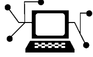
Embedded Coder™ 6
User's Guide

MATLAB®
&SIMULINK®

MathWorks®

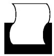
How to Contact MathWorks



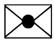
www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

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.
3 Apple Hill Drive
Natick, MA 01760-2098

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

Embedded Coder™ User's Guide

© COPYRIGHT 2011 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

April 2011	Online only	New for Version 6.0 (Release 2011a)

Contents

1 Introduction to the Embedded Coder Product

2 Bug Reports

Developing Models for Code Generation

3 Setting Up Your Modeling Environment

4 Architecture Considerations

Generating Code for Variant Systems **4-2**
Overview **4-2**
Why Generate Code for Variant Systems? **4-3**
How to Generate Preprocessor Conditionals for Variant Systems **4-3**
Reviewing Code Variants in the Code Generation Report **4-6**
Example of Model Variants in the Generated Code **4-7**
Example of Variant Subsystems in the Generated Code . . **4-9**
Restrictions on Code Generation of a Variant Subsystem . . **4-13**
Special Considerations for Generating Preprocessor Conditionals **4-14**
Limitations on Generating Code for Variants **4-15**
Exceptions to Conditionally Compiled Components in the Generated Code **4-15**

Demos for Generating Code for Variants 4-16

Creating and Using Host-Based Shared Libraries **4-17**
Overview 4-17
Generating a Shared Library Version of Your Model Code 4-18
Creating Application Code to Load and Use Your Shared Library File 4-19
Host-Based Shared Library Limitations 4-23

5 Scheduling Considerations

Using Discrete and Continuous Time **5-2**
Generating Code for Discrete and Continuous Time Blocks 5-2
Generating Code that Supports Continuous Solvers 5-2
Generating Code that Honors a Stop Time 5-3

Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets **5-4**
Overview 5-4
Using rtmStepTask 5-5
Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target 5-5
Suppressing Redundant Scheduling Calls 5-6

6 Developing Model Patterns that Generate Specific C Constructs

About Modeling Patterns **6-2**

Standard Methods to Prepare a Model for Code Generation **6-3**

Configuring a Signal 6-3
Configuring Input and Output Ports 6-4
Initializing States 6-4
Setting Up Configuration Parameters for Code Generation 6-4
Setting Up an Example Model With a Stateflow Chart 6-5
Setting Up an Example Model With a MATLAB Function Block 6-7

Types, Operators, and Expressions 6-8
Data Declaration 6-8
Data Type Conversion 6-11
Type Qualifiers 6-15
Relational and Logical Operators 6-17
Bitwise Operations 6-21

Control Flow 6-25
If-Else 6-25
Switch 6-32
For loop 6-38
While loop 6-46
Do While loop 6-58

Functions 6-65
Function Call 6-65
Function Prototyping 6-67
External C Functions 6-70

Preprocessor Directives 6-77
Macro Definitions (#define) 6-77
Conditional Inclusions (#if / #endif) 6-79

Structures 6-81
Typedef 6-81
Structures for Parameters 6-83
Structures for Signals 6-85
Nested Structures 6-89
Bitfields 6-92

Arrays 6-95
Arrays for Parameters 6-95

Arrays for Signals 6-97

Pointers **6-99**
Pointers for Signals 6-99
Pointers for Signals and Parameters Using Simulink Data Objects 6-100

Defining Data Representation and Storage for Code Generation

7 Using mpt Data Objects

8 Creating and Using Custom Storage Classes

Introduction to Custom Storage Classes 8-2
Custom Storage Class Memory Sections 8-3
Registering Custom Storage Classes 8-3
Custom Storage Class Demos 8-4

Resources for Defining Custom Storage Classes 8-5

Simulink Package Custom Storage Classes 8-6

Creating Packages that Support CSC Definitions 8-8

Designing Custom Storage Classes and Memory Sections 8-12
Using the Custom Storage Class Designer 8-12
Editing Custom Storage Class Properties 8-19
Using Custom Storage Class References 8-26
Creating and Editing Memory Section Definitions 8-31
Using Memory Section References 8-34

Applying CSCs to Parameters and Signals **8-37**
About Applying Custom Storage Classes **8-37**
Applying a Custom Storage Class to a Parameter **8-38**
Applying a Custom Storage Class to a Signal **8-40**
Applying a CSC Using a Base Workspace Signal Object .. **8-41**
Applying a CSC Using an Embedded Signal Object **8-43**
Specifying a Custom Storage Class Using the GUI **8-50**
Specifying a Custom Storage Class Using the API **8-53**

Generating Code with Custom Storage Classes **8-58**
Code Generation Prerequisites **8-58**
Code Generation Example **8-58**

Defining Advanced Custom Storage Class Types **8-62**
Introduction **8-62**
Create Your Own Parameter and Signal Classes **8-62**
Create a Custom Attributes Class for Your CSC (Optional) **8-62**
Write TLC Code for Your CSC **8-63**
Register Custom Storage Class Definitions **8-63**

GetSet Custom Storage Class for Data Store Memory .. **8-66**
Overview **8-66**
GetSet CSC Properties **8-66**
Using the GetSet CSC **8-67**
GetSet CSC Restrictions **8-67**
GetSet Custom Storage Class Example **8-68**

Custom Storage Class Implementation **8-70**

Custom Storage Class Limitations **8-72**

9 Memory Sections

Introduction to Memory Sections **9-2**
Overview **9-2**
Memory Sections Demo **9-2**

Additional Information 9-2

Requirements for Defining Memory Sections **9-4**

Defining Memory Sections **9-7**
Editing Memory Section Properties 9-7
Specifying the Memory Section Name 9-9
Specifying a Qualifier for Custom Storage Class Data Definitions 9-9
Specifying Comment and Pragma Text 9-9
Surrounding Individual Definitions with Pragmas 9-9
Including Identifier Names in Pragmas 9-10

Configuring Memory Sections **9-11**

Declaring Constant Data as Volatile **9-12**

Applying Memory Sections **9-15**
Assigning Memory Sections to Custom Storage Classes 9-15
Applying Memory Sections to Model-Level Functions and Internal Data 9-17
Applying Memory Sections to Atomic Subsystems 9-19

Examples of Generated Code with Memory Sections **9-23**
Sample ERT-Based Model with Subsystem 9-23

Model-Level Data Structures **9-25**
Model-Level Functions 9-25
Subsystem Function 9-26

Memory Section Limitation **9-28**

10 Optimizing Buses for Code Generation

Introduction **10-2**

Setting Bus Diagnostics **10-3**

Optimizing Virtual and Nonvirtual Buses **10-4**
Use Virtual Buses Wherever Possible **10-4**
Avoid Nonlocal Nested Buses in Nonvirtual Buses **10-5**

Using Single-Rate and Multi-Rate Buses **10-7**
Introduction **10-7**
Techniques for Combining Multiple Rates **10-7**
Larger Buses and Multiple Rates **10-9**
Specifying Sample Time Rates **10-10**

Setting Bus Signal Initial Values **10-11**
Introduction **10-11**
Initializing Bus Signals in Simulink **10-11**
Bus Initialization in Stateflow **10-12**
Creating a Bus of Constants **10-14**

Buses and Atomic Subsystems **10-16**
Extract Nonvirtual Bus Signals Inside of Atomic Subsystems **10-16**
Virtual Bus Signals Crossing Atomic Boundaries **10-17**
Atomic Subsystems and Buses of Constants **10-19**

11 Renaming and Replacing Data Types

Defining Application-Specific Data Types Based On Built-In Types **11-2**

Code Generation with User-Defined Data Types **11-4**
Overview **11-4**
Specifying Type Definition Location for User-Defined Data Types **11-5**
Using User-Defined Data Types for Code Generation **11-6**

12 Managing Data Definitions and Declarations With the Data Dictionary

Overview of the Data Dictionary **12-2**

Creating Simulink and mpt Data Objects **12-4**
Overview **12-4**
Creating Simulink Data Objects with Data Object Wizard **12-5**
Creating mpt Data Objects with Data Object Wizard **12-11**
Comparing Simulink and mpt Data Objects **12-12**
Creating Data Objects Based on an External Data Dictionary **12-16**

Creating a Data Dictionary for a Model **12-19**
Using Data Object Wizard **12-19**
Inspect the Data Dictionary **12-23**
Generate and Inspect Code **12-24**

Defining All Global Data Objects in a Separate File ... **12-26**

Defining a Specific Global Data Object in Its Own File **12-28**

Saving and Loading Data Objects **12-29**

Applying Naming Rules to Identifiers Globally **12-30**
Overview **12-30**
Changing Names of Identifiers **12-31**
Specifying Simulink Data Object Naming Rules **12-34**
Defining Rules That Change All Signal Names **12-35**
Defining Rules That Change All Parameter Names **12-35**
Defining Rules That Change All #defines **12-36**

Creating User Data Types **12-38**
Overview **12-38**
Registering User Data Types Using sl_customization.m .. **12-39**
Example User Data Type Customization Using sl_customization.m **12-41**

Selecting User Data Types for Signals and Parameters 12-43
Preparing User Data Types 12-43
Selecting the User Data Types 12-45

Registering mpt User Object Types 12-48
Introduction 12-48
Registering mpt User Object Types Using sl_customization.m 12-48
Example mpt User Object Type Customization Using sl_customization.m 12-50

Replacing Built-In Data Type Names in Generated Code 12-52
Replacing Built-In Data Type Names 12-52
Replacing `boolean` with an Integer Data Type 12-57
Data Type Replacement Limitations 12-59

Customizing Data Object Wizard User Packages 12-60
Introduction 12-60
Registering Data Object Wizard User Packages Using sl_customization.m 12-61
Example Data Object Wizard User Package Customization Using sl_customization.m 12-62

13 Managing Placement of Data Definitions and Declarations

Overview of Data Placement 13-2

Priority and Usage 13-3
Overview 13-3
Read-Write Priority 13-4
Global Priority 13-7
Definition File, Header File, and Ownership Priorities 13-8

Ownership Settings 13-10

Memory Section Settings **13-11**

Data Placement Rules **13-12**

Example Settings **13-13**
Introduction **13-13**
Read-Write Example **13-15**
Ownership Example **13-17**
Header File Example **13-18**
Definition File Example **13-20**

Data Placement Rules and Effects **13-22**
Effects of Ownership Settings **13-22**
Example Settings and Resulting Generated Files **13-23**
Data Placement Rules **13-25**

14 Specifying the Persistence Level for Signals and Parameters

Preparing Models for Code Generation

15 Mapping Application Objectives to Model Configuration Parameters

Considerations When Mapping Application Objectives **15-2**

Defining High-Level Code Generation Objectives **15-3**

Determining Whether the Model is Configured for Specified Objectives **15-4**
Specifying Code Generation Objectives Using the GUI . . . **15-4**

Specifying Code Generation Objectives at the Command Line 15-6
Reviewing Objectives in Referenced Models 15-7
Reviewing the Model Without Generating Code 15-7
Reviewing the Model During Code Generation 15-9

Creating Custom Objectives 15-11
Specifying Parameters in Custom Objectives 15-11
Specifying Checks in Custom Objectives 15-12
Determining Checks and Parameters in Existing Objectives 15-12
How to Create Custom Objectives 15-14

16 Selecting and Configuring an Embedded Real-Time Target

Introduction 16-2

Selecting an ERT Target 16-4

Customizing an ERT Target 16-6

17 Specifying Code Appearance and Documentation

Customizing Comments in Generated Code 17-2
Adding Custom Comments to Generated Code 17-2
Adding Global Comments 17-5

Configuring the Appearance of Generated Identifiers 17-12
Customizing Generated Identifiers 17-12
Configuring Symbols 17-13

Controlling Code Style **17-22**

Configuring Templates for Customizing Code Organization and Format **17-23**
Overview **17-24**
Custom File Processing Components **17-25**
Custom File Processing User Interface Options **17-25**
Code Generation Template (CGT) Files **17-27**
Using Custom File Processing (CFP) Templates **17-31**
Custom File Processing (CFP) Template Structure **17-31**
Changing the Organization of a Generated File **17-33**
Generating Source and Header Files with a Custom File Processing (CFP) Template **17-35**
Comparison of a Template and Its Generated File **17-44**
Code Template API Summary **17-47**
Generating Custom File and Function Banners **17-50**
Template Symbols and Rules **17-59**

Configuring the Placement of Data in Generated Code **17-68**

Ensuring Delimiter Is Specified for All #Includes **17-69**

18 Defining Model Configuration Variations

Introduction **18-2**

Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer **18-3**

Generating Code and Building Executables

19 Generating Code Modules

Code Modules **19-2**
Introduction **19-2**
Generated Code Modules **19-2**
User-Written Code Modules **19-5**
Customizing Generated Code Modules **19-5**

20 Generating Reports for Code Reviews and Traceability Analysis

About HTML Code Generation Report Extensions **20-2**

Generating an HTML Code Generation Report **20-4**

Using the Code Interface Report to Analyze the Generated Code Interface **20-6**
Code Interface Report Overview **20-6**
Generating a Code Interface Report **20-7**
Navigating Code Interface Report Subsections **20-9**
Interpreting the Entry Point Functions Subsection **20-10**
Interpreting the Inports and Outports Subsections **20-13**
Interpreting the Interface Parameters Subsection **20-14**
Interpreting the Data Stores Subsection **20-16**
Code Interface Report Limitations **20-17**

21 Optimizing Generated Code

Configuring Production Code Optimizations **21-2**

Optimization Dependencies **21-5**

Optimizing Your Model with Configuration Wizard Blocks and Scripts **21-7**
Overview **21-7**
Adding a Configuration Wizard Block to Your Model **21-9**
Using Configuration Wizard Blocks **21-11**
Creating a Custom Configuration Wizard Block **21-11**

Tips for Optimizing the Generated Code **21-19**
Introduction **21-19**
Using Configuration Wizard Blocks **21-19**
Setting Hardware Implementation Parameters Correctly **21-20**
Removing Unnecessary Initialization Code **21-22**
Generating Pure Integer Code If Possible **21-23**
Disabling MAT-File Logging **21-23**
Using Virtualized Output Ports Optimization **21-24**
Controlling Signal Storage **21-25**
Using External Mode with the ERT Target **21-26**
Optimizing Generated Code Using Specified Minimum and Maximum Values **21-27**

22 Developing Models and Code That Comply with Industry Standards and Guidelines

What Are the Standards and Guidelines? **22-2**

Developing Models and Code That Comply with MAAB Guidelines **22-4**

Developing Models and Code That Comply with MISRA C Guidelines **22-5**

Developing Models and Code That Comply with the IEC 61508 Standard **22-6**

Applying Simulink and Embedded Coder to the IEC 61508 Standard 22-6
Checking for IEC 61508 Standard Compliance Using the Model Advisor 22-6
Validating Traceability 22-7

Developing Models and Code That Comply with the ISO 26262 Standard 22-8
Applying Simulink and Embedded Coder to the ISO 26262 Standard 22-8
Checking for ISO 26262 Standard Compliance Using the Model Advisor 22-8
Validating Traceability 22-8

Developing Models and Code That Comply with the DO-178B Standard 22-10
Applying Simulink and Embedded Coder to the DO-178B Standard 22-10
Checking for Standard Compliance Using the Model Advisor 22-10
Validating Traceability 22-11

23 Generating Reentrant Code from MATLAB Code

What Is Reentrant Code? 23-2

When to Generate Reentrant Code 23-3

How to Generate Reentrant Code 23-4
Prerequisites 23-4
Procedure 23-4

Generated Code API 23-5

How to Call Reentrant Code in a Single-Thread Environment 23-6

How to Call Reentrant Code in a Multithreaded Environment **23-7**
Multithreaded Examples **23-7**

Example: Calling Reentrant Code with No Persistent or Global Data (UNIX Only) **23-9**
MATLAB Code Used for This Example **23-9**
Providing a main Function **23-9**
Generating Reentrant C Code **23-12**
Examining the Generated Code **23-13**
Running the Code **23-14**

Example: Calling Reentrant Code — Multithreaded with Persistent Data (Windows Only) **23-15**
MATLAB Code Used for This Example **23-15**
Providing a main Function **23-16**
Generating Reentrant C Code **23-18**
Examining the Generated Code **23-19**
Running the Code **23-20**

Example: Calling Reentrant Code — Multithreaded with Persistent Data (UNIX Only) **23-21**
MATLAB Code Used for This Example **23-21**
Providing a main Function **23-22**
Generating Reentrant C Code **23-24**
Examining the Generated Code **23-26**
Running the Code **23-27**

24 Generating Code for AUTOSAR Software Components

Overview of AUTOSAR Support **24-2**

Simulink Modeling Patterns for AUTOSAR **24-3**
About Simulink Modeling Patterns for AUTOSAR **24-3**
AUTOSAR Software Components **24-3**
AUTOSAR Communication **24-9**

Calibration Parameters 24-15
Inter-Runnable Variables 24-16
Data Types 24-17
Per-Instance Memory 24-22
AUTOSAR Terminology 24-23

Workflow for AUTOSAR **24-26**

Importing an AUTOSAR Software Component **24-28**

Preparing a Simulink Model for AUTOSAR Code Generation **24-31**
Using the Configure AUTOSAR Interface Dialog Box 24-31
Configuring Ports for Basic Software and Error Status Receivers 24-37
Configuring Client-Server Communication 24-38
Configuring Multiple Runnables 24-47
Configuring Calibration Parameters 24-53
Using Data Store Memory Blocks to Specify Per-Instance Memory 24-55
Modifying and Validating an Existing AUTOSAR Interface 24-57

Generating AUTOSAR Code and Description Files **24-58**
Selecting an AUTOSAR Schema 24-58
Specifying Maximum SHORT-NAME Length 24-58
Configuring AUTOSAR Compiler Abstraction Macros 24-59
Root-Level Matrix I/O 24-61
Exporting AUTOSAR Software Component 24-61

Configuring AUTOSAR Options Programmatically **24-64**

Verifying the AUTOSAR Code with SIL and PIL Simulations **24-65**
Overview 24-65
Using the SIL and PIL Simulation Modes 24-65
Using a SIL or PIL Block for AUTOSAR Verification 24-66

Limitations and Tips **24-68**
Cannot Import Internal Behavior 24-68

Cannot Copy Subsystem Blocks Without Losing Interface Information 24-68
Error If No Default Configuration 24-69
The Generate Code Only Check Box 24-69
Specify Sample Time Independent Server Operation Model 24-69
Invoke AUTOSAR Server Operation Block in Referenced Model 24-69
Cannot Save Importer Objects in MAT-Files 24-69
Using the Merge Block for Inter-Runnable Variables 24-70
Using Goto and From Blocks Within Wrapper Subsystems 24-70
AUTOSAR Compiler Abstraction Macros 24-70
Intrinsic Fixed-Point Types for Model Configured as Server 24-71
Server Operation Model with Tunable Parameters 24-71
Migrating AUTOSAR Development Kit Models 24-72

Demos and Further Reading 24-73
AUTOSAR Demos 24-73
Further Reading 24-74

Integrating External Code and Generated C and C++ Code

25 About External Code Integration Extensions

26 Generating S-Function Wrappers

About S-Function Wrapper Generation 26-2

Creating a SIL Block 26-3

S-Function Wrapper Generation Limitations 26-4

27 Exporting Function-Call Subsystems

Overview 27-2
Exported Subsystems Demo 27-3
Additional Information 27-3

Requirements for Exporting Function-Call Subsystems 27-4
Requirements for All Exported Subsystems 27-4
Requirements for Exported Virtual Subsystems 27-5

Techniques for Exporting Function-Call Subsystems .. 27-7
General Workflow 27-7
Specifying a Custom Initialize Function Name 27-8
Specifying a Custom Description 27-8

Optimizing Exported Function-Call Subsystems 27-10

Exporting Function-Call Subsystems That Depend on Elapsed Time 27-11

Function-Call Subsystem Export Example 27-12

Function-Call Subsystems Export Limitations 27-16

28 Nonvirtual Subsystem Modular Function Code Generation

Overview 28-2

Configuring Nonvirtual Subsystems for Generating Modular Function Code **28-4**

Examples of Modular Function Code for Nonvirtual Subsystems **28-9**
H File Differences for Nonvirtual Subsystem Function Data Separation **28-11**
C File Differences for Nonvirtual Subsystem Function Data Separation **28-12**

Nonvirtual Subsystem Modular Function Code Limitations **28-15**

29 Controlling Generation of Function Prototypes

Overview **29-2**

Configuring Model Function Prototypes **29-4**
Launching the Model Interface Dialog Boxes **29-4**
Default Model Initialize and Step Functions View **29-4**
Model Specific C Prototypes View **29-5**
Configuring Function Prototypes for Nonvirtual Subsystems **29-9**

Model Function Prototypes Example **29-12**

Configuring Model Function Prototypes Programmatically **29-18**

Sample Script for Configuring Model Function Prototypes **29-22**

Verifying Generated Code for Customized Functions .. **29-23**

Model Function Prototype Control Limitations **29-24**

30 Controlling Generation of Encapsulated C++ Model Interfaces

Overview of C++ Encapsulation **30-2**

C++ Encapsulation Quick-Start Example **30-4**

Generating and Configuring C++ Encapsulation Interfaces to Model Code **30-11**
- Selecting the C++ (Encapsulated) Option **30-11**
- Configuring Code Interface Options **30-12**
- Configuring the Step Method for Your Model Class **30-15**
- Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems **30-19**

Configuring C++ Encapsulation Interfaces Programmatically **30-21**

Sample Script for Configuring the Step Method for a Model Class **30-24**

C++ Encapsulation Interface Control Limitations **30-26**

31 Replacing Math Functions and Operators Using Target Function Libraries

Introduction to Target Function Libraries **31-2**
- Overview of Target Function Libraries **31-2**
- Target Function Libraries General Workflow **31-7**
- Target Function Libraries Quick-Start Example **31-9**

Creating Function Replacement Tables **31-16**
- Overview of Function Replacement Table Creation **31-16**
- Creating Table Entries **31-20**

Example: Mapping Math Functions to Target-Specific Implementations 31-27
Example: Mapping the memcpy Function to a Target-Specific Implementation 31-34
Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations 31-38
Example: Mapping Scalar Operators to Target-Specific Implementations 31-43
Mapping Nonscalar Operators to Target-Specific Implementations 31-49
Mapping Fixed-Point Operators to Target-Specific Implementations 31-78
Remapping Operator Outputs to Implementation Function Input Positions 31-113
Refining TFL Matching and Replacement Using Custom TFL Table Entries 31-115
Replacing Math Functions Based on Computation Method 31-132
Specifying Build Information for Function Replacements 31-134
Adding Target Function Library Reserved Identifiers 31-137

Examining and Validating Function Replacement Tables 31-139
Overview of Function Replacement Table Validation 31-139
Invoking the Table Definition File 31-139
Using the Target Function Library Viewer to Examine Your Table 31-140
Using the Target Function Library Viewer to Examine Registered TFLs 31-141
Tracing Code Generated Using Your Target Function Library 31-143
Examining TFL Cache Hits and Misses 31-144

Registering Target Function Libraries 31-148
Overview of TFL Registration 31-148
Using the sl_customization API to Register a TFL with Simulink Software 31-149
Using the rtwTargetInfo API to Register a TFL with MATLAB® Coder Software 31-153
Registering Multiple TFLs 31-154

