.
`ModelVersion`	Documentation	N/A	Version number of the Simulink model.
`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	
`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.

Table A-2: Template Symbols (Continued)

Symbol Name* (Each must be enclosed within %< >)	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`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, `%<CFunctions>`.

** See "Entering Chart Notes" in the Stateflow documentation, "Annotations" in the Using Simulink documentation, or "DocBlock" in the Simulink documentation. "`<S:symbol name>`" must precede the note or annotation on the model (without the quotation marks), where `symbol name` is the name of the MPF symbol. The code generator searches for this and then will populate the generated file with the note or annotation. For example, if "`<S:History>This is the history of this model.`" is a note or annotation on the model, and "`HISTORY: %<History>`" is on the template, then "`HISTORY: This is the history of this model.`" will appear on the generated file including any spaces.

*** The description can be deduced from the symbol name. For example, `ExternalCalibrationLookup2D` is a symbol that identifies a two-dimensional lookup table. It contains calibratable data of external scope (global scope from the perspective of the present file).

Table A-3: Parent-Child Relationships of Template Symbols

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`
	`ExportAccessMethods`
Definitions	`FilescopeCalibrationLookup1D`
	`FilescopeCalibrationLookup2D`
	`FilescopeCalibrationScalar`
	`FilescopeVariableScalar`
	`GlobalCalibrationLookup1D`
	`GlobalCalibrationLookup2D`
	`GlobalCalibrationScalar`
	`GlobalVariableScalar`

Table A-3: Parent-Child Relationships of Template Symbols (Continued)

Symbol Group	Symbol Names in This Group

Documentation	`Abstract`
	`Banner`
	`Created`
	`Creator`
	`Date`
	`Description`
	`FileName`
	`History`
	`LastModificationDate`
	`LastModifiedBy`
	`ModelName`
	`ModelVersion`
	`ModifiedBy`
	`ModifiedComment`
	`ModifiedDate`
	`ModifiedHistory`
	`Notes`
	`ToolVersion`

Functions	`CFunctionCode`
Types	This parent has no children.

Table A-4: Rules for Modifying or Creating a Template

Rules for All MPF Templates:
1 Place a symbol on a template within the `%< >` delimiter. For example, the symbol named `IncludeFile` should look like this on a template: `%<IncludeFile>`. *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 text without the `%< >` delimiter for that text to appear in the generated file. For example, `"HISTORY: "` on the template will result in `"HISTORY: "` verbatim (including the space after the colon) at the corresponding location in the generated file.
4 Use the `.cgt` extension for every template filename. ("`cgt`" stands for *c*ode *g*eneration *t*emplate.)
5 Note that `%% $Revision: x.y $` 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 comments on the template between `/* */` as in standard ANSI C. This will result in `/*comment*/` on the generated file.
7 Each MPF template must have all of the Base group symbols. They are listed in Table A-3, Parent-Child Relationships of Template Symbols, on page A-19. 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 and before the `Functions` symbol. Children can be in any order. They can be anywhere in the template after their parent, even after another parent. For example, if the `Defines` parent is on line 23 of the template and the `Declarations` parent is on line 32, the Defines child `LocalDefines` can be on any line on or after line 24, even on line 33.

Table A-4: Rules for Modifying or Creating a Template (Continued)

Rules for All MPF Templates:
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 assoicated 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** dialog box. Also, a table describes the effects that example changes to property values have on the generated code.

Table A-5: Parameter and Signal Property Values

Parameter Class, Signal Class, or Both Classes	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 3-10.) 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** dialog box 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.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
Parameter Only	`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`		Use 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 "Specifying Data Types," "Data Type Functions," and "Data Type Classes" in the Simulink documentation.
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.
Both	`Complexity`	`*auto`	Complexity specifies whether the signal or parameter is a real or complex number. Select `auto` for the code generator to decide.
		`real`	
		`complex`	

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
Signal Only	`Sample time`	`*-1`	Model or block execution rate.
Signal Only	`Sample mode`	`*auto`	Determines how the signal propagates through the model. Select `auto` for the code generator to decide.
Signal Only		`Sample based`	The signal propagates through the model one sample at a time.
Signal Only		`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. (Enter information using a dialog box.)
		Any number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
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`		

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
	`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.	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.
Both		`MemConst`	Places the `const` type qualifier in the declaration.
Both		`MemVolatile`	Places the `volatile` type qualifier in the declaration.
Both		`MemConstVolatile`	Places the `const volatile` type qualifier in the declaration.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	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.
Both	`Owner`	`*null`	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 5-8 and Table A-7, Effects of Ownership Settings, on page A-37.
Both	`Definition file`	`*Blank`	Name of the file that defines the data object.
		Any valid text string. (We recommend that this string be concise, that is, contain alphabet or numeric characters, underscore, no spaces.)	
Both	`Alternate name`		