Target Function Library Limitations 31-156

Setting Up Generated Code To Interface With Components in the Run-Time Environment

32 Configuring the Target Hardware Environment

Configuring Support for Numeric Data 32-2

Configuring Support for Time Values 32-3

Setting Up Support for Non-Inlined S-Functions 32-4

Configuring Model Function Generation and Argument Passing 32-5

Setting Up Support for Code Reuse 32-7

Configuring Target Function Libraries 32-8

33 Model Entry Points

34 Interfacing With Hardware That is Not Running an Operating System (Bare Board)

About Standalone Program Execution 34-2

Generating a Standalone Program 34-3

Standalone Program Components 34-4

Main Program 34-5
Overview of Operation 34-5
Guidelines for Modifying the Main Program 34-5

rt_OneStep and Scheduling Considerations 34-7
Overview of Operation 34-7
Single-Rate Single-tasking Operation 34-8
Multirate Multitasking Operation 34-9
Multirate Single-Tasking Operation 34-11
Guidelines for Modifying rt_OneStep 34-12

Static Main Program Module 34-14
Overview 34-14
Rate Grouping and the Static Main Program 34-15
Modifying the Static Main Program 34-16

Rate Grouping Compliance and Compatibility Issues 34-19
Main Program Compatibility 34-19
Making Your S-Functions Rate Grouping Compliant 34-19

35 Wind River Systems VxWorks Example Main Program

Introduction to the VxWorks Example Main Program 35-2

Task Management 35-3
Overview of Operation 35-3
Single-Rate Single-tasking Operation 35-3
Multirate Multitasking Operation 35-4
Multirate Single-tasking Operation 35-4

Verifying Generated Code Applications

36 Tracing Generated Code to Requirements

About Generated Code and Requirements Traceability 36-2

Goals of Generated Code and Requirements Traceability 36-3

37 Verifying Generated Code

Traceability for Production Code Generation 37-2
About Traceability 37-2
Tracing Code to Model Objects Using Hyperlinks 37-2
Tracing Model Objects to Generated Code 37-4
Reloading Existing Traceability Information 37-6
Customizing Traceability Reports 37-8
Generating a Traceability Matrix (DO Qualification Kit or IEC Certification Kit) 37-9
Traceability Limitations 37-10

Checking Code Correctness 37-11
About Checking Code Correctness 37-11
How To Check Code Correctness 37-11

38 Rapid Prototyping On a Target System

About On-Target Rapid Prototyping 38-2

Goals of On-Target Rapid Prototyping 38-3

Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping 38-4

39 Verifying Generated Code With SIL and PIL Simulations

About SIL and PIL Simulations **39-2**
Overview **39-2**
What are SIL and PIL Simulations? **39-2**
Why Use SIL and PIL **39-3**

How SIL and PIL Simulations Work **39-6**

Comparison of SIL and PIL Simulation **39-7**

Choosing a SIL or PIL Approach **39-9**
About Choosing a SIL or PIL Simulation **39-9**
When to Use Top-Model SIL or PIL **39-9**
When to Use Model Block SIL or PIL **39-9**
When to Use the SIL or PIL Block **39-14**

Configuring a SIL or PIL Simulation **39-16**
Top-Model SIL or PIL Simulation **39-16**
Model Block SIL or PIL Simulation **39-18**
Using a SIL or PIL Block **39-20**
Verifying a SIL or PIL Configuration **39-22**
Compatible Models **39-23**

Code Coverage **39-25**
Using a Code Coverage Tool in a SIL Simulation **39-25**
Code Coverage for a PIL Simulation **39-32**
Configuring Code Coverage Programmatically **39-32**

Code Execution Profiling **39-34**
About Code Execution Profiling **39-34**
Configuring Code Execution Profiling **39-34**
Viewing and Analyzing Code Execution Profiles **39-35**

Tips and Limitations **39-39**

Running a Top Model as a SIL or PIL Simulation 39-41

Running a Referenced Model as a SIL or PIL Simulation 39-43
Verifying Internal Signals of a Component **39-43**
Simulation Mode Override Behavior in Model Reference Hierarchy **39-44**

SIL and PIL Code Interfaces 39-47
Code Interface for Top-Model SIL or PIL **39-47**
Code Interface for Model Block SIL or PIL **39-48**

Configuring Hardware Implementation Settings for SIL 39-49
Compiling Generated Code That Supports Portable Word Sizes **39-51**
Portable Word Sizes Limitations **39-51**

Programming PIL Support for Third-Party Tools and Target Hardware 39-53

Creating a Connectivity Configuration for a Target . . . 39-54
What Is a PIL Connectivity Configuration? **39-54**
Overview of the Target Connectivity API **39-55**
Creating a Connectivity API Implementation **39-58**
Registering a Connectivity API Implementation **39-58**
Demos of the Target Connectivity API **39-59**

SIL and PIL Simulation Support and Limitations 39-60
About SIL and PIL Simulation Support and Limitations . . **39-61**
Code Source Support **39-62**
Block Support **39-65**
Configuration Parameters Support **39-67**
I/O Support **39-71**
Hardware Implementation Support **39-84**
Other Feature Support **39-88**

40 Verifying a Component in the Target Environment

About Component Verification in the Target Environment 40-2

Goals of Component Verification in the Target Environment 40-3

Maximizing Code Portability and Configurability 40-4

Simplifying Code Integration and Maximizing Code Efficiency 40-5

Running Component Tests in the Target Environment 40-7

41 Verifying a Component by Building a Complete Real-Time Target Environment

About Component Verification With a Complete Real-Time Target Environment 41-2

Goals of Component Verification With a Complete Real-Time Target Environment 41-4

Testing a Component as Part of a Complete Real-Time Target Environment 41-5

42 Verifying Numerical Equivalence of Results with Code Generation Verification API

Verifying Numerical Equivalence with Code Generation Verification **42-2**
Code Generation Verification API Overview **42-2**
Verifying Numerical Equivalence with CGV Workflow . . . **42-2**
Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model **42-3**
Example of Plotting Output Signals **42-10**

Embedded IDEs and Embedded Targets

43 Project and Build Configurations

Model Setup **43-2**
Block Selection **43-2**
Target Preferences **43-4**
Configuration Parameters **43-7**
Model Reference **43-17**

IDE Projects **43-18**
Third Party Product Setup **43-18**
Installation of MathWorks Products on 64-bit Host Computers **43-20**
IDE Link Configuration **43-20**
Code Generation and Build **43-21**
Automation of IDE Tasks and Processes **43-22**

Makefiles **43-24**
Using XMakefile to Generate and Build Software **43-24**
Making an XMakefile Configuration Operational **43-31**
Example: Creating a New XMakefile Configuration **43-31**
XMakefile User Configuration Dialog Box **43-38**

44 Verification and Profiling

What Is Verification? **44-2**

Processor-in-the-Loop (PIL) Simulation **44-3**
Overview **44-3**
Approaches **44-4**
Communications **44-9**
Definitions **44-11**
PIL Issues and Limitations **44-13**

Execution Profiling **44-14**
What Is Execution Profiling? **44-14**
Execution Profiling during Standalone Execution Mode .. **44-15**
Execution Profiling during PIL Simulation **44-19**

Stack Profiling **44-21**
What is Stack Profiling? **44-21**
Profiling System Stack Use **44-22**

45 Processor-Specific Optimizations

Target Function Library (TFL) **45-2**
About Target Function Libraries and Optimization **45-2**
Using a Processor-Specific Target Function Library to Optimize Code **45-4**
Process of Determining Optimization Effects Using Real-Time Profiling Capability **45-5**
Reviewing Processor-Specific Target Function Library Changes in Generated Code **45-5**
Reviewing Target Function Library Operators and Functions **45-8**
Creating Your Own Target Function Library **45-8**

46 Working with Altium TASKING IDE

Getting Started **46-2**
Overview **46-2**
Supported Altium TASKING Toolsets **46-6**
Using This Guide **46-7**
Setting Target Preferences for Altium TASKING **46-8**
Working with Configuration Sets **46-13**
Accessing Utilities for TASKING **46-20**
Option Sets **46-24**

Components **46-27**
Project Generator **46-27**
Automation Interface **46-37**

Verification **46-50**
Processor-in-the-Loop (PIL) Simulation **46-50**
C Code Coverage Reports **46-58**
Execution Profiling **46-60**
Stack Profiling **46-63**
Bidirectional Traceability Between Code and Model **46-66**
MISRA C Rule Checking **46-67**

Optimization **46-69**
Compiler / Linker Optimization Settings **46-69**
Target Memory Placement / Mapping **46-69**
Execution and Stack Profiling **46-70**
Target Specific Optimizations **46-70**
Model Advisor **46-74**

Tutorials **46-75**
Tutorial: Using Option Sets **46-75**
Tutorial: Creating New Template Projects **46-76**
Tutorial: Configuring an Existing Model for Embedded Coder Software **46-81**

Code Generation Pane — IDE Link **46-83**
Overview **46-84**
Build Action **46-85**
Target Preference Configuration **46-87**

Add build directory suffix 46-88
Build directory suffix 46-89
Export EDE handle to MATLAB base workspace 46-90
EDE handle name 46-90
Export CrossView Pro handle to MATLAB base workspace 46-92
CrossView Pro handle name 46-92
Configure model to build PIL algorithm object code 46-94

Limitations and Tips **46-95**
General Issues 46-95
Debugger Issues 46-97
Build Process Issues 46-98
Processor-in-the-Loop Issues 46-107
Issues Using Simulink® Coder Software Without Embedded Coder Software 46-110

47 Working with Analog Devices™ VisualDSP++ IDE

Getting Started **47-2**
Overview 47-2
Software Structure and Components 47-3
Software Requirements 47-5
Installation and Configuration 47-6

Automation Interface **47-7**
Getting Started with Automation Interface 47-7
Constructing Objects 47-22
Properties and Property Values 47-23
adivdsp Object Properties 47-27

Project Generator **47-30**
Introducing Project Generator 47-30
Project Generator Tutorial 47-31
Model Reference 47-35

Reported Limitations and Tips **47-40**

Reported Issues 47-40

48 Working with Eclipse IDE

Tested Software Versions **48-2**

Installing Third-Party Software for Eclipse **48-4**
Installing Sun Java Runtime Environment (JRE) **48-4**
Installing Eclipse IDE for C/C++ Developers **48-4**
Verifying the GNU Tool Chain on Linux **48-5**
Installing the GNU Tool Chain on Windows **48-7**

Configuring Your MathWorks Software to Work with Eclipse **48-10**
Additional Configuration Steps to Run Your Executable on a Remote Embedded Linux Target **48-13**

Troubleshooting with Eclipse IDE **48-15**
SIGSEGV Segmentation Fault for GDB **48-15**
GDB Stops on Each Semaphore Post **48-15**
Build Errors **48-16**
Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux .. **48-16**
Eclipse Message: “Can’t find a source file” **48-16**
Eclipse Message: “Cannot access memory at address” **48-17**

49 Working with Freescale MPC5xx Processors

Getting Started **49-2**
Overview **49-2**
Additional Blocks on MATLAB Central Web Site **49-8**
Using This Guide **49-8**
CAN Hardware Requirements for Freescale MPC5xx **49-9**

Supported Cross-Development Tools for Freescale MPC5xx **49-9**
Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation **49-10**
Setting Target Preferences for MPC5xx **49-11**
Accessing Utilities for Freescale MPC555 .... **49-18**
Data Type Support and Scaling for Device Driver Blocks .. **49-20**

Generating Stand-Alone Real-Time Applications **49-24**
Overview **49-24**
Tutorial: Creating a New Application **49-26**
Downloading Boot and Application Code **49-39**
Parameter Tuning and Signal Logging **49-53**
HTML Code Profile (RAM/ROM) Report **49-67**
Execution Profiling **49-68**
Summary of the Real-Time Target **49-76**
Performance Tips **49-79**

PIL Simulation **49-82**
Overview of PIL Simulation **49-82**
Tutorial 1: Building and Running a PIL Simulation **49-84**
Tutorial 2: Using the Demo Model in Simulation **49-97**
PIL Target Summary **49-98**
Algorithm Export Target **49-103**
HTML Code Analysis (RAM/ROM) Report **49-104**
Algorithm Export Target Summary **49-106**

Configuration Parameters **49-109**
Code Generation Pane: ET MPC5xx (Algorithm Export) Options **49-109**
Code Generation Pane: ET MPC5xx (Processor-in-the-Loop) Options **49-111**
Code Generation Pane: ET MPC5xx Real-Time Options (1) **49-115**
Code Generation Pane: ET MPC5xx Real-Time Options (2) **49-119**

Toolchains and Hardware **49-124**
Setting Up Your Toolchain **49-124**
Setting Up Your Installation with Wind River Compiler and Wind River Systems SingleStep Debugger **49-124**

Setting Up Your Installation with Freescale CodeWarrior **49-129**
Setting Up Your Target Hardware **49-133**
CAN Hardware and Drivers **49-139**
Configuration for Nondefault Hardware **49-141**
Integrating External Blocksets **49-144**

50 Working with Green Hills® MULTI IDE

Getting Started **50-2**
Overview **50-2**
Software Structure and Components **50-3**

Automation Interface **50-10**
Getting Started with Automation Interface **50-10**
Constructing Objects **50-26**
Properties and Property Values **50-27**
ghsmulti Object Properties **50-30**

Project Generator **50-33**
Introducing Project Generator **50-33**
Project Generator Tutorial **50-34**
Model Reference **50-39**

Breakpoints and PIL **50-44**

51 Working with Infineon C166 Processors

Getting Started **51-2**
Overview **51-2**
Using This Guide **51-4**
Supported Hardware for Infineon C166 ..... **51-5**
Supported Cross-Development Tools for Infineon C166 ..... **51-7**

Switching Between Hardware Variants **51-7**
Setting Up and Verifying Your Installation **51-8**
Setting Up Your Target Hardware **51-12**
Setting C166 Target Preferences **51-14**
Code Generation Configuration for Nondefault Processors **51-15**
Supported Blocks and Data Types **51-18**
Accessing Utilities for Infineon® C166 . . . . **51-20**
Overview of C166 Options in the Configuration Parameters Dialog Box **51-20**

Tutorial: Simple Example Applications for C166 Microcontrollers **51-22**
Introduction **51-22**
Tutorial: Creating a New Application **51-22**
Debugging and Using The Code Profile Report **51-30**
Parameter Tuning and Signal Logging **51-34**

Integrating Your Own Device Drivers **51-38**
Integrating Manually Coded Device Drivers with a Simulink Model **51-38**
Preparing Input and Output Signals to the Device Driver Functions **51-39**
Calling the Device Driver Functions from c166_main.c . . . **51-41**
Adding the I/O Driver Source to the List of Files to Build . . **51-41**
Tutorial: Using the Example Driver Functions **51-43**

Custom Storage Class for C166 Microcontroller Bit-Addressable Memory **51-49**
Specifying C166 Microcontroller Bit-Addressable Memory **51-49**
Using the Bitfield Example Model **51-50**

Execution Profiling **51-56**
Overview of Execution Profiling **51-56**
Options for Execution Profiling **51-60**
Multitasking Demo Model **51-62**

Configuration Parameters **51-71**
Code Generation Pane: C166 Options **51-71**

52 Working with Linux Target

Disambiguation **52-2**

Preparing Models to Run on Linux **52-3**

Scheduler **52-4**
Base Rate **52-4**
Running Target Applications on Multicore Processors **52-4**
Avoiding Lock-Up in Free-Running, Multirate, Multitasking Models **52-6**
Limitations **52-6**

Example: Build Generated Code on a BeagleBoard Running Linux **52-7**
Overview **52-7**
Configure the Windows Host **52-7**
Configure the BeagleBoard **52-7**
Configure MATLAB **52-8**

Example: Build Generated Code on a Linux Host, Then Run It Remotely on BeagleBoard **52-9**
Overview **52-9**
Prerequisites **52-9**
Set up your environment for Linux-ARM Code Generation **52-9**
Generate Code for Linux-ARM **52-12**
External Mode Simulation **52-12**

Embedded Linux Topics **52-14**
Troubleshooting "sched_setaffinity: Bad address" Error .. **52-14**

53 Working with Microsoft Windows Target

Preparing Models to Run on Windows **53-2**

Scheduler **53-3**
Selecting the Operating System and Scheduling Mode ... **53-3**
Base Rate **53-4**
Running Target Applications on Multicore Processors **53-4**
Limitations **53-6**

54 Working with Texas Instruments Code Composer Studio IDE

Code Composer Studio **54-2**
Using Code Composer Studio with Embedded Coder Software **54-2**
Default Project Configuration **54-2**

Getting Started **54-4**
Overview **54-4**
Configuration Information **54-7**

Automation Interface **54-10**
Getting Started with Automation Interface **54-10**
Getting Started with RTDX **54-27**
Constructing ticcs Objects **54-48**
ticcs Properties and Property Values **54-50**
Overloaded Functions for ticcs Objects **54-50**
ticcs Object Properties **54-51**

Project Generator **54-58**
Introducing Project Generator **54-58**
Project Generation and Board Selection **54-58**
Project Generator Tutorial **54-60**
Model Reference **54-65**

Exporting Filter Coefficients from FDATool **54-69**
About FDATool **54-69**
Preparing to Export Filter Coefficients to Code Composer Studio Projects **54-70**

Exporting Filter Coefficients to Your Code Composer Studio Project . . . 54-74
Preventing Memory Corruption When You Export Coefficients to Processor Memory . . . 54-79

Tutorial: Using XMakefile with Code Composer Studio 4.x . . . **54-85**
Introduction . . . 54-85
Set Up XMakefile for CCSv4 . . . 54-85
Prepare Your Model for CCSv4 and Makefiles . . . 54-87
Create Target Configuration File in CCSv4 . . . 54-87
Load and Run the Embedded Software . . . 54-88

Reported Limitations and Tips . . . **54-91**
Demonstration Programs Do Not Run Properly Without Correct GEL Files . . . 54-92
Changing Values of Local Variables Does Not Take Effect . . . 54-92
Code Composer Studio Cannot Find a File After You Halt a Program . . . 54-93
C54x XPC Register Can Be Modified Only Through the PC Register . . . 54-94
Working with More Than One Installed Version of Code Composer Studio . . . 54-95
Changing CCS Versions During a MATLAB Session . . . 54-95
MATLAB Hangs When Code Composer Studio Cannot Find a Board . . . 54-95
Using Mapped Drives . . . 54-97
Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting . . . 54-97
PostCodeGenCommand Commands Do Not Affect Embedded Coder Projects . . . 54-98

55 Working with Texas Intruments C2000 Processors

Setting Up and Configuring . . . **55-2**
Installing and Configuring Software . . . 55-2
Verifying the Configuration . . . 55-3

Data Type Support **55-5**

Scheduling and Timing **55-6**
Overview **55-6**
Timer-Based Interrupt Processing **55-6**
Asynchronous Interrupt Processing **55-7**

Sharing General Purpose Timers between C281x Peripherals **55-12**
Example 1 ..... **55-13**
Example 2 ..... **55-17**

Overview of Creating Models for C2000 Processors **55-21**
Accessing the Embedded Coder Block Library **55-21**
Building Your Model **55-21**

Using the c2000lib Blockset **55-23**
Introduction **55-23**
Hardware Setup **55-23**
Starting the c2000lib Library **55-24**
Setting Up the Model **55-24**
Adding Blocks to the Model **55-26**
Generating Code from the Model **55-28**

Configuring Timing Parameters for CAN Blocks **55-29**
The CAN Blocks **55-29**
Setting Timing Parameters **55-29**
Parameter Tuning and Signal Logging **55-34**

Configuring Acquisition Window Width for ADC Blocks **55-47**
What Is an Acquisition Window? **55-47**
Configuring ADC Parameters for Acquisition Window Width **55-49**

Using the IQmath Library **55-53**
About the IQmath Library **55-53**
Fixed-Point Numbers **55-54**
Building Models **55-59**

Programming Flash Memory **55-62**
Introduction **55-62**
Installing TI Flash APIs **55-62**
Configuring the DSP Board Bootloader **55-63**
Configuring the Software for Automatic Flash Programming **55-64**
Selectively Erase, Program, or Verify Specific Flash Sectors **55-64**
Placing Additional Code or Data on Unused Flash Sectors **55-65**

Configuring LIN Communications **55-68**
Overview **55-68**
Configuring Your Model **55-68**

56 Working with Texas Instruments C6000 Processors

Getting Started **56-2**
Overview **56-2**
Using This Guide **56-2**
Configuration Information **56-3**
Setting Up and Configuring **56-4**

Targeting C6000 DSP Hardware **56-7**
Introduction to Targeting **56-7**
C6000 and Code Composer Studio IDE **56-8**
Targeting Tutorial — Single Rate Application **56-11**
Schedulers and Timing **56-21**
Model Reference and Embedded Coder Software **56-34**
Targeting Supported Boards **56-38**
Simulink Models and Targeting **56-43**
Targeting Tutorial II — A More Complex Application **56-43**
Targeting Your C6713 DSK and Other Hardware **56-50**
Creating Code Composer Studio Projects Without Building **56-54**
Targeting Custom Hardware **56-55**
Using Embedded Coder Software **56-69**

Targeting with DSP/BIOS Options **56-71**
Introducing DSP/BIOS **56-71**
DSP/BIOS and Targeting Your C6000 DSP **56-72**
Code Generation with DSP/BIOS **56-75**
Profiling Generated Code **56-79**
Using DSP/BIOS with Your Target Application **56-92**
Generating Code for Any C64x+ Processor or Board **56-93**

Using the C62x and C64x DSP Libraries **56-99**
About the C62x and C64x DSP Libraries **56-99**
Fixed-Point Numbers **56-101**
Building Models **56-106**

Configuring Timing Parameters for CAN Blocks **56-109**
Setting Timing Parameters **56-109**

Hardware Issues **56-113**
Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP **56-113**
Requirements for the DM642 EVM **56-113**
Installing and Configuring the Avnet Board Support Library **56-116**
Continuing Issues with Embedded Coder Software **56-118**

57 Working with Wind River VxWorks Target

Overview of Support for Wind River VxWorks **57-2**

Tutorial: Building and Running Embedded Software on VxWorks **57-4**
Install and Set Up the Wind River Development Environment **57-4**
Setting VxWorks Environment Variables and Starting MATLAB **57-5**
Setting Up XMakefile for VxWorks **57-6**
Customizing XMakefile to Automatically Download and Build Your Software **57-7**
Prepare Your Model for VxWorks and Makefiles **57-8**

Build Your Embedded Software 57-8

Generating Code for VxWorks Running on Other Targets **57-9**

Schedulers **57-10**
Running Target Applications on Multicore Processors 57-10

A Examples

Code Generation **A-2**

Custom Storage Classes **A-2**

Memory Sections **A-2**

Advanced Code Generation **A-3**

Target Function Libraries **A-3**

Data Structures, Code Modules, and Program Execution **A-4**

Verifying Generated Code **A-4**

Makefiles **A-4**

Verification **A-4**

Tutorials **A-4**

Automation Interface **A-4**

Working with adivdsp Objects A-5

Project Generator A-5

Real-Time Target A-5

Processor-in-the-Loop Target A-5

Algorithm Export Target A-6

Working with ghsmulti Objects A-6

Simple Example Applications A-6

Integrating Manually Coded Device Drivers A-6

Bit-Addressable Memory A-6

Execution Profiling A-7

Working with ticcs Objects A-7

Exporting Filter Coefficients from FDATool A-7

Q Format Examples A-7

Targeting Tutorials A-7

Asynchronous Scheduler A-8

Profiling Code A-8

Target Preferences A-8

Index

1

Introduction to the Embedded Coder Product

The Embedded Coder™ product *extends* the Simulink® Coder™ product with features that are important for embedded software development. Using the Embedded Coder add-on product, you gain access to all aspects of Simulink Coder technology and can generate code that has the clarity and efficiency of professional handwritten code. For example, you can

- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems
- Customize the appearance of the generated code
- Optimize the generated code for a specific target environment
- Integrate existing (legacy) applications, functions, and data
- Enable tracing, reporting, and testing options that facilitate code verification activities

For detailed information on how the Embedded Coder product fits into the complete Simulink Coder technology picture, see “Product Overview” in the Simulink Coder documentation. That topic positions the Embedded Coder in terms of what you can accomplish with it, how it can fit into your development process, how you might apply it to the V-model for system development, and how you might apply it to relevant use cases.

Because Embedded Coder extends Simulink Coder for code generation, to use the Embedded Coder product effectively, you should be familiar with information in parts of the Simulink Coder documentation that align with corresponding parts in the Embedded Coder documentation.

- “Introduction to Code Generation Technology”
- “Developing Models for Code Generation”
- “Defining Data Representation and Storage for Code Generation”
- “Preparing Models for Code Generation”
- “Generating Code and Building Executables”
- “Integrating External Code With Generated C and C++ Code”
- “Setting Up Generated Code To Interface With Components in the Run-Time Environment”
- “Verifying Generated Code Applications”

Bug Reports

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at http://www.mathworks.com/support/bugreports/. Use the **Saved Searches and Watched Bugs** tool with the search phrase “Incorrect Code Generation” to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Developing Models for Code Generation

- Chapter 3, "Setting Up Your Modeling Environment"
- Chapter 4, "Architecture Considerations"
- Chapter 5, "Scheduling Considerations"
- Chapter 6, "Developing Model Patterns that Generate Specific C Constructs"

Setting Up Your Modeling Environment

When developing a system, it is important to use the correct combination of products to model each system component based on the domain to which it applies.

The following table guides you to information and demos that pertain to use of the Embedded Coder product to meet goals for specific domains.

Goals	Related Product Information	Demos
Generate a software design description	Simulink® Report Generator™ Simulink Report Generator documentation	`rtwdemo_codegenrpt`
Trace model requirements to generated code	Simulink® Verification and Validation™ "Including Requirements Information with Generated Code" in the Simulink Verification and Validation documentation	`rtwdemo_requirements`

Goals	Related Product Information	Demos

Implement application on fixed-point processors	Simulink® Fixed Point™
	
	“Data Types and Scaling” and “Code Generation” in the Simulink Fixed Point documentation
		`rtwdemo_fixpt1`
		`rtwdemo_fuelsys_fxp_publish`
		`rtwdemo_dspanc_fixpt`

Use an integrated development environment (IDE) to integrate an application on a target processor automatically	Embedded Targets topics in the Embedded Coder documentation Desktop Targets topics in the Simulink Coder documentation	In `rtwdemos`, select one of the following folders: **Desktop IDEs**, **Desktop Targets**, **Embedded IDEs**, or **Embedded Targets**

4

Architecture Considerations

- "Generating Code for Variant Systems" on page 4-2
- "Creating and Using Host-Based Shared Libraries" on page 4-17

Generating Code for Variant Systems

In this section...

"Overview" on page 4-2
"Why Generate Code for Variant Systems?" on page 4-3
"How to Generate Preprocessor Conditionals for Variant Systems" on page 4-3
"Reviewing Code Variants in the Code Generation Report" on page 4-6
"Example of Model Variants in the Generated Code" on page 4-7
"Example of Variant Subsystems in the Generated Code" on page 4-9
"Restrictions on Code Generation of a Variant Subsystem" on page 4-13
"Special Considerations for Generating Preprocessor Conditionals" on page 4-14
"Limitations on Generating Code for Variants" on page 4-15
"Exceptions to Conditionally Compiled Components in the Generated Code" on page 4-15
"Demos for Generating Code for Variants" on page 4-16

Overview

The Embedded Coder software generates code from a Simulink® model containing one or more Model Variants blocks or Variant Subsystem blocks. To learn how to create a model containing variant blocks, see "Modeling Variant Systems" in the Simulink documentation.

By default, the Simulink Coder software generates code for the active variant. The Embedded Coder software can generate code for only the active variant or for all variants. Code generated for all variants is surrounded by C preprocessor conditionals, `#if`, `#elif`, and `#endif`. Therefore, the active variant is selected at C compile time and the preprocessor conditionals determine which sections of the code to execute.

Why Generate Code for Variant Systems?

When you implement variants in the generated code, you can:

- Reuse generated code from a set of application models that share functionality with minor variations.
- Share generated code with a third party that activates one of the variants in the code.
- Validate all of the supported variants for a model and then choose to activate one variant for a particular application, without regenerating and revalidating the code.

How to Generate Preprocessor Conditionals for Variant Systems

Defining Variant Control Variables and Variant Objects for Generating Code

To learn about variant control variables and variant objects, see Variant Objects in the Simulink documentation. Variant control variables used for code generation have additional requirements than variant control variables used for simulation. Perform the following steps to define variant control variables for generating code.

1. Open the Model Explorer and click the **Base Workspace**.

2. A variant control variable can be a `Simulink.Parameter` object or a `mpt.Parameter` object. In the Model Explorer, select **Add** and choose either **Simulink Parameter** or **MPT Parameter**. Specify a name for the new parameter.

3. On the `Simulink.Parameter` or `mpt.Parameter` property dialog box, specify the **Value** and **Data type**.

4. Specify the **Storage class** parameter by choosing one of the following:

   - `ImportedDefine(Custom)` custom storage class.
   - `CompilerFlag(Custom)` custom storage class.

- A user-defined storage class created using the Custom Storage Class Designer. Your storage class must have the **Data initialization** parameter set to `Macro` and the **Data scope** parameter set to `Imported`. See "Using the Custom Storage Class Designer" on page 8-12 for more information.

5. Specify the value of the variant control variable. If the storage class is either `ImportedDefine(Custom)` or a user-defined custom storage class, do the following:

   a. Specify the **Header File** parameter as an external header file in the Custom Attributes section of the `Simulink.Parameter` property dialog box.

   b. Supply the values of the variant control variables in the external header file.

Note The generated code refers to a variant control variable as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control variable determines the active variant in the compiled code.

If the variant control variable is a `CompilerFlag` custom storage class the value of the variant control variable is set at compile time. On the **Code Generation > General** pane of the Configuration Parameters dialog box, add a makefile option to the "Make command" parameter. For example, for variant control variable, `MODE`, enter `make_rtw OPTS="-DMODE=1"` in the **Make command** field.

Note If you want to modify the value of the variant control variable after generating the makefile, use a makefile option when compiling your code. For example, at a command line outside of MATLAB, enter:

```
makecommand -f model.mk OPTS="-DMODE=1"
```

---

6. Follow the instructions for "Creating Variant Objects" to implement variant objects for code generation. Ensure that only one variant object is active in the generated code by implementing the condition expressions of the variant objects such that only one evaluates to `true`. The generated code includes a test of the variant objects to determine that there is only one active variant. If this test fails, your code will not compile.

Note You can define the variant object condition values using `Simulink.Parameter` object of enumerated type. This provides meaningful names and improves the readability of the conditions. The generated code includes preprocessor conditionals to check that the variant object condition contains valid values of the enumerated type.

Configure Your Model for Generating Preprocessor Conditional Directives

In order to generate preprocessor conditional directives configure your model as follows:

1. On the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box, select **Inline parameters**.

2. On the **Code Generation** pane of the Configuration Parameter dialog box, clear "Ignore custom storage classes". In order to generate preprocessor conditionals, you must use custom storage classes.

3. On the **Interface** pane of the Configuration Parameter dialog box, select the `Use Local Settings` option of the **Generate preprocessor conditionals** parameter. This parameter is a global setting for the parent model. This setting enables the **Generate preprocessor conditionals** parameter located in the Model Variants block parameters dialog box or Variant Subsystem parameters dialog box. See "Generate preprocessor conditionals" for more information.

4. Open the Model Variants block parameters dialog box or the Variant Subsystem block parameter dialog box, depending on your application. Select the **Generate preprocessor conditionals** parameter. If the block parameters dialog box was already open, close and reopen the dialog box to see the enabled **Generate preprocessor conditionals** parameter.

5. Clear the parameter, **Override variant conditions and use following variant**.

Build Your Model

After configuring your model to generate code, build your model.

Reviewing Code Variants in the Code Generation Report

The Code Variants Report displays a list of the variant objects in alphabetical order and their condition. The report also lists the model blocks that have Variants, and the referenced models that use them. In the **Contents** section of the code generation report, click the link to the Code Variants Report:

Code Variants Report for rtwdemo_preprocessor

Table of Contents

- Variant Objects
- Model Blocks that have Variants

Variant Objects

[-]

Variant	Condition	Used in Model Blocks
LINEAR	MODE == 0	<Root>/Left Controller
		<Root>/Right Controller
NONLINEAR	MODE == 1	<Root>/Left Controller
		<Root>/Right Controller

Model Blocks that have Variants

[-]

Model Block	Variant	Referenced Model
<Root>/Left Controller	LINEAR	rtwdemo_linl
	NONLINEAR	rtwdemo_nlinl
<Root>/Right Controller	LINEAR	rtwdemo_linr
	NONLINEAR	rtwdemo_nlinr

Example of Model Variants in the Generated Code

To open a model for generating preprocessor conditionals, enter rtwdemo_preprocessor.

After building the model, look at the variants in the generated code. `rtwdemo_preprocessor_types.h` includes the following:

- Call to external header file, `rtwdemo_preprocessor_macros.h`, which contains the macro definition for the variant control variable, `MODE`.

  ```
  /* Includes for objects with custom storage classes. */
  #include "rtwdemo_importedmacros.h"
  ```

- Preprocessor directives defining the variant objects, `LINEAR` and `NONLINEAR`. The values of these macros depend on the value of the variant control variable, `MODE`. The condition expression associated with each macro, LINEAR and NONLINEAR, determine the active variant.

  ```
  /* Model Code Variants */
  #ifndef LINEAR
  #define LINEAR                         (MODE == 0)
  #endif

  #ifndef NONLINEAR
  #define NONLINEAR                      (MODE == 1)
  #endif
  ```

- A check to ensure that exactly one variant is active at a time:

  ```
  /* Exactly one variant for '<Root>/Left Controller' should be active */
  #if (LINEAR) + (NONLINEAR) != 1
  #error Exactly one variant for '<Root>/Left Controller' should be active
  #endif
  ```

Calls to the step and initialization functions are conditionally compiled as shown in a portion of the step function, `rtwdemo_preprocessor_step`, in `ModRefVar.c`:

```
#if LINEAR

/* ModelReference: '<Root>/Left Controller' */
mr_rtwdemo_linl(&rtb_Add, &rtb_LeftController_merge_1,
                &(rtwdemo_preprocessor_DWork.LeftController_1_DWORK1.rtdw));

#elif NONLINEAR
```

```
/* ModelReference: '<Root>/Left Controller' */

mr_rtwdemo_nlinl(&rtb_Add, &rtb_LeftController_merge_1,
                 &(rtwdemo_preprocessor_DWork.LeftController_2_DWORK1.rtdw));

#endif                        /* LINEAR */
```

and

```
#if LINEAR

/* ModelReference: '<Root>/Right Controller' */
mr_rtwdemo_linr(&rtb_Add1, &rtb_RightController_merge_1,
                &(rtwdemo_preprocessor_DWork.RightController_1_DWORK1.rtdw));

#elif NONLINEAR

/* ModelReference: '<Root>/Right Controller' */
mr_rtwdemo_nlinr(&rtb_Add1, &rtb_RightController_merge_1,
                 &(rtwdemo_preprocessor_DWork.RightController_2_DWORK1.rtdw));

#endif                                 /* LINEAR */
```

Example of Variant Subsystems in the Generated Code

Open the Example Model

Open model, `AutoSSVar.mdl`, which contains a variant subsystem.

Define the Variant Control Variables

To recreate the variant control variables specifically for code generation:

1. Open the Model Explorer and click the **Base Workspace**.
2. A variant control variable can be a `Simulink.Parameter` object or a `mpt.Parameter` object. In the Model Explorer, remove the current variant control variables, `EMIS` and `FUEL`, and re-create them as

`Simulink.Parameter` objects. Select **Add** and **Simulink Parameter** to create two variant control variables, `EMIS` and `FUEL`.

3. In the `Simulink.Parameter` property dialog box, specify the **Value** as 1 and the **Data type** as `int8` for both `EMIS` and `FUEL`.

4. Specify the **Storage class** parameter for both FUEL and EMIS as `ImportedDefine(Custom)`:

5. Specify the value of the variant control variable. Because the storage class is `ImportedDefine(Custom)`, specify the **Header File** parameter as an external header file, `AutoSSVar_variables.h`, in the **Custom Attributes** section of the `Simulink.Parameter` property dialog box.

6. Supply the values of the variant control variables in the external header file.

```
#define FUEL 1
#define EMIS 1
```

Note The generated code refers to a variant control variable as a user-defined macro. The generated code does not contain the value of the macro. The value of the variant control variable determines the active variant in the compiled code.

---

7. The variant objects for this model already reside in the base workspace. For information on how to create the variant objects, follow the instructions for “Creating Variant Objects”. Ensure that only one variant object is active in the generated code by implementing the condition expressions of the variant objects such that only one evaluates to `true`. The generated code includes a test of the variant objects to determine that there is only one active variant. If this test fails, your code does not compile.

Make Each Child Subsystem an Atomic Subsystem

1. Double-click the Variant Subsystem block, `Engine`, to display the child subsystems.

2. For each child subsystem, right-click the subsystem and select **Subsystem Parameters** from the list. The Block parameters dialog box opens.

3. To specify each child subsystem as an atomic subsystem, in the Block parameters dialog box, select the **Treat as atomic unit** parameter.

Configure Your Model for Generating Preprocessor Conditional Directives

In order to generate preprocessor conditional directives configure your model as follows:

1. On the **Code Generation** pane of the Configuration Parameter dialog box, specify the **System target file** parameter as `ert.tlc` and clear "Ignore custom storage classes". In order to generate preprocessor conditionals, you must use custom storage classes.

2. On the **Optimization > Signals and Parameters**MATLAB® pane of the Configuration Parameters dialog box, select **Inline parameters**.

3. On the **Code Generation > Interface** pane of the Configuration Parameter dialog box, select the `Enable All` option of the **Generate preprocessor conditionals** parameter. This parameter is a global setting for the parent model and enables generating preprocessor conditionals for all variants of all variant blocks in the model. For more information, see "Generate preprocessor conditionals".

4. On the **Code Generation > Report** pane of the Configuration Parameter dialog box, select **Create code generation report**.

View the Generated Code

The generated code contains all child subsystems of the Variant Subsystem block protected by C preprocessor conditionals. In this case, the selection of the active variant (subsystem) is deferred until the generated code is compiled. Only one variant object, which is encoded in C macros, must evaluate to true.

After building the model, look at the variants in the generated code. `AutoSSVar_types.h` includes the following:

- Call to external header file, `AutoSSVar_variables.h`, which contains the macro definitions for the variant control variables, `FUEL` and `EMIS`.

```
/* Includes for objects with custom storage classes. */
   #include "AutoSSVar_variables.h"
```

- Preprocessor directives defining the variant objects. The values of these macros depend on the value of the variant control variables, `FUEL` and `EMIS`. The condition expression associated with each macro determine the active variant.

```
/* Model Code Variants */
#ifndef DE
#define DE                              ((FUEL == 2) && (EMIS == 2))
#endif

#ifndef DU
#define DU                              ((FUEL == 2) && (EMIS == 1))
#endif

#ifndef GE
#define GE                              ((FUEL == 1) && (EMIS == 2))
#endif

#ifndef GU
#define GU                              ((FUEL == 1) && (EMIS == 1))
#endif
```

- A check to ensure that exactly one variant is active at a time:

```
/* Exactly one variant for '<Root>/Engine' should be active */
#if (GU) + (GE) + (DU) + (DE) != 1
#error Exactly one variant for '<Root>/Engine' should be active
#endif
```

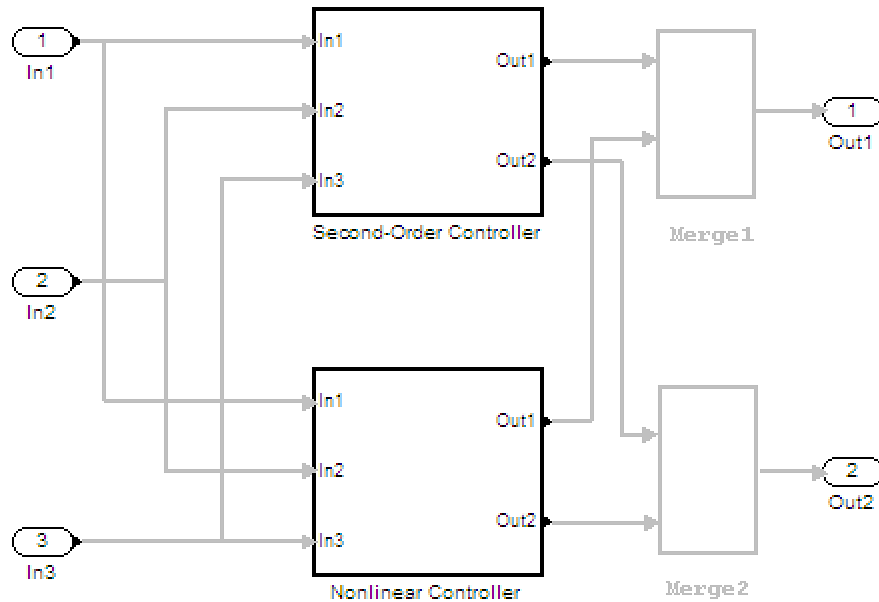
Calls to the step and initialization functions are conditionally compiled as shown in a portion of the step function, `AutoSSVar_step`, in `AutoSSVar.c`:

```
#if DE

  rtb_MergeForOutportOut1 = 2.2 * AutoSSVar_U.In1;

#elif DU

  rtb_MergeForOutportOut1 = 2.1 * AutoSSVar_U.In1;

#elif GE

  rtb_MergeForOutportOut1 = 1.2 * AutoSSVar_U.In1;

#elif GU

  rtb_MergeForOutportOut1 = 1.1 * AutoSSVar_U.In1;

#endif                                /* DE */
```

Restrictions on Code Generation of a Variant Subsystem

To generate preprocessor conditionals, the types of blocks that you can place within the child subsystems of a Variant Subsystem block are limited. Connections are not allowed in the Variant Subsystem block diagram. However, during the code generation process, one Merge block is placed at the input of each Outport block within the Variant Subsystem block diagram. All of the child subsystems connect to each of the Merge blocks.

In the following example, the code generation process makes the following connections and adds Merge blocks to the sldemo_variant_subsystems.



The restrictions placed on Merge blocks apply to the contents of the Variant Subsystem blocks. Furthermore, variant subsystems cannot contain continuous states. The restriction checks are performed only when generating code. In addition, the child subsystems of the Variant Subsystem block must be atomic subsystems. In the Subsystem block parameters dialog box, select the **Treat as atomic unit** parameter.

Special Considerations for Generating Preprocessor Conditionals

When you select the **Generate preprocessor conditionals** parameter, consider the following:

- The code generation process checks that the inports and outports of a Variant Subsystem block or a Model Variants block must be identical (same port numbers and names) to the corresponding inports and outports of its variants. The build process for simulation does not make this check.

Therefore, if your variant block contains mismatched inports or outports, the code generation process issues an error for a model that previously ran without error.

- The code generation process checks that there is at least one active variant by using the variant control variable values stored in the base workspace. If you are generating preprocessor conditionals and using an external header file for the values of the variant control variables, the code generator issues an error if the values in the base workspace do not indicate an active variant.
- If you comment out any child subsystems listed in the Variant Choices table in the Variant Subsystem block parameter dialog box, the code generator does not generate code for the commented out subsystems.

Limitations on Generating Code for Variants

When you are generating code for Model Variants blocks and Variant Subsystem blocks, the blocks cannot have:

- Continuous states or mass matrices
- Function call ports
- Non-mergeable output
- Outports with constant sample time

In addition, the Model Variants block and all of its referenced models must have the same number of inports and outports. The Variant Subsystem block and all of its active child subsystems must have the same number of inports and outports. All of the port numbers and names for each active child subsystem in a Variant Subsystem block must also match.

Exceptions to Conditionally Compiled Components in the Generated Code

The following components in the generated code are not conditionally compiled. This is true even if they are referenced only by code for variant subsystems or models that are conditionally compiled.

- `rtModel` data structure fields

- `#include`'s of utility files
- Global non-constant parameter structure fields; when the configuration parameter **Optimization > Signals and Parameters > Parameter structure** is set to `NonHeirarchical`
- Global constant parameter structure fields that are referenced by multiple subsystems activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are referenced by multiple subsystems that are activated by different variants
- Parameters that are configured to use an imported, exported, or custom code generation storage class, and are used by variant model blocks

Demos for Generating Code for Variants

To construct model reference variants and generate preprocessor directives in the generated code, see the demo rtwdemo_preprocessor_script.

To construct variant subsystems and generate preprocessor directives in the generated code, see the demo rtwdemo_preprocessor_subsys_script.

Creating and Using Host-Based Shared Libraries

In this section...

"Overview" on page 4-17

"Generating a Shared Library Version of Your Model Code" on page 4-18

"Creating Application Code to Load and Use Your Shared Library File" on page 4-19

"Host-Based Shared Library Limitations" on page 4-23

Overview

The Embedded Coder product provides an ERT target, `ert_shrlib.tlc`, for generating a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code that is appropriate for your host platform, either a Windows® dynamic link library (`.dll`) file or a UNIX® shared object (`.so`) file. This feature can be used to package your source code securely for easy distribution and shared use. The generated `.dll` or `.so` file is shareable among different applications and upgradeable without having to recompile the applications that use it.

Code generation for the `ert_shrlib.tlc` target exports

- Variables and signals of type `ExportedGlobal` as data
- Real-time model structure (*`model`*`_M`) as data
- Functions essential to executing your model code

To view a list of symbols contained in a generated shared library file, you can

- On Windows, use the Dependency Walker utility, downloadable from http://www.dependencywalker.com
- On UNIX, use `nm -D` *`model`*`.so`

To generate and use a host-based shared library, you

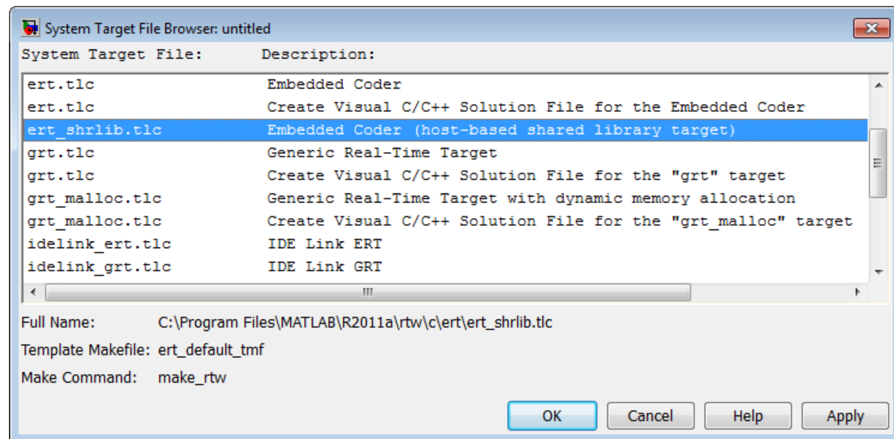
1 Generate a shared library version of your model code

2 Create application code to load and use your shared library file

Generating a Shared Library Version of Your Model Code

This section summarizes the steps needed to generate a shared library version of your model code.