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	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 Table A-1, 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 Only	`Const (Custom)`		Places the `const` type qualifier in the declaration.
Parameter Only	`Header file`		See above.
Parameter Only	`Owner`		See above.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
Parameter Only	`Definition file`		See above.
Parameter Only	`Alternate name`		See above.
Parameter Only	`Persistence level`		See above.
Both	`Volatile (Custom)`		Places the `volatile` type qualifier in the declaration.
Both	`Header file`		See above.
Both	`Owner`		See above.
Both	`Definition file`		See above.
Both	`Alternate name`		See above.
Both	`Persistence level`		See above.
Parameter Only	`ConstVolatile (Custom)`		Places the `const volatile` type qualifier in declaration.
Parameter Only	`Header file`		See above.
Parameter Only	`Owner`		See above.
Parameter Only	`Definition file`		See above.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
Parameter Only	`Alternate name`		See above.
Parameter Only	`Persistence level`		See above.
Parameter Only	`Define (Custom)`		Represents parameters with a `#define` macro.
Parameter Only	`Header file`		See above.
Parameter Only	`Alternate name`		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.
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.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
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 Only	`GetSet (Custom)`		Reads (gets) and writes (sets) data using functions.
Signal Only	`Header file`		See above.
Signal Only	`Get function`		Specify the Get function.
Signal Only	`Set function`		Specify the Set function.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
Both	`Alias`	*`null`	As explained in detail in "Applying Naming Rules to Identifiers Globally" on page 3-19, for an mpt data object (identifier), selecting the **Alias overrides naming rule** check box causes the name you specify in the **Alias** field to override the naming rule selection you make on the **Configuration Parameters** dialog box. But for a Simulink data object, selecting other than `None` in a naming rule field overrides **Alias** on the **Model Explorer** regardless of whether or not you specify an **Alias** name. (The **Alias overrides naming rule** check box is not available for a Simulink data object.)
		Any valid ANSI C variable name	
	`Alias overrides naming rule`		See **Alias** above.
Signal Only	`Initial value`		Numeric value used as an initial value in the generated code.

Table A-5: Parameter and Signal Property Values (Continued)

Parameter Class, Signal Class, or Both Classes	Property	Available Property Values (* Indicates Default)	Description
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	

Table A-6: Some Examples of the Effect of Property Value Changes on Generated Code

What I noticed when inspecting the .c file	Change I made to property value settings	What I noticed after regenerating and reinspecting the .c 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** dialog box, 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 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.

Table A-6: Some Examples of the Effect of Property Value Changes on Generated Code (Continued)

What I noticed when inspecting the .c file	Change I made to property value settings	What I noticed after regenerating and reinspecting the .c file
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** dialog box, I notice that the property value for the **Header file** property is blank. I changed this to `"filename.h"`. (I chose the ANSI 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 compiler, placing it either in the system path or local directory.	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.
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` property is blank. I changed this to `data_in_alias`, which I know is a valid ANSI C variable name.	The identifier `data_in_alias` now appears in the `.c` file everywhere `data_in` appeared.

Interdependent Settings

The following tables show the effects that example changes to the interdependent MPF settings have on the generated code. See “Example Settings” on page 5-10.

Table A-7: 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` 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 `.c` source files, as needed.
2	`Not specified`**	A name is specified.	There is no definition for the selected data object. But there is an `extern` declaration for the selected data object. This `extern` declaration is placed in one or more `.c` source files, as needed.
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.

Table A-7: Effects of Ownership Settings (Continued)

Row Number	Module Naming Setting	Owner Setting	Effect*
5	Either `Same as model` or `User specified` is selected, and this name is different than that specified as the **Owner** property.	A name is specified but it is different from that specified in the **Module naming** (**Module name**) field.	Same as for Row 2.

* See also "Ownership" on page 5-8.

** Default

Table A-8: Example Settings and Resulting Generated Files

	Data Def.	Data Dec.	Owner-ship*	Defin. File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	`Data defined in source file`	`Data dec.in source file`	Blank	Blank	Blank	`.c` source file
Example Settings 2 (Owner-ship Example)	`Data defined in source file`	`Data dec.in source file`	Name of module spec-ified	Blank	Blank	`.c` source file

Table A-8: Example Settings and Resulting Generated Files (Continued)

	Data Def.	Data Dec.	Owner-ship*	Defin. File**	Header File	Generated Files
Example Settings 3 (Header File Example)	`Data defined in source file`	`Data dec.in source file`	Blank	Blank	Desired include file-name speci-fied.	`.c` source file
Example Settings 4 (Def. File Example)	`Data defined in source file`	`Data dec.in source file`	Blank	Desired definition file-name speci-fied.	Desired include file-name speci-fied.	`.c` source file; `.c` definition file*; `.h` definition file*
Example Settings 5	`Data defined in single sep. source file`	`Data dec.in source file`	Blank	Blank	Blank	`.c` source file; global `.c`
Example Settings 6	`Data defined in single sep. source file`	`Data dec. in single sep. header file`	Blank	Blank	Blank	`.c` source file; `global .c`; `global.h`
Example Settings 7	`Data defined in single sep. source file`	`Data dec. in single sep. header file`	Name of module speci-fied	Blank	Blank	`.c source` file; `global .c`; `global.h`;

Table A-8: Example Settings and Resulting Generated Files (Continued)

	Data Def.	Data Dec.	Owner-ship*	Defin. File**	Header File	Generated Files
Example Settings 8	`Data defined in single sep. source file`	`Data dec. in single sep. header file`	Blank	Blank	Desired include file-name speci-fied.	`.c` source file; `global.c`; `global.h`

* "Blank" in ownership setting means `Not specified` is selected in the **Module naming** field on the **Data Placement** pane, and the **Owner** property on the **Model Explorer** is blank. "Name of module specified" can be a variety of ownership settings as defined in Table A-7, Effects of Ownership Settings, on page A-37.

** The code generator generates a definition `.c` and declaration `.h` file for every data object for which you specified a definition filename (unless you selected `#DEFINE` for the **Memory section** property). For example, if you specify the same definition filename for all data objects, only one definition `.c` and only one declaration `.h` is generated. But if you specify a definition filename for each data object, the code generator generates one definition `.c` file for *each* data object plus one declaration `.h` file for *each* data object.

Index