1 To configure your model code for shared use by applications, open your model and select the `ert_shrlib.tlc` target on the **Code Generation** pane of the Configuration Parameters dialog box. Click **OK**.



Selecting the `ert_shrlib.tlc` target causes the build process to generate a shared library version of your model code into your current working folder. The selection does not change the code that is generated for your model.

2 Build the model.

3 After the build completes, you can examine the generated code in the model subfolder, and the `.dll` file or `.so` file that has been generated into your current folder.

Creating Application Code to Load and Use Your Shared Library File

To illustrate how application code can load an ERT shared library file and access its functions and data, MathWorks provides the demo model `rtwdemo_shrlib`. Clicking the blue button in the demo model runs a script that:

1. Builds a shared library file from the model (for example, `rtwdemo_shrlib_win32.dll` on 32-bit Windows)
2. Compiles and links an example application, `rtwdemo_shrlib_app`, that will load and use the shared library file
3. Executes the example application

Note It is recommended that you change directory to a new or empty folder before running the `rtwdemo_shrlib` script.

The demo model uses the following example application files, which are located in *`matlabroot`*`/toolbox/rtw/rtwdemos/shrlib_demo`.

File	Description
`rtwdemo_shrlib_app.h`	Example application header file
`rtwdemo_shrlib_app.c`	Example application that loads and uses the shared library file generated for the demo model
`run_rtwdemo_shrlib_app.m`	Script to compile, link, and execute the example application

You can view each of these files by clicking white buttons in the demo model window. Additionally, running the script places the relevant source and generated code files in your current folder. The files can be used as templates for writing application code for your own ERT shared library files.

The following sections present key excerpts of the example application files.

Example Application Header File

The example application header file `rtwdemo_shrlib_app.h` contains type declarations for the demo model's external input and output.

```
#ifndef _APP_MAIN_HEADER_
#define _APP_MAIN_HEADER_

typedef struct {
    int32_T Input;
} ExternalInputs_rtwdemo_shrlib;

typedef struct {
    int32_T Output;
} ExternalOutputs_rtwdemo_shrlib;

#endif /*_APP_MAIN_HEADER_*/
```

Example Application C Code

The example application `rtwdemo_shrlib_app.c` includes the following code for dynamically loading the shared library file. Notice that, depending on platform, the code invokes Windows or UNIX library commands.

```
#if (defined(_WIN32)||defined(_WIN64)) /* WINDOWS */
#include <windows.h>
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

#else /* UNIX */
#include <dlfcn.h>
#define GETSYMBOLADDR dlsym
#define LOADLIB dlopen
#define CLOSELIB dlclose

#endif

int main()
{
    void* handleLib;
```

```
...
#if defined(_WIN64)
    handleLib = LOADLIB("./rtwdemo_shrlib_win64.dll");
#else
#if defined(_WIN32)
    handleLib = LOADLIB("./rtwdemo_shrlib_win32.dll");
#else /* UNIX */
    handleLib = LOADLIB("./rtwdemo_shrlib.so", RTLD_LAZY);
#endif
#endif
...
    return(CLOSELIB(handleLib));
}
```

The following code excerpt shows how the C application accesses the demo model's exported data and functions. Notice the hooks for adding user-defined initialization, step, and termination code.

```
    int32_T i;
 ...
    void (*mdl_initialize)(boolean_T);
    void (*mdl_step)(void);
    void (*mdl_terminate)(void);

    ExternalInputs_rtwdemo_shrlib (*mdl_Uptr);
    ExternalOutputs_rtwdemo_shrlib (*mdl_Yptr);

    uint8_T (*sum_outptr);
...
#if (defined(LCCDLL)||defined(BORLANDCDLL))
    /* Exported symbols contain leading underscores when DLL is linked with
       LCC or BORLANDC */
    mdl_initialize =(void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
                     "_rtwdemo_shrlib_initialize");
    mdl_step       =(void(*)(void))GETSYMBOLADDR(handleLib ,
                     "_rtwdemo_shrlib_step");
    mdl_terminate  =(void(*)(void))GETSYMBOLADDR(handleLib ,
                     "_rtwdemo_shrlib_terminate");
    mdl_Uptr       =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                     "_rtwdemo_shrlib_U");
```

```
    mdl_Yptr       =(ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                      "_rtwdemo_shrlib_Y");
    sum_outptr     =(uint8_T*)GETSYMBOLADDR(handleLib , "_sum_out");
#else
    mdl_initialize =(void(*)(boolean_T))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_initialize");
    mdl_step       =(void(*)(void))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_step");
    mdl_terminate  =(void(*)(void))GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_terminate");
    mdl_Uptr       =(ExternalInputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_U");
    mdl_Yptr       =(ExternalOutputs_rtwdemo_shrlib*)GETSYMBOLADDR(handleLib ,
                      "rtwdemo_shrlib_Y");
    sum_outptr     =(uint8_T*)GETSYMBOLADDR(handleLib , "sum_out");
#endif

    if ((mdl_initialize && mdl_step && mdl_terminate && mdl_Uptr && mdl_Yptr &&
         sum_outptr)) {
        /* === user application initialization function === */
        mdl_initialize(1);
        /* insert other user defined application initialization code here */

        /* === user application step function === */
        for(i=0;i<=12;i++){
            mdl_Uptr->Input = i;
            mdl_step();
            printf("Counter out(sum_out): %d\tAmplifier in(Input): %d\tout(Output): %d\n",
                   *sum_outptr, i, mdl_Yptr->Output);
            /* insert other user defined application step function code here */
        }

        /* === user application terminate function === */
        mdl_terminate();
        /* insert other user defined application termination code here */
    }
    else {
        printf("Cannot locate the specified reference(s) in the shared library.\n");
        return(-1);
    }
```

Example Application Script

The application script `run_rtwdemo_shrlib_app.m` loads and rebuilds the demo model, and then compiles, links, and executes the demo model's shared library target file. You can view the script source file by opening `rtwdemo_shrlib` and clicking the appropriate white button. The script constructs platform-dependent command strings for compilation, linking, and execution that may apply to your development environment. To run the script, click the blue button.

Host-Based Shared Library Limitations

The following limitations apply to using ERT host-based shared libraries:

- Code generation for the `ert_shrlib.tlc` target exports only the following as data:
  - Variables and signals of type `ExportedGlobal`
  - Real-time model structure (*`model`*`_M`)
- Code generation for the `ert_shrlib.tlc` target supports the C language only (not C++). When you select the `ert_shrlib.tlc` target, language selection is greyed out on the **Code Generation** pane of the Configuration Parameters dialog box.
- On Windows systems, the `ert_shrlib` target by default does not generate or retain the `.lib` file for implicit linking (explicit linking is preferred for portability).

  You can change the default behavior and retain the `.lib` file by modifying the corresponding template makefile (TMF). If you do this, be aware that the generated *`model`*`.h` file will need a small modification to be used together with the generated `ert_main.c` for implicit linking. For example, if you are using Visual C++®, you will need to declare `__declspec(dllimport)` in front of all data to be imported implicitly from the shared library file.
- To reconstruct a model simulation using a generated host-based shared library, the application author must maintain the timing between system and shared library function calls in the original application. The timing needs to be consistent to ensure correct simulation and integration results.

Scheduling Considerations

- "Using Discrete and Continuous Time" on page 5-2
- "Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets" on page 5-4

Using Discrete and Continuous Time

In this section...

"Generating Code for Discrete and Continuous Time Blocks" on page 5-2

"Generating Code that Supports Continuous Solvers" on page 5-2

"Generating Code that Honors a Stop Time" on page 5-3

Generating Code for Discrete and Continuous Time Blocks

The ERT target supports code generation for discrete and continuous time blocks. If the **Support continuous time** option is selected, you can use any such blocks in your models, without restriction.

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

```
showblockdatatypetable
```

Generating Code that Supports Continuous Solvers

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

Note Custom targets must be modified to support continuous time. The required modifications are described in "Custom Targets" in the Simulink Coder documentation.

Generating Code that Honors a Stop Time

The ERT target supports the stop time for a model. When generating host-based executables, the stop time value is honored when any one of the following is true:

- **GRT compatible call interface** is selected on the **Interface** pane
- **External mode** is selected in the **Data exchange** subpane of the **Interface** pane
- **MAT-file logging** is selected on the **Interface** pane

Otherwise, the executable runs indefinitely.

Note The ERT target provides both generated and static examples of the `ert_main.c` file. The `ert_main.c` file controls the overall model code execution by calling the *`model`*`_step` function and optionally checking the `ErrorStatus`/`StopRequested` flags to terminate execution. For a custom target, if you provide your own custom static `main.c`, you should consider including support for checking these flags.

Optimizing Task Scheduling for Multirate Multitasking Models on RTOS Targets

In this section...

"Overview" on page 5-4

"Using rtmStepTask" on page 5-5

"Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target" on page 5-5

"Suppressing Redundant Scheduling Calls" on page 5-6

Overview

Using the `rtmStepTask` macro, targets that employ the task management mechanisms of an RTOS can eliminate certain redundant scheduling calls during the execution of tasks in a multirate, multitasking model, thereby improving performance of the generated code.

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

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

The event flags indicate whether or not a given task is scheduled for execution. For a multirate, multitasking model, the event flags are maintained by code in the model's generated example main program (`ert_main.c`) . For each task, the code maintains a task counter. When the counter reaches 0, indicating that the task's sample period has elapsed, the event flag for that task is set.

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

For bareboard targets that cannot rely on an external RTOS, the event flags are mandatory to allow overlapping task preemption. However, an RTOS target uses the operating system itself to manage overlapping task preemption, making the maintenance of the event flags redundant.

Using rtmStepTask

The `rtmStepTask` macro is defined in `model.h` and its syntax is as follows:

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

The arguments are:

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

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

If your target supports the **Generate an example main program** parameter, you can generate calls to `rtmStepTask` using the TLC function `RTMTaskRunsThisBaseStep`.

Task Scheduling Code for Multirate Multitasking Model on Wind River Systems VxWorks Target

The following task scheduling code, from `ertmainlib.tlc`, is designed for multirate multitasking operation on a Wind River® Systems VxWorks® target. The example uses the TLC function `RTMTaskRunsThisBaseStep` to generate calls to the `rtmStepTask` macro. A loop iterates over each subrate task, and `rtmStepTask` is called for each task. If `rtmStepTask` returns `TRUE`, the VxWorks `semGive` function is called, and the VxWorks RTOS schedules the task to run.

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

Suppressing Redundant Scheduling Calls

Redundant scheduling calls are still generated by default for backward compatibility. To change this setting and suppress them, add the following TLC variable definition to your system target file before the `%include "codegenentry.tlc"` statement:

```
%assign SuppressSetEventsForThisBaseRateFcn = 1
```

6

Developing Model Patterns that Generate Specific C Constructs

- "About Modeling Patterns" on page 6-2
- "Standard Methods to Prepare a Model for Code Generation" on page 6-3
- "Types, Operators, and Expressions" on page 6-8
- "Control Flow" on page 6-25
- "Functions" on page 6-65
- "Preprocessor Directives" on page 6-77
- "Structures" on page 6-81
- "Arrays" on page 6-95
- "Pointers" on page 6-99

About Modeling Patterns

Several standard methods are available for setting up a model to generate specific C Constructs in your code. For preparing your model for code generation, some of these methods include: configuring signals and ports, initializing states, and setting up configuration parameters for code generation. Depending on the components of your model, some of these methods are optional. Methods for configuring a model to generate specific C constructs are organized by category, for example, the Control Flow category includes constructs `if-else`, `switch`, `for`, and `while`. Refer to the name of a construct to see how you should configure blocks and parameters in your model. Different modeling methodologies are available, such as Simulink blocks, Stateflow® charts, and MATLAB Function blocks, to implement a C construct.

Model examples have the following naming conventions:

Model Components	Naming Convention
Inputs	`u1`, `u2`, `u3`, and so on
Outputs	`y1`, `y2`, `y3`, and so on
Parameters	`p1`, `p2`, `p3`, and so on
States	`x1`, `x2`, `x3`, and so on

Input ports are named to reflect the signal names that they propagate.

Standard Methods to Prepare a Model for Code Generation

In this section...

"Configuring a Signal" on page 6-3

"Configuring Input and Output Ports" on page 6-4

"Initializing States" on page 6-4

"Setting Up Configuration Parameters for Code Generation" on page 6-4

"Setting Up an Example Model With a Stateflow Chart" on page 6-5

"Setting Up an Example Model With a MATLAB Function Block" on page 6-7

Configuring a Signal

1. Create a model in Simulink. See "Creating a Model" in the Simulink documentation.

2. Right-click a signal line. Select **Signal Properties**. A Signal Properties dialog box opens. See "Signal Properties Dialog Box" for more information.

3. Enter a signal name for the **Signal name** parameter.

4. On the same Signal Properties dialog box, select the **Code Generation** tab. Use the drop down menu for the **Storage class** parameter to specify a storage class. Examples in this chapter use `Exported Global`.

Note Alternatively, on the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Then create a signal data object in the base workspace with the same name as the signal. See "Creating Simulink and mpt Data Objects" on page 12-4 for more information on creating data objects in the base workspace. (Examples use `mpt.Signal` and specify the **Storage class** as `ExportedGlobal`.

Configuring Input and Output Ports

1. In your model,

   Double-click an `Inport` or `Outport` block. A Block Parameters dialog box opens.

2. Select the **Signal Attributes** tab.

3. Specify the **Port dimensions** and **Data type**. Examples leave the default value for **Port dimensions** as `1` (for inherited) and **Data type** as `Inherit: auto`.

Initializing States

1. Double-click a block.

2. In the Block Parameters dialog box, select the **Main** tab.

3. Specify the **Initial conditions** and **Sample time**. See "Working with Sample Times".

4. Select the **State Attributes** pane. Specify the state name. See "Block State Storage and Interfacing Considerations".

5. You can also use the Data Object Wizard for creating data objects. A part of this process initializes states. See "Creating Simulink Data Objects with Data Object Wizard" on page 12-5.

Setting Up Configuration Parameters for Code Generation

1. Open the Configuration Parameter dialog box by selecting **Simulation > Configuration parameters**. You can also use the keyboard shortcut `Ctrl+E`.

2. Open the **Solver** pane and select

   - **Solver type**: `Fixed-Step`
   - **Solver**: `discrete (no continuous states)`

3. Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.

4. Open the **Code Generation** pane, and specify `ert.tlc` as the **System Target File**.

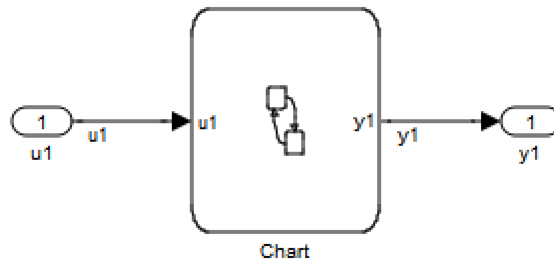
5. Clear **Generate makefile**.

6. Select **Generate code only**.

7. Enable the HTML report generation by opening the **Code Generation > Report** pane and selecting **Create code generation report**, **Launch report automatically**, and **Code-to-model**. Enabling the HTML report generation is optional.

8. Click **Apply** and then **OK** to exit.

Setting Up an Example Model With a Stateflow Chart



Follow this general procedure to create a simple model containing a Stateflow chart.

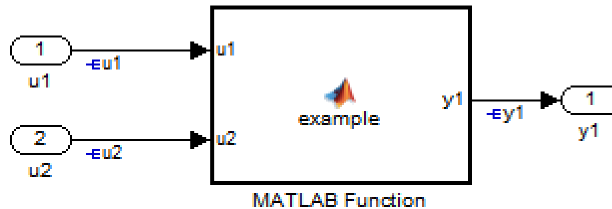
1. From the **Stateflow > Stateflow Chart** library, add a Stateflow chart to your model .

2. Add Inport blocks and Outport blocks according to the example model.

3. Open the **Stateflow Editor** by performing one of the following:
   - Double-click the Stateflow chart.
   - Select **Tools > Explore**.
   - Press **Ctrl+R**.
4. Select **Add > Data > Input from Simulink** to add the inputs to the chart. A Data dialog box opens for each input.
5. Specify the **Name** (`u1, u2, ...`) and the **Type** (`Inherit: Same as Simulink`) for each input, unless specified differently in the example. Click **OK**.

   Click **Apply** and close each dialog box.
6. Select **Add > Data > Output from Simulink** to add the outputs to the chart. A Data dialog opens for each output.
7. Specify the **Name** (`y1, y2, ...`) and **Type** (`Inherit: Same as Simulink`) for each output, unless specified differently in the example. Click **OK**.
8. Click **Apply** and close each dialog box.
9. In the **Stateflow Editor**, create the Stateflow diagram specific to the example.
10. The inputs and outputs appear on the chart in your model.
11. Connect the Inport and Outport blocks to the Stateflow Chart.
12. Configure the input and output signals; see "Configuring a Signal" on page 6-3.

Setting Up an Example Model With a MATLAB Function Block



1. Add the number of Inport and Outport blocks according to a C construct example included in this chapter.

2. From the Simulink User-defined Functions library drag a MATLAB Function block into the model.

3. Double-click the block. The MATLAB Function Block Editor opens. Edit the function to implement your application.

4. Click **File > Save** and close the MATLAB Function Block Editor.

5. Connect the Inport and Outport blocks to the MATLAB Function block. See “Configuring a Signal” on page 6-3.

6. Save your model.

Types, Operators, and Expressions

In this section...

"Data Declaration" on page 6-8
"Data Type Conversion" on page 6-11
"Type Qualifiers" on page 6-15
"Relational and Logical Operators" on page 6-17
"Bitwise Operations" on page 6-21

Data Declaration

C Construct

```
int32 p1 = 3;
```

Declare a Variable for a Block Parameter Using a Data Object

You can specify certain block parameters as a variable. If you define the variable as a data object, the variable is global. Where the variable is declared in the generated code depends on the custom storage class that you choose (and whether you select **Inline Parameters** on the **Optimization > Signals and Parameters** pane). If you choose **Inline Parameters**, then the data object name is used in the generated code. If you did not choose **Inline Parameters**, the generated code creates a global structure that stores all of the parameters. For more information on how to create a data object, see Defining Data Representation and Storage for Code Generation on page 1.

Block	Parameter
Constant	Value
Gain	Value
For Iterator	Iteration Limit

There are several methods for configuring data objects:

- For a model with many parameters, use the Data Object Wizard, which analyzes your model and finds all of the unresolved data objects and data types. You can then create the data objects in the Data Object Wizard. The procedure for using the Data Object Wizard for a parameter is similar to the procedure for a signal. For an example, see "Declare a Variable for a Signal using a Data Object" on page 6-10.
- To add, delete, edit, and configure data objects, use the base workspace in the Model Explorer.
- To create and configure data objects, use the MATLAB command line.

The following example demonstrates how to create a data object using the Model Explorer. The declaration statement in the generated code is as follows:

```
int Kp = 3;
```

1. Create a model containing a Constant block and a Gain block.
2. Press **Ctrl+E** to open the Configuration Parameters dialog box.
3. On the **Optimization > Signals and Parameters** pane of the Configuration Parameter dialog box, select **Inline parameters**.
4. Click **Apply** and **OK**. The Configuration Parameter dialog box closes.
5. In your model, double-click the Constant block. The Block Parameters dialog box opens.
6. In the **Value** field, enter a variable name. In this example, the variable name is `p1`.
7. In your model, double-click the Gain block. The Block Parameters dialog box opens.
8. In the **Value** field, enter a variable name. In this example, the variable name is `p2`.
9. Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.

10. To add two MPT parameter objects, in the menu bar, select **Add > MPT Parameter** in the menu bar twice. On the **Contents of: Base Workspace** pane, you see the parameters.

11. Double-click each `mpt.Parameter` object and change their names to `p1` and `p2`.

12. Click the `p1` parameter. The data object parameters are displayed in the right pane of the Model Explorer.

13. In the **Value** field, enter `3` for `p1`. For the **Data type**, select `int32`. Because you chose an MPT parameter, the **Storage Class** is already set to `Global(Custom)`.

14. In the **Value** field, enter `5` for `p2`. For the **Data type**, select `int32`.

15. Press **Ctrl+B** to generate code.

In the *model*.c file you see:

```
int32 p1 = 3;
int32 p2 = 5;
```

Note Depending on the storage class, the global variable is represented differently in the generated code. For more information, see "Parameter Objects".

C Construct

```
int p1 = 3;
```

Declare a Variable for a Signal using a Data Object

1. Create a model and label the signals.

2. In the model tool bar, click **Tools > Data Object Wizard** to open the **Data Object Wizard**. If you are not familiar with creating Simulink Data Objects using the wizard, refer to "Data Object Wizard" .

3. Click **Find**. The list of unresolved parameters and objects populates the Data Object Wizard. You can do mass edits for identical data objects.

4. Select the signals individually or select all signals by clicking **Check All**.

5. From the parameter **Choose package for selected data objects** drop-down list, select the `mpt package`. Click **Apply Package**. When you open the Model Explorer the data objects appear in the base workspace.

6. In the base workspace, click the `p1` data object . The data object parameters appear in the right pane of the Model Explorer.

7. From the **Data type** drop-down list, select `int16`.

8. You can also specify the storage class. The data object is an `mpt.Parameter` object, therefore the Storage Class is automatically set to `Global (Custom)`.

Note The Storage class alters the data object implementation in the generated code. For more information, see “Signal Objects”.

Data Type Conversion

C Construct

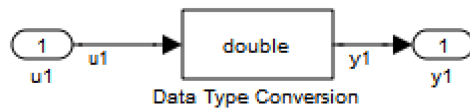
```
y1 = (double)u1;
```

Modeling Patterns

- “Modeling Pattern for Data Type Conversion — Simulink Block” on page 6-12
- “Modeling Pattern for Data Type Conversion — Stateflow Chart” on page 6-13
- “Modeling Pattern for Data Type Conversion — MATLAB Function Block” on page 6-14

Modeling Pattern for Data Type Conversion — Simulink Block

One method to create a data type conversion is to use a Data Type Conversion block from the **Simulink > Commonly Used Blocks** library.



ex_data_type_SL

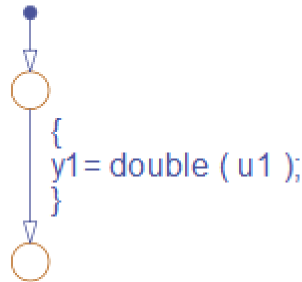
1. From the **Commonly Used Blocks** library, drag a Data Type Conversion block into your model and connect to the Inport and Outport blocks.
2. Double-click on the Data Type Conversion block to open the Block Parameters dialog box.
3. Select the **Output data type** parameter as `double`.
4. Press **Ctrl+B** to build the model and generate code.

The generated code appears in `ex_data_type_SL.c`, as follows:

```
int32_T u1;
real_T y1;

void ex_data_type_SL_step(void)
{
 y1 = (real_T)u1;
}
```

The Embedded Coder type definition for `double` is `real_T`.

Modeling Pattern for Data Type Conversion — Stateflow Chart



Stateflow Chart Type Conversion

Procedure.

1. Follow the steps for "Setting Up an Example Model With a Stateflow Chart" on page 6-5 . This example contains one Inport block and one Outport block.

2. Name the example model `ex_data_type_SF`.

3. Double-click the Inport block and select the **Signal Attributes** tab. Specify the **Data Type** as `int32` from the drop down menu.

4. Double-click the Outport block and select the **Signal Attributes** tab. Specify the **Data Type** as `Inherit: auto` from the drop down menu.

5. In the **Stateflow Editor**, specify the **Data Type** for `y1` as `Boolean`

6. Press **Ctrl+B** to build the model and generate code.

Results. The generated code appears in `ex_data_type_SF.c`, as follows:

```
int32_T u1;
real_T y1;
void ex_data_type_SF_step(void)
{
  y1 = (real_T)u1;
}
```

Modeling Pattern for Data Type Conversion — MATLAB Function Block

Procedure.

1 Follow the steps for "Setting Up an Example Model With a MATLAB Function Block" on page 6-7 . This example model contains one Inport block and one Outport block.

2 Name the model `ex_data_type_ML_Func`.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = typeconv(u1)
y1 = double(u1);
end
```

4 Press **Ctrl+B** to build the model and generate code.

Results. The generated code appears in `ex_data_type_ML_func.c`, where `real32_T` is a `float` and `real_T` is a `double`. Type conversion occurs across assignments.

```
real32_T u1;
real_T y1;

void ex_data_type_ML_func_step(void)
{
     y1 = u1;
}
```

Other Type Conversions in Modeling

Type conversions can also occur on the output of blocks where the output variable is specified as a different data type. For example, in the Gain block, you can select the **Inherit via internal rule** parameter to control the output signal data type. Another example of type conversion can occur at the boundary of a Stateflow chart. You can specify the output variable as a different data type.

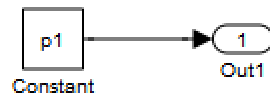
Type Qualifiers

Modeling Patterns for Type Qualifiers

- "Using a Tunable Parameter in the Base Workspace" on page 6-15
- "Using a Data Object of the `Const` Custom Storage Class" on page 6-16

Using a Tunable Parameter in the Base Workspace

A tunable parameter is a block parameter whose value can be changed at runtime. The storage class property of a parameter specifies how the parameter is declared in the generated code.



ex_type_qual

Procedure.

1 Create a model containing a Constant block and an Outport block.

2 Double-click the Constant block. In the **Constant value** field, enter the parameter name `p1` .

3 In the base workspace, create a MATLAB variable for `p1` and specify its **Value** as `9.8` and its **Data type** as `double`.

4 Press **Ctrl+E** to open the Configuration Parameters dialog box and select the **Optimization > Signals and Parameters** pane.

5 Select the **Inline parameters** parameter, which activates the **Configure** button.

6 Click the **Configure** button to open the Model Parameter Configuration dialog box.

7 To declare a tunable parameter, from the **Source list**, select the variable `p1`.

8 Click the **Add to table** button to add `p1` to the **Global (tunable) parameters** section.

9 Click the **Storage Class** and select `Exported Global`.

10 Click the **Storage Type Qualifier** arrow and select `const`.

11 Click **Apply** to save all of the changes.

12 Press **Ctrl+B** to build the model and generate code.

Results. The generated code appears in `ex_type_qual.c` as follows:

```
/* Exported block parameters */
   const real_T p1 = 9.8;          /* Variable: p1
                                    * Referenced by: '<Root>/Constant'
                                    */
```

Using a Data Object of the `Const` Custom Storage Class

One way to create a type qualifier in the generated code is to create a data object and specify the appropriate custom storage class. Use the previous model, `ex_type_qual`, for this example. Specify `p1` differently as follows:

Procedure.

1 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the base workspace.

2 In the menu bar, select **Add > MPT Parameter** to add an MPT parameter object. The parameter is displayed in the **Contents of: Base Workspace** pane.

3 Double-click the `mpt.Parameter` object and change the **Name** to `p1`.

4 Click the `p1` parameter which displays the data object parameters on the right pane of the Model Explorer.

5 In the **Value** field, enter `9.8` for `p1`. Specify the **Data type** as `auto` for 64–bit double.

6 You can use the different type qualifiers by selecting a custom storage class from the **Storage class** list. For this example, select `ConstVolatile (custom)`.

7 In the Configuration Parameters dialog box, on the **Optimization > Signals and Parameters** pane, select the **Inline parameters**.

8 Press **Ctrl+B** to build the model and generate code.

Results. The generated code produces the type qualifier in `ex_type_qual.c`:

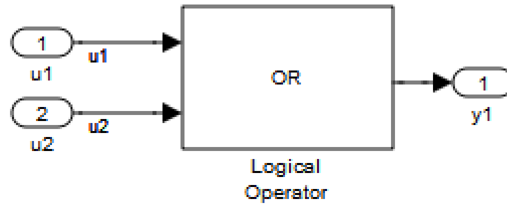
```
const volatile real_T p1 = 9.8;
```

Relational and Logical Operators

Modeling Patterns for Relational and Logical Operators

- "Modeling Pattern for Relational or Logical Operators — Simulink Blocks" on page 6-18
- "Modeling Pattern for Relational and Logical Operators —Stateflow Chart" on page 6-19
- "Modeling Pattern for Relational and Logical Operators — MATLAB Function Block" on page 6-20

Modeling Pattern for Relational or Logical Operators — Simulink Blocks



ex_logical_SL

Procedure.

1. From the **Logic and Bit Operations** library, drag a Logical Operator block into your model.

2. Double-click the block to configure the logical operation. Set the **Operator** field to `OR`.

3. Name the blocks, as shown in the model `ex_logical_SL`.

4. Connect the blocks and name the signals, as shown in the model `ex_logical_SL`.

5. Press **Ctrl+B** to build the model and generate code.

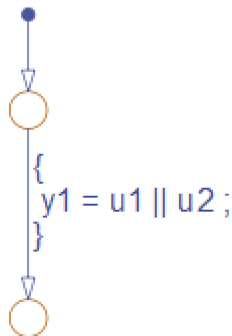
Note You can use the above procedure to implement relational operators by replacing the Logical Operator block with a Relational Operator block.

Results. Code implementing the logical operator `OR` is in the `ex_logical_SL_step` function in `ex_logical_SL.c`:

```
/* Exported block signals */
   boolean_T u1;                          /* '<Root>/u1' */
   boolean_T u2;                          /* '<Root>/u2' */
```

```
boolean_T y1;                         /* '<Root>/Logical Operator'*/

/* Logic: '<Root>/Logical Operator' incorporates:
 *  Inport: '<Root>/u1'
 *  Inport: '<Root>/u2'
 */
 y1 = (u1 || u2);
```

Modeling Pattern for Relational and Logical Operators —Stateflow Chart



ex_logical_SF/Logical Operator Stateflow® Chart

Procedure.

1. Follow the steps for “Setting Up an Example Model With a Stateflow Chart” on page 6-5. This example model contains two Inport blocks and one Outport block.

2. Name the example model `ex_logical_SF`.

3. In the **Stateflow Editor**, specify the **Data Type** for `y1` as `Boolean`.

4. In the **Stateflow Editor**, create the Stateflow diagram as shown. The relational or logical operation actions are on the transition from one junction

to another. Relational statements specify conditions to conditionally allow a transition. In that case, the statement would be within square brackets.

5 Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the logical operator OR is in the `ex_logical_SF_step` function in `ex_logical_SF.c`:

```
boolean_T u1;                 /* '<Root>/u1' */
boolean_T u2;                 /* '<Root>/u2' */
boolean_T y1;                 /* '<Root>/Chart' */

void ex_logical_SF_step(void)
{
    y1 = (u1 || u2);
}
```

Modeling Pattern for Relational and Logical Operators — MATLAB Function Block

This example demonstrates the MATLAB Function block method for incorporating operators into the generated code using a relational operator.

Procedure.

1 Follow the steps for “Setting Up an Example Model With a MATLAB Function Block” on page 6-7 . This example model contains two Inport blocks and one Outport block.

2 Name the example model `ex_rel_operator_ML`.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)
y1 = u1 > u2;
end
```

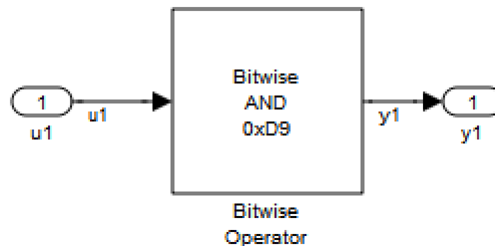
4 Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the relational operator '>' is in the `ex_rel_operator_ML_step` function in `ex_rel_operator_ML.c`:

```
real_T u1;                          /* '<Root>/u1' */
real_T u2;                          /* '<Root>/u2' */
boolean_T y;                        /* '<Root>/MATLAB Function' */

void ex_rel_operator_ML_step(void)
{
  y = (u1 > u2);
 }
```

Bitwise Operations

Simulink Bitwise-Operator Block



ex_bit_logic_SL

Procedure.

1. Drag a Bitwise Operator block from the **Logic and Bit Operations** library into your model.

2. Double-click the block to open the Block Parameters dialog.

3. Select the type of **Operator**. In this example, select `AND`.

4. In order to perform Bitwise operations with a bit-mask, select **Use bit mask**.

Note If another input uses Bitwise operations, clear the **Use bit mask** parameter and enter the number of input ports.

---

5. In the **Bit Mask** field, enter a decimal number. Use `bin2dec` or `hex2dec` to convert from binary or hexadecimal. In this example, enter `hex2dec('D9')`.

6. Name the blocks, as shown in, model `ex_bit_logic_SL`.

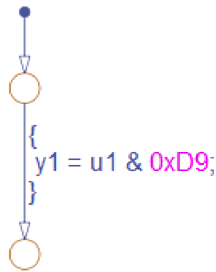
7. Connect the blocks and name the signals, as shown in, model `ex_bit_logic_SL`.

8. Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the logical operator OR is in the `ex_bit_logic_SL_step` function in `ex_bit_logic_SL.c`:

```
uint8_T u1;
uint8_T y1;

void ex_bit_logic_SL_step(void)
{
  y1 = (uint8_T)(u1 & 217);
}
```

Stateflow Chart



ex_bit_logic_SF/Bit_Logic Stateflow Chart

Procedure.

1 Follow the steps for "Setting Up an Example Model With a Stateflow Chart" on page 6-5. This example contains one Inport block and one Outport block.

2 Name the example model `ex_bit_logic_SF`.

3 From the **Stateflow Editor**, select**Tools > Explore** to open the Model Explorer.

4 In the Model Explorer, on the right pane, select **Enable C-bit operations**.

5 In the **Stateflow Editor**, create the Stateflow diagram, `ex_bit_logic_SF/Bit_Logic`.

6 Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the logical operator OR is in the `ex_bit_logic_SF_step` function in `ex_bit_logic_SF.c`:

```
uint8_T u1;
uint8_T y1;

void bit_logic_SF_step(void)
{
   y1 = (uint8_T)(u1 & 0xD9);
```

```
}
```

MATLAB Function Block

In this example, to demonstrate theMATLAB Function block method for implementing bitwise logic into the generated code, use the bitwise OR, '|'.

Procedure.

1 Follow the steps for "Setting Up an Example Model With a MATLAB Function Block" on page 6-7. This example model contains two Inport blocks and one Outport block.

2 Name your model ex_bit_logic_ML.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)

y1 = bitor(u1, u2);
end
```

4 Press **Ctrl+B** to build the model and generate code.

Results. Code implementing the bitwise operator OR is in the ex_bit_logic_ML_step function in ex_bit_logic_ML.c:

```
uint8_T u1;
uint8_T u2;
uint8_T y1;

void ex_bit_logic_ML_step(void)
{
 y1 = (uint8_T)(u1 | u2);
}
```

Control Flow

In this section...

"If-Else" on page 6-25

"Switch" on page 6-32

"For loop" on page 6-38

"While loop" on page 6-46

"Do While loop" on page 6-58

If-Else

C Construct

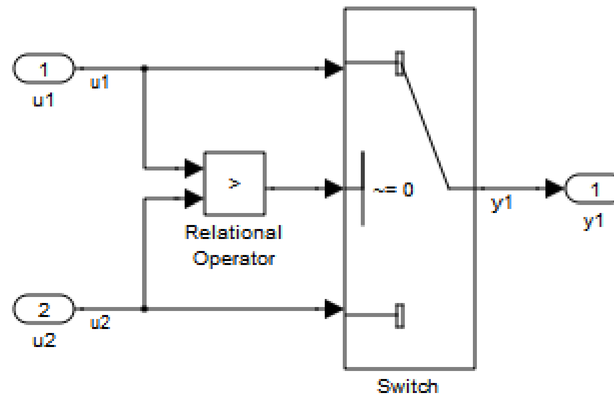
```
if (u1 > u2)
{
  y1 = u1;
}
else
{
  y1 = u2;
}
```

Modeling Patterns

- "Modeling Pattern for If-Else: Switch block" on page 6-26
- "Modeling Pattern for If-Else: Stateflow Chart" on page 6-28
- "Modeling Pattern for If-Else: MATLAB Function Block" on page 6-30

Modeling Pattern for If-Else: Switch block

One method to create an `if-else` statement is to use a Switch block from the **Simulink > Signal Routing** library.



Model ex_if_else_SL

Procedure.

1 Drag the Switch block from the **Simulink>Signal Routing** library into your model.

2 Connect the data inputs and outputs to the block.

3 Drag a Relational Operator block from the Logic & Bit Operations library into your model.

4 Connect the signals that are used in the if-expression to the Relational Operator block. The order of connection determines the placement of each signal in the if-expression.

5 Configure the Relational Operator block to be a greater than operator.

6 Connect the controlling input to the middle input port of the Switch block.

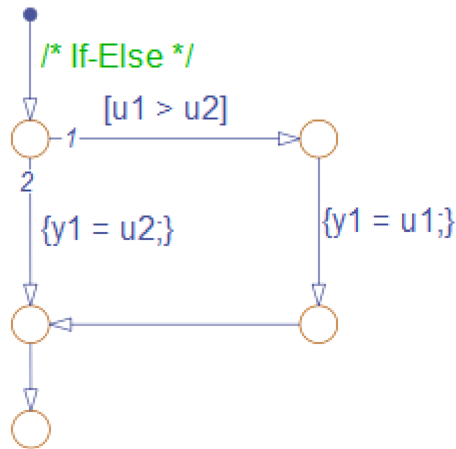
7 Double-click the Switch block and set **Criteria for passing first input** to `u2~=0`. This condition ensures that Simulink selects `u1` if `u2` is `TRUE`; otherwise `u2` passes.

8 Enter `Ctrl+B` to build the model and generate code.

Results. The generated code includes the following `ex_if_else_SL_step` function in the file `ex_if_else_SL.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_if_else_SL_step(void)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Outport: '<Root>/y1'
   *  RelationalOperator: '<Root>/Relational Operator'
   */
 if (U.u1 > U.u2) {
    Y.y1 = U.u1;
  } else {
    Y.y1 = U.u2;
  }
}
```

Modeling Pattern for If-Else: Stateflow Chart



ex_if_else_SF/Chart

Procedure.

1 Follow the steps for "Setting Up an Example Model With a Stateflow Chart" on page 6-5. This example model contains two Inport blocks and one Outport block.

2 Name your model `ex_if_else_SF`.

3 When configuring your Stateflow chart, select **Patterns > Add Decision > If-Else**. The Stateflow Pattern dialog opens. Fill in the fields as follows:

Description	`If-Else` (optional)
If condition	`u1 > u2`
If action	`y1 = u1`
Else action	`y1 = u2`

4 Press **Ctrl+B** to build the model and generate code.

Results. The generated code includes the following `ex_if_else_SF_step` function in the file `If_Else_SF.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_if_else_SF_step(void)
{
  /* Stateflow: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Outport: '<Root>/y1'
   */
  /* Gateway: Chart */
  /* During: Chart */
  /* Transition: '<S1>:14' */
  /*  If-Else  */
  if (U.u1 > U.u2) {
    /* Transition: '<S1>:13' */
    /* Transition: '<S1>:12' */
    Y.y1 = U.u1;

    /* Transition: '<S1>:11' */
  } else {
    /* Transition: '<S1>:10' */
    Y.y1 = U.u2;
  }

  /* Transition: '<S1>:9' */
}
```

Modeling Pattern for If-Else: MATLAB Function Block

Procedure.

1 Follow the steps for “Setting Up an Example Model With a MATLAB Function Block” on page 6-7. This example model contains two Inport blocks and one Outport block.

2 Name your model `ex_if_else_ML`.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2)
if u1 > u2;
  y1 = u1;
else y1 = u2;
end
```

4 Press **Ctrl+B** to build the model and generate code.

Results. The generated code includes the following `ex_if_else_ML_step` function in the file `ex_if_else_ML.c`:

```
/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_if_else_ML_step(void)
{
  /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   *  Inport: '<Root>/u2'
   *  Outport: '<Root>/y1'
   */
  /* MATLAB Function 'MATLAB Function': '<S1>:1' */
  if (U.u1 > U.u2) {
    /* '<S1>:1:4' */
    /* '<S1>:1:5' */
```

```
    Y.y1 = U.u1;
  } else {
    /* '<S1>:1:6' */
    Y.y1 = U.u2;
  }
}
```

Switch

C Construct

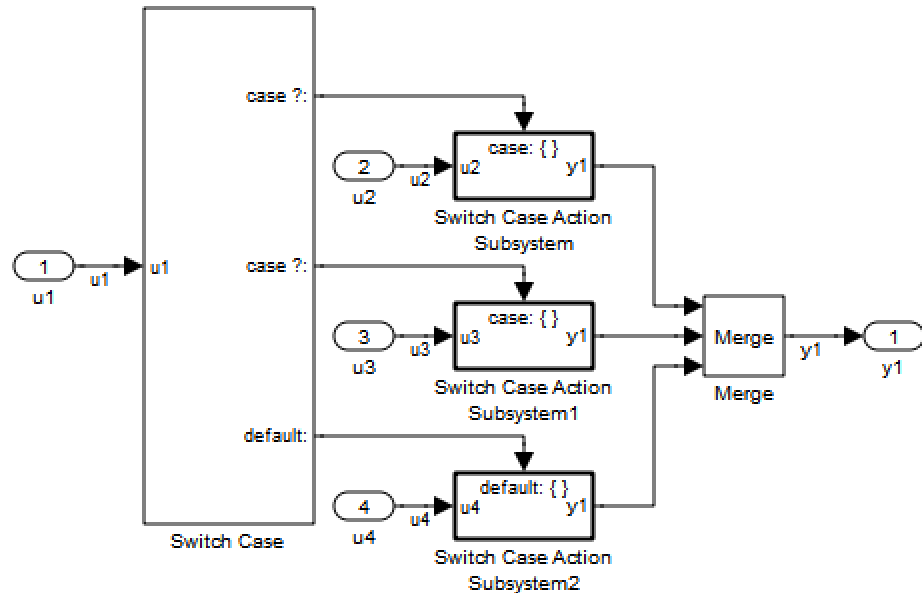
```
switch (u1)
{
 case 2:
    y1 = u2;
    break;
 case 3:
    y1 = u3;
    break;
 default:
    y1 = u4;
    break;
}
```

Modeling Patterns

- "Modeling Pattern for Switch: Switch Case block" on page 6-33
- "Modeling Pattern for Switch: MATLAB Function block" on page 6-36
- "Converting If-Elseif-Else to Switch statement" on page 6-37

Modeling Pattern for Switch: Switch Case block

One method for creating a `switch` statement is to use a Switch Case block from the **Simulink > Ports and Subsystems** library.



Model ex_switch_SL

Procedure.

1 Drag a Switch Case block from the **Simulink > Ports and Subsystems** library into your model.

2 Double-click the block. In the Block Parameters dialog box, fill in the **Case Conditions** parameter. In this example, the two cases are: `{2,3}`.

3 Select the **Show default case** parameter. The default case is optional in a `switch` statement.

4 Connect the condition input `u1` to the input port of the Switch block.

5. Drag Switch Case Action Subsystem blocks from the **Simulink>Ports and Subsystems** library to correspond with the number of cases.

6. Configure the Switch Case Action Subsystem subsystems.

7. Drag a Merge block from the **Simulink > Signal Routing** library to merge the outputs.

8. The Switch Case block takes an integer input, therefore, the input signal `u1` is type cast to an `int32`.

9. Enter `Ctrl+B` to build the model and generate code.

Results. The generated code includes the following `ex_switch_SL_step` function in the file `ex_switch_SL.c`:

```
/* Exported block signals */
int32_T u1;                            /* '<Root>/u1' */

/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_switch_SL_step(void)
{
  /* SwitchCase: '<Root>/Switch Case' incorporates:
   *  ActionPort: '<S1>/Action Port'
   *  ActionPort: '<S2>/Action Port'
   *  ActionPort: '<S3>/Action Port'
   *  Inport: '<Root>/u1'
   *  SubSystem: '<Root>/Switch Case Action Subsystem'
   *  SubSystem: '<Root>/Switch Case Action Subsystem1'
   *  SubSystem: '<Root>/Switch Case Action Subsystem2'
   */
  switch (u1) {
   case 2:
    /* Inport: '<S1>/u2' incorporates:
     *  Inport: '<Root>/u2'
```

```
     *  Outport: '<Root>/y1'
     */
    Y.y1 = U.u2;
    break;

   case 3:
    /* Inport: '<S2>/u3' incorporates:
     *  Inport: '<Root>/u3'
     *  Outport: '<Root>/y1'
     */
    Y.y1 = U.u3;
    break;

   default:
    /* Inport: '<S3>/u4' incorporates:
     *  Inport: '<Root>/u4'
     *  Outport: '<Root>/y1'
     */
    Y.y1 = U.u4;
    break;
  }
}
```

Modeling Pattern for Switch: MATLAB Function block

Procedure.

1 Follow the steps for "Setting Up an Example Model With a MATLAB Function Block" on page 6-7. This example model contains four Inport blocks and one Outport block.

2 Name your model `ex_switch_ML`.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1, u2, u3, u4)

switch u1
    case 2
        y1 = u2;
    case 3
        y1 = u3;
    otherwise
        y1 = u4;
end
```

4 Press **Ctrl+B** to build the model and generate code.

Results. The generated code includes the following `ex_switch_ML_step` function in the file `ex_switch_ML.c`:

```
/* External inputs (root inport signals with auto storage) */
   ExternalInputs U;

   /* External outputs (root outports fed by signals with auto storage) */
   ExternalOutputs Y;

   /* Model step function */
   void ex_switch_ML_step(void)
   {
     /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
      *  Inport: '<Root>/u1'
      *  Inport: '<Root>/u2'
      *  Inport: '<Root>/u3'
```

```
 *  Inport: '<Root>/u4'
 *  Outport: '<Root>/y1'
 */
/* MATLAB Function 'MATLAB Function': '<S1>:1' */
/* '<S1>:1:4' */
switch (U.u1) {
 case 2:
  /* '<S1>:1:6' */
  Y.y1 = U.u2;
  break;

 case 3:
  /* '<S1>:1:8' */
  Y.y1 = U.u3;
  break;

 default:
  /* '<S1>:1:10' */
  Y.y1 = U.u4;
  break;
}
}
```

Converting If-Elseif-Else to Switch statement

If a MATLAB Function block or a Stateflow chart uses `if-elseif-else` decision logic, you can convert it to a `switch` statement by using a configuration parameter. In the Configuration Parameters dialog box, on the **Code Generation > Code Style** pane, select the "Convert if-elseif-else patterns to switch-case statements" parameter. For more information, see "Converting If-Elseif-Else Code to Switch-Case Statements" in the Simulink documentation. For more information on this conversion using a Stateflow chart, see "Converting If-Elseif-Else Code to Switch-Case Statements" and "Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements" in the Stateflow documentation.

For loop

C Construct

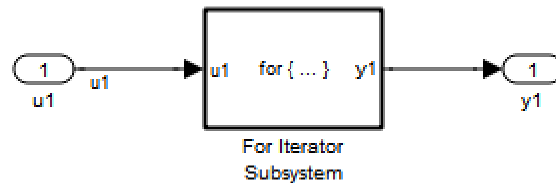
```
y1 = 0;
for(inx = 0; inx <10; inx++)
{
  y1 = u1[inx] + y1;
}
```

Modeling Patterns:

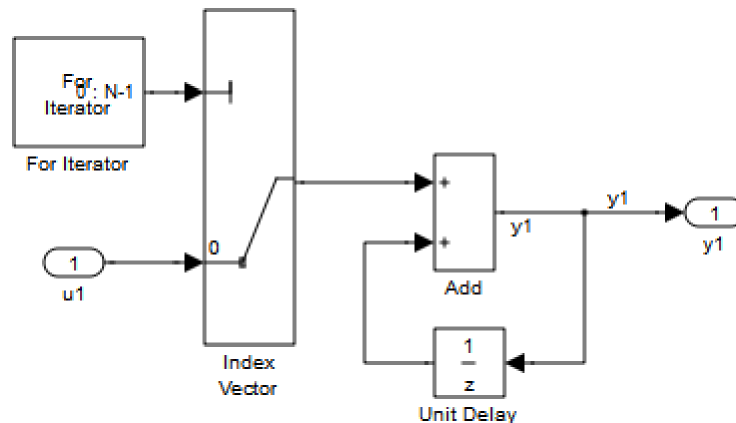
- “Modeling Pattern for For Loop: For-Iterator Subsystem block” on page 6-39
- “Modeling Pattern for For Loop: Stateflow Chart” on page 6-42
- “Modeling Pattern for For Loop: MATLAB Function block” on page 6-45

Modeling Pattern for For Loop: For-Iterator Subsystem block

One method for creating a `for` loop is to use a For Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



Model ex_for_loop_SL



For Iterator Subsystem

Procedure.

1 Drag a For Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into your model.

2 Connect the data inputs and outputs to the For Iterator Subsystem block.

3 Open the Inport block.

4 In the Block Parameters dialog box, select the **Signal Attributes** pane and set the **Port dimensions** parameter to 10.

5 Double-click the For Iterator Subsystem block to open the subsystem.

6 Drag an Index Vector block from the Signal-Routing library into the subsystem.

7 Open the For Iterator block. In the Block Parameters dialog box set the **Index-mode** parameter to `Zero-based` and the **Iteration limit** parameter to 10.

8 Connect the controlling input to the topmost input port of the Index Vector block, and the other input to the second port.

9 Drag an Add block from the **Math Operations** library into the subsystem.

10 Drag a Unit Delay block from **Commonly Used Blocks** library into the subsystem.

11 Double-click the Unit Delay block and set the **Initial Conditions** parameter to `0`. This parameter initializes the state to zero.

12 Connect the blocks as shown in the model diagram.

13 Save the subsystem and the model.

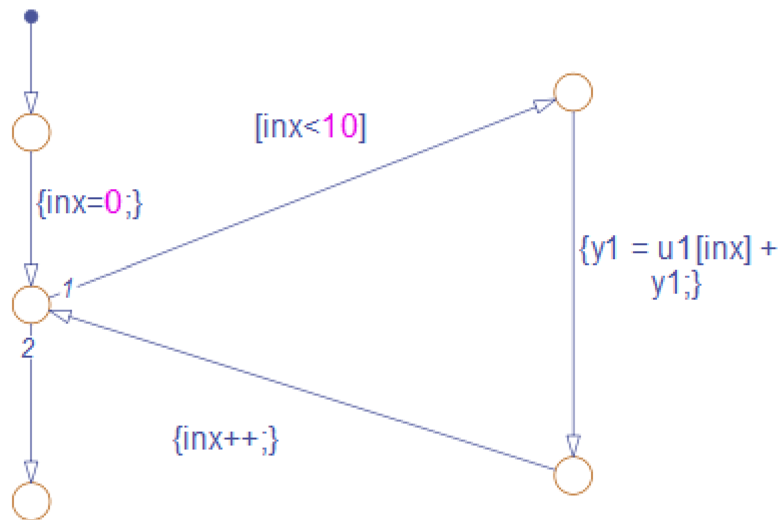
14 Enter `Ctrl+B` to build the model and generate code.

Results. The generated code includes the following `ex_for_loop_SL_step` function in the file `ex_for_loop_SL.c`:

```
  /* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_for_loop_SL_step(void)
```

```
{
  int32_T s1_iter;
  int32_T rtb_y1;

  /* Outputs for iterator SubSystem: '<Root>/For Iterator Subsystem' incorporates:
   *  ForIterator: '<S1>/For Iterator'
   */
  for (s1_iter = 0; s1_iter < 10; s1_iter++) {
    /* Sum: '<S1>/Add' incorporates:
     *  Inport: '<Root>/u1'
     *  MultiPortSwitch: '<S1>/Index Vector'
     *  UnitDelay: '<S1>/Unit Delay'
     */
    rtb_y1 = U.u1[s1_iter] + DWork.UnitDelay_DSTATE;

    /* Update for UnitDelay: '<S1>/Unit Delay' */
    DWork.UnitDelay_DSTATE = rtb_y1;
  }

  /* end of Outputs for SubSystem: '<Root>/For Iterator Subsystem' */

  /* Outport: '<Root>/y1' */
  Y.y1 = rtb_y1;
}
```

Modeling Pattern for For Loop: Stateflow Chart



Procedure.

1 Follow the steps for “Setting Up an Example Model With a Stateflow Chart” on page 6-5. This example model contains one Inport block and one Outport block.

2 Name the model `ex_for_loop_SF`.

3 Enter `Ctrl+R` to open the Model Explorer.

4 In the Model Explorer, select the output variable, `u1`, and in the right pane, select the **General** tab and set the **Initial Value** to `0`.

5 In the **Stateflow Editor**, select **Patterns > Add Loop > For**. The Stateflow Pattern dialog opens.

6 Fill in the fields in the Stateflow Pattern dialog box as follows:

Description	`For Loop` (optional)
Initializer expression	`inx = 0`
Loop test expression	`inx < 10`
Counting expression	`inx++`
For loop body	`y1 = u1[inx] + y1`

The Stateflow diagram is shown.

7 Press **Ctrl+B** to build the model and generate code.

Results. The generated code includes the following `ex_for_loop_SF_step` function in the file `ex_for_loop_SF.c`:

```
/* Block signals (auto storage) */
BlockIO B;

/* External inputs (root inport signals with auto storage) */
ExternalInputs U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs Y;

/* Model step function */
void ex_for_loop_SF_step(void)
{
  int32_T sf_inx;

  /* Stateflow: '<Root>/Chart' incorporates:
   *  Inport: '<Root>/u1'
   */
  /* Gateway: Chart */
  /* During: Chart */
  /* Transition: '<S1>:24' */
  /*  For Loop  */
  /* Transition: '<S1>:25' */
  for (sf_inx = 0; sf_inx < 10; sf_inx++) {
    /* Transition: '<S1>:22' */
    /* Transition: '<S1>:23' */
    B.y1 = U.u1[sf_inx] + B.y1;
```

```
      /* Transition: '<S1>:21' */
    }

    /* Transition: '<S1>:20' */

    /* Outport: '<Root>/y1' */
    Y.y1 = B.y1;
  }
```

Modeling Pattern for For Loop: MATLAB Function block

Procedure.

1. Follow the directions for “Setting Up an Example Model With a MATLAB Function Block” on page 6-7. This example model contains one Inport block and one Outport block.

2. Name your model `ex_for_loop_ML`.

3. In the MATLAB Function Block Editor enter the function, as follows:

```
function y1 = fcn(u1)

y1 = 0;

for inx=1:10
    y1 = u1(inx) + y1 ;
end
```

4. Press **Ctrl+B** to build the model and generate code.

Results. The generated code includes the following `ex_for_loop_ML_step` function in the file `ex_for_loop_ML.c`:

```
/* Exported block signals */
real_T u1[10];                        /* '<Root>/u1' */
real_T y1;                            /* '<Root>/MATLAB Function' */

/* Model step function */
void ex_for_loop_ML_step(void)
{
  int32_T inx;

  /* MATLAB Function Block: '<Root>/MATLAB Function' incorporates:
   *  Inport: '<Root>/u1'
   */
  /* MATLAB Function 'MATLAB Function': '<S1>:1' */
  /* '<S1>:1:3' */
  y1 = 0.0;
  for (inx = 0; inx < 10; inx++) {
```

```
        /* '<S1>:1:5' */
        /* '<S1>:1:6' */
        y1 = u1[inx] + y1;
      }
    }
```

While loop

C Construct

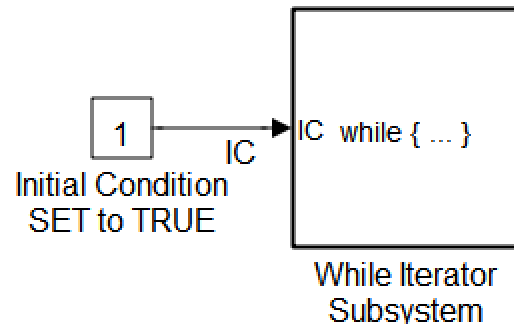
```
while(flag && (num_iter <= 100)
{
  flag = func ();
  num_iter ++;
}
```

Modeling Patterns

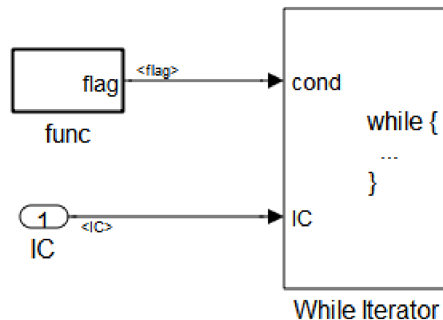
- “Modeling Pattern for While Loop: While Iterator Subsystem block” on page 6-47
- “Modeling Pattern for While Loop: Stateflow Chart” on page 6-51
- “Modeling Pattern for While Loop: MATLAB Function Block” on page 6-55

Modeling Pattern for While Loop: While Iterator Subsystem block

One method for creating a `while` loop is to use a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



Model ex_while_loop_SL



ex_while_loop_SL/While Iterator Subsystem

Procedure.

1 Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.

2 Drag a Constant block from the **Simulink > Commonly Used Blocks** library into the model. In this case, set the **Initial Condition** to `1` and the **Data Type** to `Boolean`. You do not always have to set the initial condition to `FALSE`. The initial condition can be dependent on the input to the block.

3 Connect the Constant block to the While Iterator Subystem block.

4 Double-click the While Iterator Subsystem block to open the subsystem.

5 Place a Subsystem block next to the While Iterator block.

6 Right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.

7 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

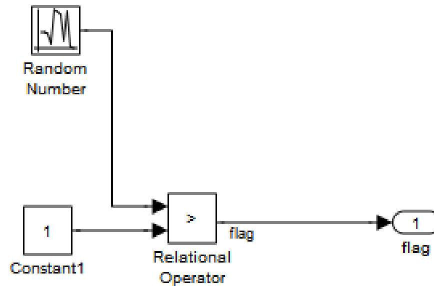
8 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Function`.

9 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

10 Specify the name as `func`.

11 Click **Apply**.

12 Double-click the `func` subsystem block. In this example, function `func()` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func( ). Create the func() algorithm as shown in the following diagram:



func

13 Double-click the While Iterator block to set the **Maximum number of iterations** to 100.

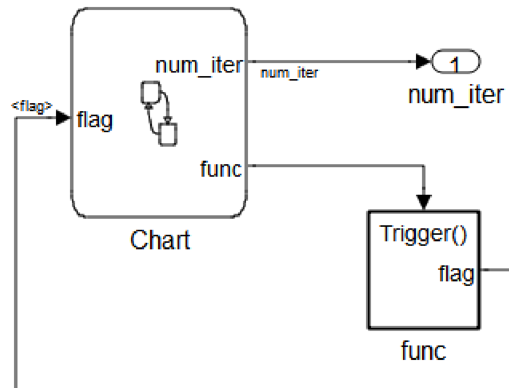
14 Connect blocks as shown in the model and subsystem diagrams.

Results. The generated code includes the following `ex_while_loop_SL_step` function in the file `ex_while_loop_SL.c`:

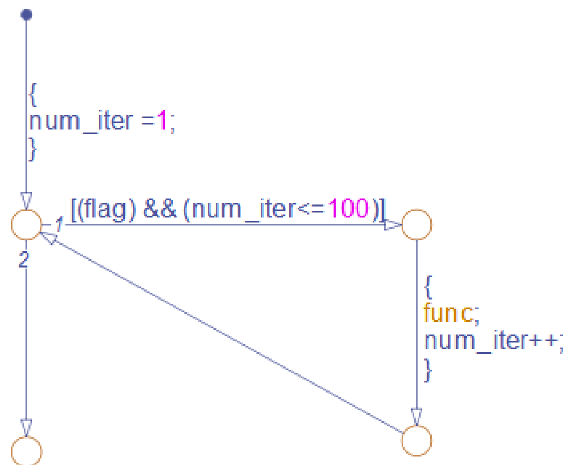
```
/* Exported block signals */
boolean_T IC;                           /* '<Root>/Initial Condition SET to TRUE' */
boolean_T flag;                         /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Start for atomic system: '<S1>/func( ) Is a function that updates the flag' */
void func_Start(void)
{
  /* Start for RandomNumber: '<S2>/Random Number' */
  DWork.RandSeed = 1144108930U;
  DWork.NextOutput = rt_NormalRand(&DWork.RandSeed) * 1.7320508075688772E+000;
}

/* Output and update for atomic system:
 *  '<S1>/func( ) Is a function that updates the flag' */
void func(void)
{
```

```
  /* RelationalOperator: '<S2>/Relational Operator' incorporates:
   *  Constant: '<S2>/Constant1'
   *  RandomNumber: '<S2>/Random Number'
   */
  flag = (DWork.NextOutput > 1.0);

  /* Update for RandomNumber: '<S2>/Random Number' */
  DWork.NextOutput = rt_NormalRand(&DWork.RandSeed) * 1.7320508075688772E+000;
}

/* Model step function */
void ex_while_loop_SL_step(void)
{
  int32_T s1_iter;
  boolean_T loopCond;

  /* Outputs for iterator SubSystem:
   *     '<Root>/While Iterator Subsystem' incorporates:
   *  WhileIterator: '<S1>/While Iterator'
   */
  s1_iter = 1;
  loopCond = IC;
  while (loopCond && (s1_iter <= 100)) {
    /* Outputs for atomic SubSystem:
     * '<S1>/func( ) Is a function that updates the flag' */
    func();

    /* end of Outputs for SubSystem:
     * '<S1>/func( ) Is a function that updates the flag' */
    loopCond = flag;
    s1_iter++;
  }

  /* end of Outputs for SubSystem: '<Root>/While Iterator Subsystem' */
}
```

Modeling Pattern for While Loop: Stateflow Chart



Model ex_while_loop_SF



ex_while_loop_SF/Chart Executes the desired while-loop

Procedure.

1. Add a Stateflow Chart to your model from the **Stateflow > Stateflow Chart** library.

2. Double-click the chart and select **Tools > Explore** or enter `Ctrl+R` to open the Model Explorer.

3. Add the input, `flag`, and output, `func`, to the chart and specify their data type.

4. Connect the data input and output to the Stateflow chart as shown in the model diagram.

5. In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to `0`.

6. Select **Patterns > Add Loop > While**. The Stateflow Pattern dialog opens.

7. Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	`While Loop` (optional)
While condition	`(flag) && (num_iter<=100)`
Do action	`func; num_iter++;`

8. Place a Subsystem block in your model.

9. Right-click the subsystem and select Subsystem Parameters. The Block Parameters dialog box opens.

10. Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

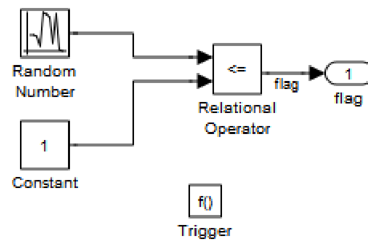
11. Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Function`.

12. From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

13 Specify the name as `func`.

14 Click **Apply** to apply all changes.

15 Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func( ). The Trigger block parameter **Trigger type** is `function-call`. Create the func() algorithm, as shown in the following diagram:



ex_while_loop_SF/func A function that updates the flag

16 Save and close the subsystem.

17 Connect blocks to the Stateflow chart as shown in the model diagram for `ex_while_loop_SF`.

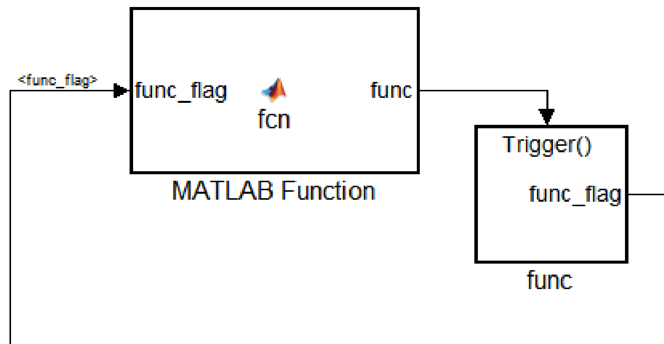
18 Save your model.

Results. The generated code includes the following `ex_while_loop_SF_step` function in the file `ex_while_loop_SF.c`:

```
/* Exported block signals */
int32_T num_iter;  /* '<Root>/Chart Executes the desired while-loop' */
boolean_T flag;    /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Model step function */
void ex_while_loop_SF_step(void)
{
```

```
  /* Stateflow: '<Root>/Chart Executes the desired
   * while-loop' incorporates:
   *  SubSystem: '<Root>/func() A function that
   *              updates the flag'
   */
  /* Gateway: Chart
     Executes the desired while-loop */
  /* During: Chart
     Executes the desired while-loop */
  /* Transition: '<S1>:2' */
  num_iter = 1;
  while (flag && (num_iter <= 100)) {
    /* Transition: '<S1>:3' */
    /* Transition: '<S1>:4' */
    /* Event: '<S1>:12' */
    func();
    num_iter = num_iter + 1;

    /* Transition: '<S1>:5' */
  }

  /* Transition: '<S1>:1' */
}
```

Modeling Pattern for While Loop: MATLAB Function Block



Model ex_while_loop_ML

Procedure.

1 In the Simulink Library Browser, click **Simulink > User Defined Functions**, and drag a MATLAB Function block into your model.

2 Double-click the MATLAB Function block. The MATLAB Function Block Editor opens.

3 In the MATLAB Function Block Editor enter the function, as follows:

```
function fcn(func_flag)

flag = true;
num_iter = 1;

while(flag && (num_iter<=100))
    func;
    flag = func_flag;
    num_iter = num_iter + 1;
end
```

4 Click **File > Save** and close the MATLAB Function Block Editor.

5 Place a Subsystem block in your model, right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.

6 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

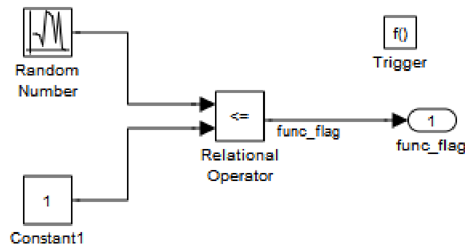
7 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Function`.

8 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

9 Specify the name as `func`.

10 Click **Apply**.

11 Double-click the `func()` subsystem block. In this example, function `func()` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func( ). The Trigger block parameter **Trigger type** is `function-call`. Create the func() algorithm, as shown in the following diagram:



12 Save and close the subsystem.

13 Connect the MATLAB Function block to the `func()` subsystem.

14 Save your model.

Results. The generated code includes the following `while_loop_ML_step` function in the file `while_loop_EML.c`. In some cases an equivalent `for` loop might be generated instead of a `while` loop.

```
/* Exported block signals */
boolean_T func_flag;               /* '<S2>/Relational Operator' */

/* Block states (auto storage) */
D_Work DWork;

/* Model step function */
void while_loop_ML_step(void)
{
  boolean_T func_flag_0;
  boolean_T flag;
  int32_T num_iter;

  /* MATLAB Function Block: '<Root>/MATLAB Function Executes
   * the desired While-Loop' incorporates:
   *  SubSystem: '<Root>/func() updates the "flag"'
   */
  func_flag_0 = func_flag;

  /* MATLAB Function 'MATLAB Function
   * Executes the desired While-Loop': '<S1>:1' */
  /* '<S1>:1:3' */
  flag = TRUE;

  /* '<S1>:1:4' */
  num_iter = 1;
  while (flag && (num_iter <= 100);
       num_iter++) {
    /* '<S1>:1:6' */
    /* '<S1>:1:7' */
    func();

    /* '<S1>:1:8' */
    flag = func_flag_0;

    /* '<S1>:1:9' */
```

```
      num_iter++;
    }
  }
```

Do While loop

C Construct

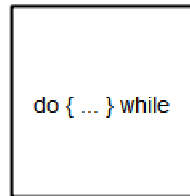
```
num_iter = 1;
do {
   flag = func();
   num_iter++;
   }
while (flag && num_iter <= 100)
```

Modeling Patterns

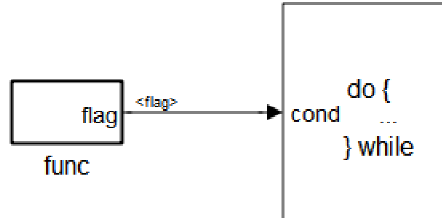
- “Modeling Pattern for Do While Loop: While Iterator Subsystem block” on page 6-59
- “Modeling Pattern for Do While Loop: Stateflow Chart” on page 6-62

Modeling Pattern for Do While Loop: While Iterator Subsystem block

One method for creating a `while` loop is to use a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library.



ex_do_while_loop_SL



ex_do_while_loop_SL/While Iterator Subsystem

Procedure.

1 Drag a While Iterator Subsystem block from the **Simulink > Ports and Subsystems** library into the model.

2 Double-click the While Iterator Subsystem block to open the subsystem.

3 Place a Subsystem block next to the While Iterator block.

4 Right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.

5 Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

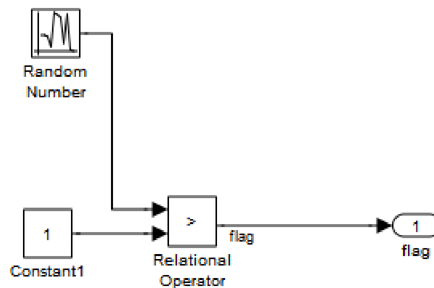
6 Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Function`.

7 From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

8 Specify the name as `func`.

9 Click **Apply**.

10 Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm in func. Create the `func` algorithm as shown in the following diagram:



ex_do_while_loop_SL/While Iterator Subsystem/func

11 Double-click the While Iterator block. This opens the Block Parameters dialog.

12 Set the **Maximum number of iterations** to 100.

13 Specify the **While loop type** as `do-while`.

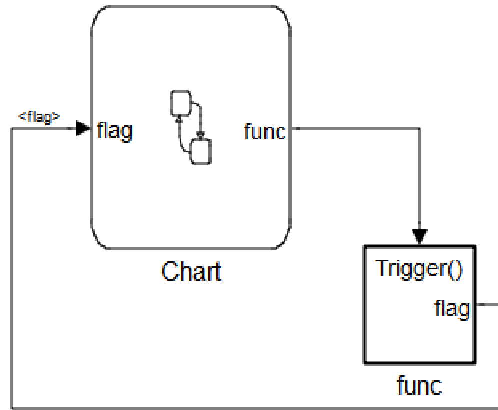
14 Connect blocks as shown in the model and subsystem diagrams.

15 Enter `Ctrl+B` to generate code.

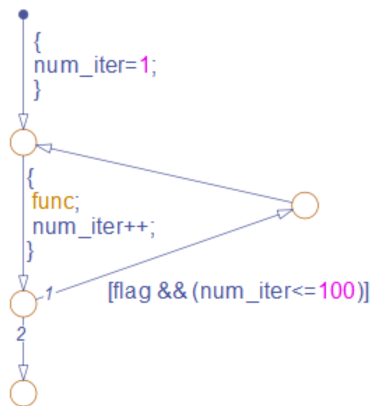
Results.

```
void func(void)
{
   flag = (DWork.NextOutput > (real_T)P.Constant1_Value);
   DWork.NextOutput =
      rt_NormalRand(&DWork.RandSeed) * P.RandomNumber_StdDev +
      P.RandomNumber_Mean;
}

void ex_do_while_loop_SL_step(void)
{
   int32_T s1_iter;

   s1_iter = 1;
   do {
      func();
      s1_iter++;
   } while (flag && (s1_iter <= 100));

}
```

Modeling Pattern for Do While Loop: Stateflow Chart



ex_do_while_loop_SF



ex_do_while_loop_SF/Chart

1 Add a Stateflow Chart to your model from the **Stateflow > Stateflow Chart** library.

2. Double-click the chart and select **Tools > Explore** or enter `Ctrl+R` to open the Model Explorer.

3. Add the inputs and outputs to the chart and specify their data type.

4. Connect the data input and output to the Stateflow chart.

5. In the Model Explorer, select the output variable, then, in the right pane, select the **General** tab and set the **Initial Value** to `0`.

6. Select **Patterns > Add Loop > While**. The Stateflow Pattern dialog opens.

7. Fill in the fields for the Stateflow Pattern dialog box as follows:

Description	`While Loop` (optional)
While condition	`(flag) && (num_iter<=100)`
Do action	`func; num_iter++;`

8. Place a Subsystem block in your model.

9. Right-click the subsystem and select **Subsystem Parameters**. The Block Parameters dialog box opens.

10. Select the **Treat as atomic unit** parameter to configure the subsystem to generate a function. This enables parameters on the **Code Generation** tab.

11. Select the **Code Generation** tab. From the **Function packaging** list, select the option, `Function`.

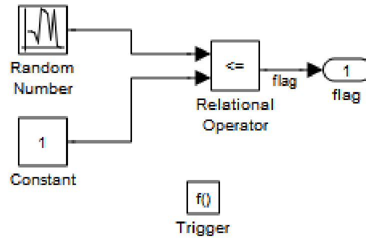
12. From the **Function name options** list, select the option, `User specified`. The **Function name** parameter is displayed.

13. Specify the name as `func`.

14. Click **Apply** to apply all changes.

15. Double-click the `func` subsystem block. In this example, function `func` has an output `flag` set to `0` or `1` depending on the result of the algorithm

in func. The Trigger block parameter **Trigger type** is `function-call`. Create the func algorithm, as shown in the following diagram:



ex_do_while_loop_SF/func Updates the flag

16 Save and close the subsystem.

17 Connect blocks to the Stateflow chart as shown in the model diagram for `ex_do_while_loop_SF`.

18 Save your model.

Results.

```
void ex_do_while_loop_SF_step(void)
{
   int32_T sf_num_iter;
  num_iter = 1;
   do {
      func();
      num_iter++;
    } while (flag && (sf_num_iter <= 100));

}
```

Functions

In this section...

"Function Call" on page 6-65

"Function Prototyping" on page 6-67

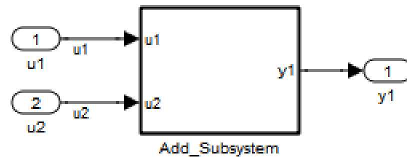
"External C Functions" on page 6-70

Function Call

To generate a function call, add a subsystem, which implements the operations that you want.

C Construct

```
void add_function(void)
{
   y1 = u1 + u2;
}
```

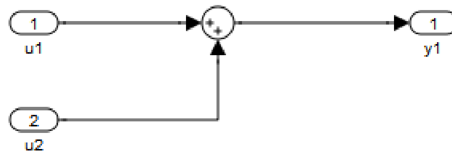


ex_function_call

Procedure

1. Create a model containing a subsystem. In this example, the subsystem has two inputs and returns one output.

2. Double-click the subsystem. Create `Add_Subsystem`, as shown.



ex_function_call/Add_Subsystem

3 Right-click the subsystem to open the Subsystem Parameters dialog box.

4 Select the **Treat as atomic unit** parameter. This enables parameters on the **Code Generation** tab.

   Select the **Code Generation** tab. For the **Function packaging** parameter, from the drop-down list, select `Function`.

5 For the **Function name options** parameter, from the drop-down list, select `User specified`.

6 In the **Function name** field, enter the subsystem name, `add_function`.

7 Click **Apply** and **OK**.

8 Press **Ctrl+B** to build and generate code.

Results

In `ex_function_call.c`, the function is called from `ex_function_call_step`:

```
void ex_function_call_step(void)
{
   add_function();
}
```

The function prototype is externed through the subsystem file, `add_function.h`.

```
extern void add_function(void);
```

The function definition is in the subsystem file `add_function.c`:

```
void add_function(void)
{
   function_call_Y.y1 = u1 + u2;
}
```

Function Prototyping

C Construct

```
double add_function(double u1, double u2)
{
   return u1 + u2;
}
```

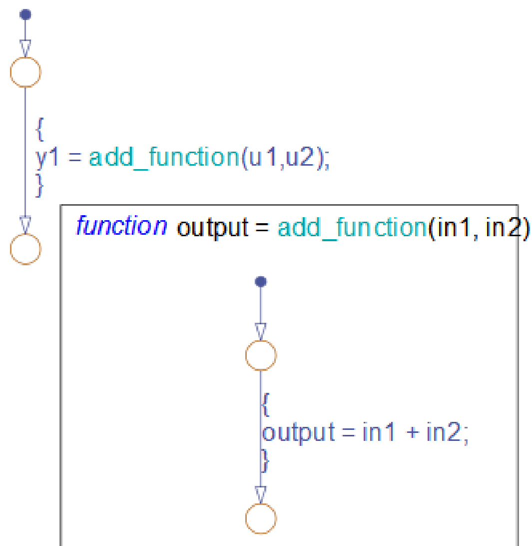
Modeling Patterns

- "Function Call Using Graphical Functions" on page 6-67
- "Control Function Prototype of the *model*_step Function" on page 6-69

Function Call Using Graphical Functions

Procedure.

1. Follow the steps for"Setting Up an Example Model With a Stateflow Chart" on page 6-5. This example model contains two Inport blocks and one Outport block.
2. Name the example model `ex_func_SF`.
3. In the **Stateflow Editor**, create a graphical function by clicking the **f()** button and dragging a graphical function into the Stateflow chart.
4. Edit the graphical function signature to: `output = add_function(u1, u2)`.
5. Add the transition action, as shown in the following diagram.



ex_func_SF/Chart
In the Stateflow chart is an example of a simple transition that calls `add_function`.

6 Open the Model Explorer. From the Model Hierarchy tree, select **ex_func_SF > Chart > f()add_function**. On the right pane, specify the **Function Inline Option** as `Function`.

7 From the Model Hierarchy tree, click **Chart** and on the right pane select the **Export Chart Level Graphical Functions(Make Global)** parameter. This makes the function available globally to the entire model.

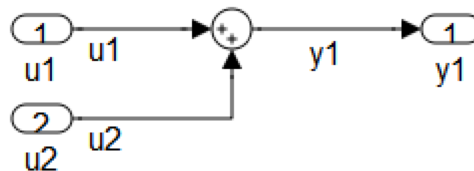
8 Press **Ctrl+B** to build the model and generate code.

Results. `ex_func_SF.c` contains the generated code:

```
extern real_T add_function(real_T sf_in1, real_T sf_in2)
{
   return sf_in1 + sf_in2;
}
.
```

```
.
.
void ex_func_SF_step(void)
{

   ex_func_SF_B.y1 = add_function(u1, u2);

   ex_func_SF_Y.y1 = ex_func_SF_B.y1;
}
```

Control Function Prototype of the *model*_step Function



ex_control_step_function

Procedure.

1. Create the model, `ex_control_step_function`. See "Configuring a Signal" on page 6-3 and "Configuring Input and Output Ports" on page 6-4, for more information.

2. Press **Ctrl+E** to open the Configuration Parameters dialog box.

3. On the **Code Generation > Interface** pane, click **Configure Model Functions** to open the Model Interface dialog box.

4. Specify the **Function specification** parameter as `Model specific C prototypes`.

5 Click **Get Default Configuration** to update the **Configure model initialize and step functions** section and list the input and output arguments.

6 To configure the function output argument to pass a pointer, in the **Step function arguments** table, specify the **Category** for the Outport as a `Pointer`. In addition, you can specify the step function arguments order and type qualifiers.

7 To validate your changes, click **Validate**.

8 Press **Ctrl+B** to build the model and generate code.

Results. `ex_control_step_function.c` contains the generated code:

```
void ex_control_step_function_custom(real_T arg_u1, real_T arg_u2, ...
                                  real_T *arg_y1)
{
    (*arg_y1) = arg_u1 + arg_u2;
}
```

External C Functions

C Construct

```
extern double add(double, double);
```

```
#include "add.h"
double add(double u1, double u2)
{
   double y1;
   y1 = u1 + u2;
   return (y1);
}
```

Modeling Patterns

There are several methods for integrating legacy C functions into the generated code. These methods either create an S-function or make a call to an external C function. For more information on S-functions, see "Introduction to S-Functions for Code Generation".

- "Using the Legacy Code Tool to Create S-functions" on page 6-71
- "Using a Stateflow Chart to Make Calls to C Functions" on page 6-73
- "Using a MATLAB Function Block to Make Calls to C Functions" on page 6-75

Using the Legacy Code Tool to Create S-functions

This method uses the Legacy Code Tool to create an S-function and generate a TLC file. The code generation software uses the TLC file to generate code from this S-function. The advantage of using the Legacy Code Tool is that the generated code is fully inlined and does not need any wrapper functions to access the custom code.

Procedure.

1 Create a C header file named `add.h` that contains the function signature:

```
extern double add(double, double);
```

2 Create a C source file named `add.c` that contains the function body:

```
double add(double u1, double u2)
{
   double y1;
   y1 = u1 + u2;
   return (y1);
}
```

3 To build an S-function for use in both simulation and code generation, Run the following script or execute each of these commands at the MATLAB command line:

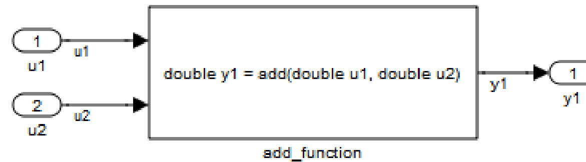
```
%% Initialize legacy code tool data structure
def = legacy_code('initialize');
```

```
%% Specify Source File
def.SourceFiles = {'add.c'};

%% Specify Header File
def.HeaderFiles = {'add.h'};

%% Specify the Name of the generated S-function
def.SFunctionName = 'add_function';

%% Create a c-mex file for S-function
legacy_code('sfcn_cmex_generate', def);

%% Define function signature and target the Output method
def.OutputFcnSpec = ['double y1 = add(double u1, double u2)'];

%% Compile/Mex and generate a block that can be used in simulation
legacy_code('generate_for_sim', def);

%% Create a TLC file for Code Generation
legacy_code('sfcn_tlc_generate', def);

%% Create a Masked S-function Block
legacy_code('slblock_generate', def);
```

The output of this script produces:

- A new model containing the S-function block
- A TLC file named `add_function.tlc`.
- A C source file named `add_function.c`.
- A mexw32 dll file named `add_function.mexw32`

4 Add inport blocks and an outport block and make the connections, as shown in the model.



ex_function_call_lct

5 Name and save your model. In this example, the model is named `ex_function_call_lct.mdl`.

6 Press **Ctrl+B** to build the model and generate code.

Results. The following code is generated in `ex_function_call_lct.c`:

```
real_T u1;
real_T u2;
real_T y1;
void ex_function_call_lct_step(void)
{
   y1 = add(u1, u2);
}
```

The user-specified header file, `add.h`, is included in `ex_function_call_lct.h`:

```
#include "add.h"
```

Using a Stateflow Chart to Make Calls to C Functions

Procedure.

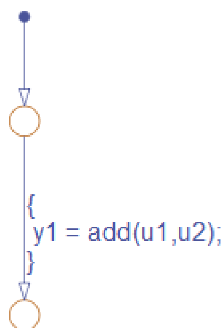
1 Create a C header file named `add.h` that contains the example function signature.

2 Create a C source file named `add.c` that contains the function body.

3 Follow the steps for "Setting Up an Example Model With a Stateflow Chart" on page 6-5. This example model contains two Inport blocks and one Outport block.

4 Name the example model `ex_exfunction_call_SF`.

5 Double-click the Stateflow chart and edit the chart as shown. Place the call to the `add` function within a transition action.



ex_exfunction_call_SF/Chart

6 On the **Stateflow Editor**, select **Tools > Open Simulation Target**. The Configuration Parameters dialog box opens with the **Simulation Target** pane displayed.

7 On the Configuration Parameters dialog box, select **Simulation Target > Custom Code**. In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the `#include` statement:

```
#include "add.h"
```

8 In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.

9 Press **Ctrl+B** to build the model and generate code.

Results. `ex_exfunction_call_SF.c` contains the following code in the step function:

```
real_T u1;
real_T u2;
real_T y1;

void exfunction_call_SF_step(void)
{
  y1 = (real_T)add(u1, u2);
}
```

`ex_exfunction_call_SF.h` contains the include statement for `add.h`:

```
#include "add.h"
```

Using a MATLAB Function Block to Make Calls to C Functions

Procedure.

1 Create a C header file named `add.h` that contains the example function signature.

2 Create a C source file named `add.c` that contains the function body.

3 In the Simulink Library Browser, click **Simulink > User Defined Functions**, and drag a MATLAB Function block into your model.

4 Double-click the MATLAB Function block. The MATLAB Function Block Editor opens.

5 Edit the function to include the statement:

```
function y1 = add_function(u1, u2)

%Set the class and size of output
y1 = u1;

%Call external C function
y1 = coder.ceval('add',u1,u2);
```

```
end
```

6 In the MATLAB Function Block Editor, select **Tools > Open Simulation Target**. The Configuration Parameters dialog box opens with the **Simulation Target** pane displayed.

7 On the left pane of the Configuration Parameters dialog box under **Simulation Target**, select **Custom Code**. In the **Include custom C code in generated** section, on the left pane, select **Header file** and in the **Header file** field, enter the statement, :

```
#include "add.h"
```

8 In the **Include list of additional** section, select **Source files** and in the **Source files** field, enter `add.c`.

9 Add two Inport blocks and one Outport block to the model and connect to the MATLAB Function block.

10 Configure the signals: `u1`, `u2`, and `y1`, as described in "Configuring a Signal" on page 6-3.

11 Save the model as `ex_exfunction_call_ML.mdl`.

12 Press **Ctrl+B** to build the model and generate code.

Results. `ex_exfunction_call_ML.c` contains the following code:

```
real_T u1;
real_T u2;
real_T y1;

void ex_exfunction_call_ML_step(void)
{
     y1 = add(u1, u2);
}
```

`ex_exfunction_call_ML.h` contains the `#include` statement for `add.h`:

```
#include "add.h"
```

Preprocessor Directives

In this section...
"Macro Definitions (#define)" on page 6-77
"Conditional Inclusions (#if / #endif)" on page 6-79

Macro Definitions (#define)

C Construct

```
#define p_1 9.8;
```

Modeling Patterns

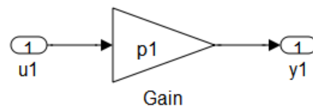
"Using a 'Define' Custom Storage Class" on page 6-77

"Using a Custom Header File" on page 6-78

Using a 'Define' Custom Storage Class

Procedure.

1 Create a model containing a Gain block.



2 Press **Ctrl+E** to open the Configuration Parameters dialog box.

3 In the Configuration Parameter dialog box, on the **Optimization > Signals and Parameters** pane, select **Inline parameters**.

4 Click **Apply** and **OK**.

5 In your model, double-click the Gain block. The Block Parameters dialog box opens.

6 In the **Value** field, enter a variable name. In this example, the variable name is `p1`.

7 Press **Ctrl+H** to open the Model Explorer. On the Model Hierarchy pane, select the Base Workspace.

8 To add an MPT parameter object, in the menu bar, select **Add > MPT Parameter**. The parameter appears in the **Contents of: Base Workspace** pane.

9 Double-click the `mpt.Parameter` object and change its name to `p1`.

10 Click the `p1` parameter. The data object parameters are displayed in the right pane of the Model Explorer.

11 In the **Value** field, enter `9.8`. In the **Code generation options** section, click the **Storage Class** drop-down list and select `Define(Custom)`.

12 Press **Ctrl+B** to generate code.

Results. The generated code includes the inlined parameter, `p1`, in `ex_define_data_object.c`:

```
/* Model step function */
void ex_define_data_object_step(void)
{
  rtY.y1 = p1 * rtU.u1;

}
```

Using a Custom Header File

Procedure.

1 Follow steps 1 through 10 of “Using a ’Define’ Custom Storage Class” on page 6-77.

2 In the Simulink.Parameter dialog box for p1, in the **Value** field, enter 9.8. In the **Code generation options** section, click the **Storage Class** drop-down list and select ImportFromFile(Custom).

3 In the **Header file** parameter, enter the name of the header file, in this example, external_params.h.

4 Click **Apply** and **OK**.

5 Create the C header file, external_params.h that contains the #define statement:

```
#ifndef _EXTERNAL_PARAMS
#define _EXTERNAL_PARAMS

#define p1 9.8

#endif

/* EOF */
```

6 Press **Ctrl+B** to generate code.

Results. The generated code includes the inlined parameter, p1, in ex_define_data_object.c:

```
/* Model step function */
void ex_define_data_object_step(void)
{
  ex_define_data_object_Y.Out1 = p1 * ex_define_data_object_U.In1;

}
```

Conditional Inclusions (#if / #endif)

You can generate preprocessor conditional directives in your code by implementing variant blocks (Model Variants block or Variant Subsystem block) in your model. In the generated code, preprocessor conditional directives select a section of code to execute at compile time. To implement variants in your model, see “Modeling Variant Systems” in the Simulink

documentation. To generate code for variants, see “Generating Code for Variant Systems” on page 4-2.

Structures

In this section...

"Typedef" on page 6-81

"Structures for Parameters" on page 6-83

"Structures for Signals" on page 6-85

"Nested Structures" on page 6-89

"Bitfields" on page 6-92

Typedef

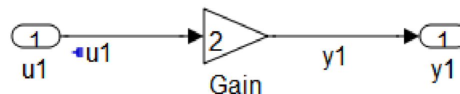
To generate a `typedef` definition, use a `Simulink.AliasType` data object.

C Construct

```
typedef double float_64;
```

Procedure

1 Create the `ex_get_typedef` model with a Gain block.



2 In the Gain block parameter dialog box, select the **Parameter Attributes** tab, and specify the **Parameter data type** as `double`.

3 Right-click the `u1` signal and select **Signal Properties**. In the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**.

4 Right-click the `y1` signal and select **Signal Properties**. In the Signal Properties dialog box, select the **Code Generation** tab, and specify the **Storage class** parameter as `ExportedGlobal`.

5 Create a new alias type by using a Simulink.AliasType data object. At the MATLAB command line, enter:

```
float_64 = Simulink.AliasType;
```

6 In the base workspace, double-click `float_64`. The Simulink.AliasType dialog box opens.

7 Specify the **Base type** parameter as `double`. Click **Apply** and **OK**.

8 Create a data object for the `u1` signal. In the base workspace, select **Add > Simulink Signal**, and name it `u1`. Specify the **Data type** parameter as `float_64` and the **Storage class** parameter as `Global(custom)`.

Note You can also specify an output data type for Simulink blocks using the new alias type.

9 Click **Apply** and **OK**.

10 Press **Ctrl+B** to generate code.

Note An alternative method for defining a `typedef` is to import the alias type from a custom header file. If you want to import all the `typedef`s from a C header file, using this alternative method is useful.

Results

The generated code includes the `typedef` definition, which is declared within `#ifndef` and `#endif` statements in the `ex_get_typedef_types.h` file.

```
#ifndef _DEFINED_TYPEDEF_FOR_float_64_
#define _DEFINED_TYPEDEF_FOR_float_64_

typedef real_T float_64;
typedef creal_T cfloat_64;

#endif
```

Note `real_T` is the Embedded Coder `typedef` for `double` .

The generated code also includes the declaration of the Simulink data objects of the alias type in `ex_get_typedef.c`.

```
float_64 y1;
float_64 u1;
```

Structures for Parameters

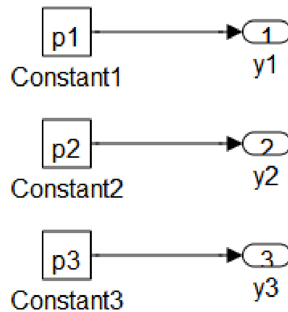
To generate a structure containing parameters, use a `mpt.Parameter` object with a `Struct (custom)` storage class.

C Construct

```
typdef struct {
  double p1;
  double p2;
  double p3;

} my_struct_type;

my_struct_type my_struct={1.0,2.0,3.0};
```

Procedure

1 Create the `ex_struct_param` model with three Constant blocks and three Outport blocks.



2 Create a data object for each parameter, `p1`, `p2`, and `p3`. At the MATLAB command line, enter:

```
p1 = mpt.Parameter;
p2 = mpt.Parameter;
p3 = mpt.Parameter;
```

3 In the base workspace, double-click one of the parameter data objects to open the `mpt.Parameter` dialog box.

4 Specify a **Value** parameter for each parameter object.

5 Specify the **Storage class** parameter as `Struct (Custom)` for each parameter object.

6 In the **Custom Attributes** section, specify the **Struct name** as `my_struct`. Click **Apply** and **OK**.

7 Press **Ctrl+E** to open the Configuration Parameters dialog box.

8 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.

9 Click **Apply** and **OK**.

10 Press **Ctrl+B** to generate code.

Results

The generated code includes the typedef definition for a structure, which is declared in the ex_struct_param_types.h file.

```
/* Type definition for custom storage class: Struct */
   typedef struct my_struct_tag {
      real_T p1;
      real_T p2;
      real_T p3;
   } my_struct_type;
```

The generated code also includes the declaration of my_struct in ex_struct_param.c.

```
/* Definition for custom storage class: Struct */
my_struct_type my_struct = {
     /* p1 */
     1.0,

     /* p2 */
     2.0,

     /* p3 */
     3.0
};
```

Structures for Signals

To generate a structure containing parameters, use a mpt.Signal object with a Struct (custom) storage class or a Simulink non-virtual bus object.

C Construct

```
typedef struct {
   double u1;
   double u2;
   double u3;
} my_signals;
```

Modeling Patterns

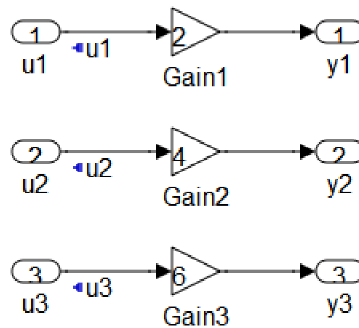
"Structure for Signals Using a 'Struct' Custom Storage Class" on page 6-86

"Structure for Signals Using a Simulink Non-Virtual Bus Object" on page 6-87

Structure for Signals Using a 'Struct' Custom Storage Class

Procedure.

1 Create the `ex_signal_struct_csc` model using the blocks shown and follow the steps to configure the signals and model.



2 Double-click a Gain block to open the block parameter dialog box. Set the values of the Gain blocks as shown in the model diagram.

3 Right-click the `u1` signal and select **Signal Properties**. In the Signal Properties dialog box, select **Signal name must resolve to Simulink signal object**. Repeat for signals `u2` and `u3`.

4 At the MATLAB command line, create a `mpt.Signal` data object for each input signal.

```
u1 = mpt.Signal;
u2 = mpt.Signal;
u3 = mpt.Signal;
```

Note You can also create a data object in the Model Explorer base workspace, by selecting **Add > MPT Signal**.

---

5. In the base workspace, configure each of the data objects, `u1`, `u2`, and `u3`. Double-click a data object, to open the `mpt.Signal` parameter dialog box.
6. Specify the **Data type** parameter as `auto` and the **Storage class** parameter as `Struct (custom)`.
7. Click **Apply** and **OK**.
8. Press **Ctrl+B** to generate code.

Results. The generated code includes the `typedef` definition for a structure, which is declared in the `ex_signal_struct_csc_types.h` file.

```
/* Type definition for custom storage class: Struct */
typedef struct my_signal_struct_tag {
   real_T u1;
   real_T u2;
   real_T u3;
 } my_signal_struct_type;
```

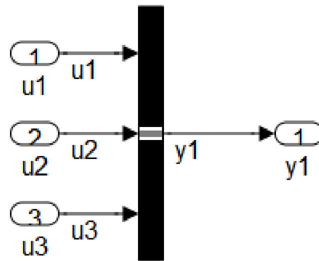
The generated code also includes the declaration of `my_signal_struct` in `ex_signal_struct_csc.c`.

```
/* Definition for custom storage class: Struct */
my_signal_struct_type my_signals;
```

Structure for Signals Using a Simulink Non-Virtual Bus Object

Procedure.

1. Create the `ex_signal_struct_bus` model using the blocks shown and follow the steps to configure the bus object and model.



2 Add the Inport blocks, an Outport block, and a Bus Creator block to your diagram.

3 Double-click the Bus Creator block to open the block parameter dialog box.

4 Specify the **Number of inputs** parameter as `3`. Click **Apply**.

5 In your model diagram, connect the three Inport blocks to the three inports of the Bus Creator block. Also, connect the outport of the Bus Creator block to the Outport block.

6 Label the signals as shown in the model diagram.

7 In the Bus Creator block parameter dialog box, **Signals in bus** now displays the signals connected to the Bus Creator block.

8 Create a bus object named `MySignals` that includes signals `u1`,`u2`, and `u3`. For more information on creating bus objects, see "Using the Bus Editor". Once the bus object, `MySignals`, is created, it appears in the base workspace.

9 In the Bus Creator block parameter dialog box, select the **Output as nonvirtual bus** parameter, which specifies that bus signals must be grouped into a structure in the generated code.

10 Click **Apply** and **OK**.

11 Press **Ctrl+B** to generate code.

Results. The generated code includes the `typedef` definition for a structure, which is declared in the `signal_struct_bus_types.h` file.

```
typedef struct {
   real_T u1;
   real_T u2;
   real_T u3;
} MySignals;
```

Nested Structures

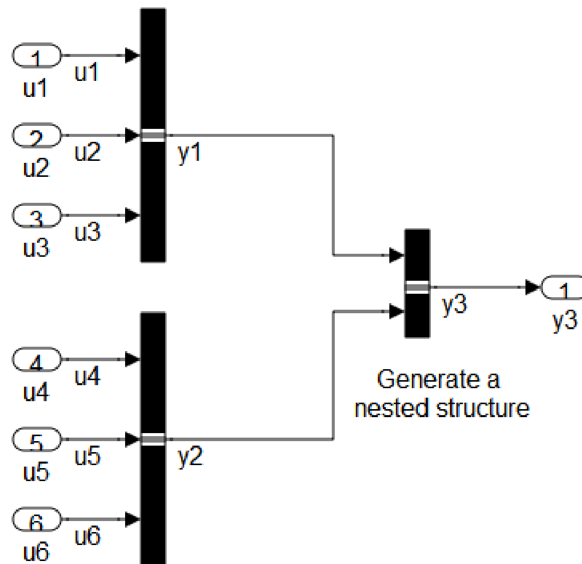
One way to create nested structures of signals in the generated code is by using multiple non-virtual bus objects. When nesting bus objects, all of the bus objects must either be non-virtual, or all of them must be virtual.

C Construct

```
typedef struct {
   double u1;
   double u2;
   double u3;
} my_signals123;

typedef struct {
   double u4;
   double u5;
   double u6;
} my_signals456;

typedef struct {
   my_signals123 y1;
   my_signals456 y2;
 } nested_signals;
```

Procedure

1 Create the `ex_nested_structure` model using the blocks shown and follow the steps to configure the bus objects and model.



2 For each bus in the model, follow the instructions for "Structure for Signals Using a Simulink Non-Virtual Bus Object" on page 6-87, creating bus objects `My_Signals_123` and `My_Signals_456`.

3 Drag a Bus Creator block into your model. Configure the Bus Creator block so that it takes in signals from different buses.

4 Double-click the Bus Creator block to open the block parameter dialog box.

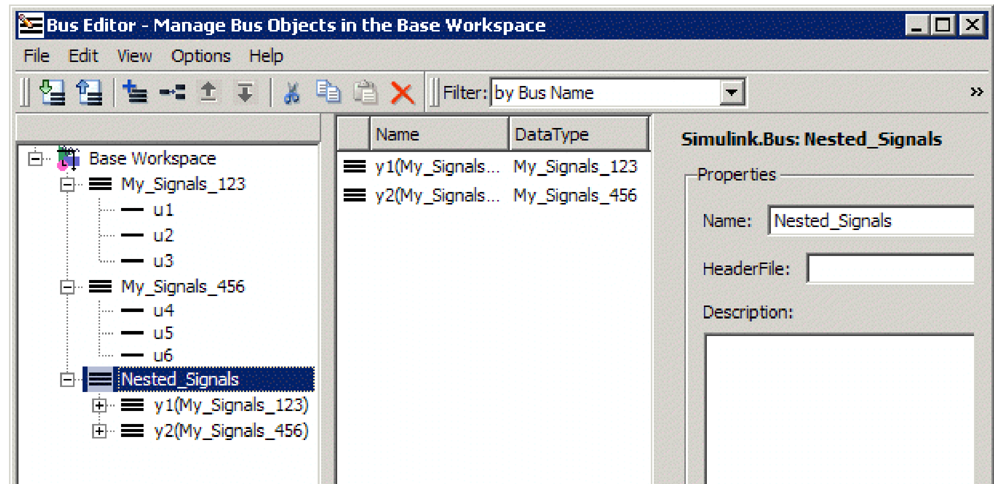
5 Specify the **Number of inputs** parameter as `2`. Click **Apply**.

6 In your model diagram, connect the two bus outports to the inports of the new Bus Creator block.

7 Label the signals as shown in the model diagram.

8 In the Bus Creator block parameter dialog box, **Signals in bus** now displays the signals, `y1` and `y2`, connected to the Bus Creator block.

9 Create a bus object named `Nested_Signals` that includes signals `y1` and `y2`, where the **DataType** for `y1` is `My_Signals_123` and the **DataType** for `y2` is `My_Signals_456`.



For more information on creating bus objects, see "Using the Bus Editor". Once the bus object, `Nested_Signals`, is created, it appears in the base workspace.

10 In the Bus Creator block parameter dialog box, select the **Output as nonvirtual bus** parameter, which specifies that bus signals must be grouped into a structure in the generated code.

11 Click **Apply** and **OK**.

12 Press **Ctrl+B** to generate code.

Results

The generated code includes the `typedef` definitions for structures, which are declared in the `ex_nested_structure_types.h` file.

```
#ifndef _DEFINED_TYPEDEF_FOR_My_Signals_123_
#define _DEFINED_TYPEDEF_FOR_My_Signals_123_

typedef struct {
```

```
   real_T u1;
   real_T u2;
   real_T u3;
} My_Signals_123;

#endif

#ifndef _DEFINED_TYPEDEF_FOR_My_Signals_456_
#define _DEFINED_TYPEDEF_FOR_My_Signals_456_

typedef struct {
   real_T u4;
   real_T u5;
   real_T u6;
} My_Signals_456;

#endif

#ifndef _DEFINED_TYPEDEF_FOR_Nested_Signals_
#define _DEFINED_TYPEDEF_FOR_Nested_Signals_

typedef struct {
   My_Signals_123 y1;
   My_Signals_456 y2;
} Nested_Signals;

#endif
```

Bitfields

One way to create bitfields in the generated code is by using a `mpt.Parameter` object with `Bitfield (Custom)` storage class.

C Construct

```
typedef struct {
   unsigned int p1 : 1;
   unsigned int p2 : 2;
   unsigned int p3 : 3;
} my_struct_type
```

Procedure

1. Using the model, `ex_struct_param`, in "Structures for Parameters" on page 6-83, rename the model as `ex_struct_bitfield_CSC`.

2. Create a data object for each parameter, `p1`, `p2`, and `p3`. At the MATLAB command line, enter:

```
p1 = mpt.Parameter;
p2 = mpt.Parameter;
p3 = mpt.Parameter;
```

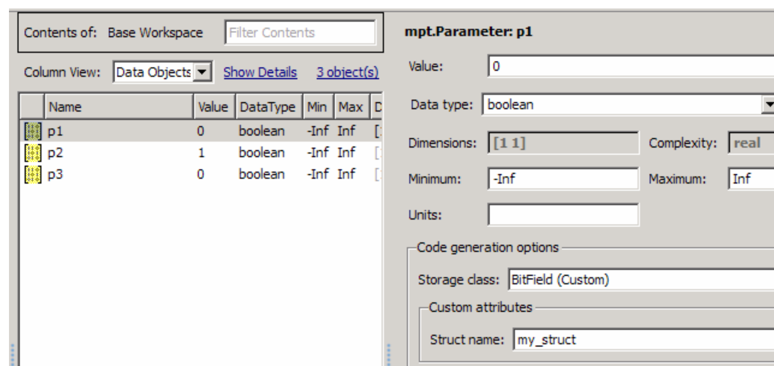
3. In the base workspace, double-click one of the parameter data objects to open the `mpt.Parameter` dialog box.

4. Specify the **Value** parameter for each parameter object.

5. Specify the **Storage class** parameter as `Bitfield (Custom)` for each parameter object.

6. In the **Custom Attributes** section, specify the **Struct name** as `my_struct`. Click **Apply** and **OK**.

7. Specify the data objects for each parameter.



8. Press **Ctrl+E** to open the Configuration Parameters dialog box.

9. Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.

10 Click **Apply** and **OK**.

11 Press **Ctrl+B** to generate code.

Results

The generated code of the model, `ex_struct_bitfield_CSC`, includes the `typedef` definition for a Bitfield, which is declared in the `ex_struct_bitfield_CSC_types.h` file.

```
/* Type definition for custom storage class: BitField */
typedef struct my_struct_tag {
  uint_T p1 : 1;
  uint_T p2 : 1;
  uint_T p3 : 1;
} my_struct_type;
```

Arrays

In this section...
"Arrays for Parameters" on page 6-95
"Arrays for Signals" on page 6-97

Arrays for Parameters

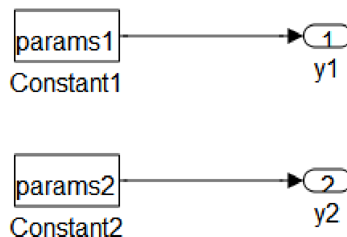
To create an array in the generated code, you can use a constant parameter in the base workspace, or a `mpt.Parameter`.

C Construct

```
int params[5]= {1,2,3,4,5};
```

Procedure

1. Create a model, `ex_array_params`, containing the Constant blocks and Outport blocks and label the blocks as shown in the model diagram.



2. Double-click the `Constant1` block and give the **Constant value** the name of a parameter, `params1`.

3. Double-click the `Constant2` block and give the **Constant value** the name of a parameter, `params2`.

4 To create the parameters in the base workspace, at the MATLAB command line, enter:

```
params1 = [1,2,3,4,5];
params2 = mpt.Parameter;
```

5 In the base workspace, double-click `params2` to open the `mpt.Parameter` dialog box.

6 In the **Value** field, specify the five dimensional array, `[1 2 3 4 5]`.

7 Press **Ctrl+E** to open the Configuration Parameters dialog box.

8 Open the **Optimization > Signals and Parameters** pane, and select the **Inline parameters** parameter.

9 Click **Apply** and **OK**.

10 Press **Ctrl+B** to generate code.

Results

The generated code includes the array, `params2`, in the `ex_array_params.c` file:

```
int16_T params2[5] = { 1, 2, 3, 4, 5 } ;
```

The data object, `params1`, is defined in the `array_params_data.c` file:

```
/* Constant parameters (auto storage) */
const ConstParam_array_params array_params_ConstP = {
  /* Computed Parameter: Constant1_Value
   * Referenced by: '/Constant1'
   */
  { 1, 2, 3, 4, 5 }
};
```

where `ConstParam_array_params` is a structure containing the array and defined in the `array_params.h` file.

```
typedef struct {
   /* Computed Parameter: Constant1_Value
```

```
    * Referenced by: '/Constant1'
    */
    int16_T Constant1_Value[5];
} ConstParam_array_params;
```

Arrays for Signals

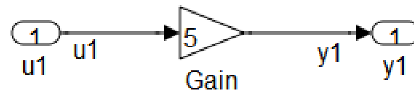
To create an array in the generated code for signal data, you can specify a signal as `ExportedGlobal`, or use a `mpt.Signal` object.

C Construct

```
int u1[5];
int y1[5];
```

Procedure

1. Create the `ex_array_signals` model using the blocks shown and follow the steps to configure the signals and model.



2. Double-click the Inport block to open the Inport block parameter dialog box.

3. Select the **Signal Attributes** tab and specify the **Port dimensions** parameter as `5`, for an array of length 5.

4. Click **OK**.

5. Right-click the `u1` signal line and select **Signal Properties**.

6. Select the **Code Generation** tab and specify the **Storage Class** parameter as `ExportedGlobal`.

7. Repeat steps 5 and 6 for signal `y1`.

8. Press **Ctrl+B** to generate code.

Note Alternatively, you can use Simulink data objects (mpt.Signal) to specify the storage class and dimensions for the signals, u1 and y1.

Results

The generated code includes arrays for u1 and y1 in the ex_array_signals.c file:

```
int16_T u1[5];
int16_T y1[5];
```

In this case, a for loop is generated to carry out the gain operations on all elements of the input signal.

```
int32_T i;
for (i = 0; i < 5; i++) {
   y1[i] = (int16_T)(5 * u1[i]);
}
```

However, if the dimension of the array is less than a threshold value (typically 5), code generation might not include a for loop for array operations.

Pointers

In this section...

"Pointers for Signals" on page 6-99

"Pointers for Signals and Parameters Using Simulink Data Objects" on page 6-100

Pointers for Signals

To create a pointer in the generated code, you can configure a signal to use the `ImportedExternPointer` storage class or use an `mpt.Signal` (or `mpt.Parameter` for parameters) object with an `ImportedExternPointer` storage class.

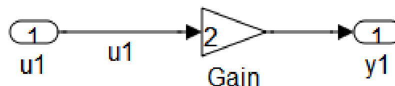
C Construct

```
extern double *u1;
```

Procedure

This is a quick method to obtain pointers in the generated code. You cannot control the data type, which is decided by the model compilation process.

1. Create the `ex_pointer_signal` model using the blocks shown and follow the steps to configure the signals and model.



2. Label the signal to be imported as a pointer, in this example, `u1`.
3. Right-click the `u1` signal line and select **Signal Properties**.
4. Select the **Code Generation** tab and specify the **Storage Class** parameter as `ImportedExternPointer`.

5 Click **OK**.

6 Press **Ctrl+B** to generate code.

Results

The generated code includes the `extern` declaration for the pointer in the `ex_pointer_signal_private.h` file.

```
extern real_T *u1;
```

Pointers for Signals and Parameters Using Simulink Data Objects

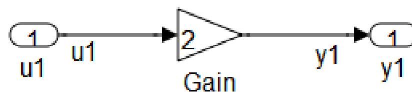
C Construct

```
extern double *u1;
```

Procedure

You can control the data type by using a Simulink data object to generate a pointer. You can use this procedure for either a signal or parameter. To create a pointer for a parameter, use an `mpt.Parameter` instead of an `mpt.Signal` data object described in step 3.

1 Create the `ex_pointer_signal_data_object` model using the blocks shown and follow the steps to configure the signals and model.



2 Label the signal to be imported as a pointer, in this example, `u1`.

3 At the MATLAB command line, create a data object for signal `u1`.

```
u1 = mpt.Signal;
```

4 In the base workspace, double-click `u1` to open the `mpt.Signal` dialog box.

5 Specify the **Storage class** parameter as `ImportedExternPointer`.

6 Click **Apply** and **OK**.

7 Press **Ctrl+B** to generate code.

Results

The generated code includes the `extern` declaration for the pointer in the `ex_pointer_signal_data_object_private.h` file.

```
extern real_T *u1;
```

The `ex_pointer_signal_data_object_private.h` file imports the pointer into the generated code. To compile the code, you must declare and define the pointer in the main program.

Defining Data Representation and Storage for Code Generation

- Chapter 7, "Using mpt Data Objects"
- Chapter 8, "Creating and Using Custom Storage Classes"
- Chapter 9, "Memory Sections"
- Chapter 10, "Optimizing Buses for Code Generation"
- Chapter 11, "Renaming and Replacing Data Types"
- Chapter 12, "Managing Data Definitions and Declarations With the Data Dictionary"
- Chapter 13, "Managing Placement of Data Definitions and Declarations"
- Chapter 14, "Specifying the Persistence Level for Signals and Parameters"

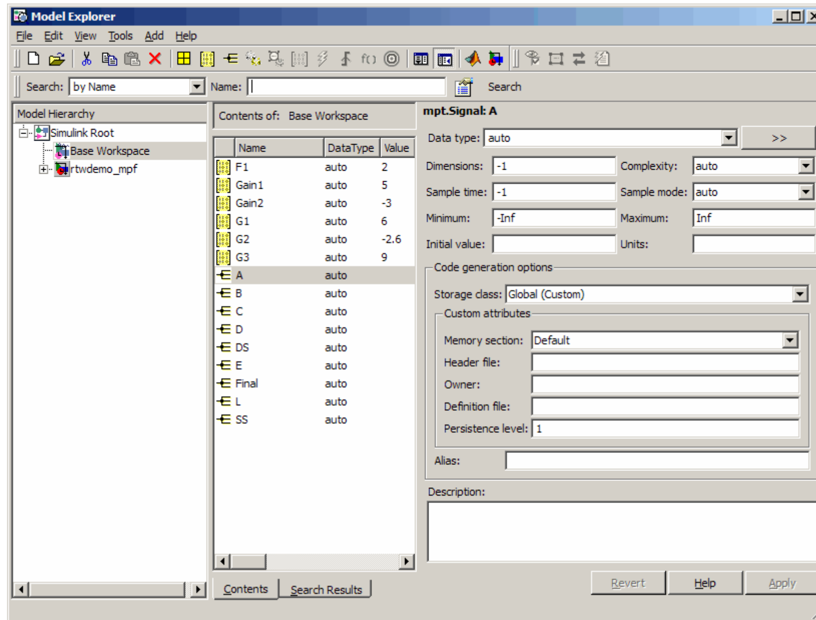
Using mpt Data Objects

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

Note You can create `mpt.Signal` and `mpt.Parameter` objects in the base MATLAB or model workspace. However, if you create the object in a model workspace, the object's storage class must be set to `auto`.

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

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



Parameter and Signal Property Values

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	User object type	*`auto`	Prenamed and predefined property sets that are registered in the `sl_customization.m` file. (See “Registering mpt User Object Types” on page 12-48.) This field is unavailable if no user object type is registered. Select `auto` if this field is available but you do not want to apply the properties of a user object type to a selected data object. The fields on the Model Explorer are populated with default values.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
		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 `sl_customization.m` file. The fields on the Model Explorer are automatically populated with those values.
Parameter	`Value`	`*0`	The data type and numeric value of the data object. For example, `int8(5)`. The numeric value is used as an initial parameter value in the generated code.
Both	Data type		Used to specify the data type for an `mpt.Signal` data object, but not for an `mpt.Parameter` data object. The data type for an `mpt.Parameter` data object is specified in the **Value** field above. See "Working with Data Types" in the Simulink documentation.
Both	Units	`*null`	Units of measurement of the signal or parameter. (Enter text in this field.)
Both	Dimensions	`*-1`	The dimension of the signal or parameter. For a parameter, the dimension is derived from its value.

Both	Complexity	`*auto`
		`real`
		`complex`
			Complexity specifies whether the signal or parameter is a real or complex number. Select `auto` for the code generator to decide. For a parameter, the complexity is derived from its value.

Signal	Sample time	`*-1`	Model or block execution rate.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Signal	Sample mode	`*auto`	Determines how the signal propagates through the model. Select `auto` for the code generator to decide.
		`Sample based`	The signal propagates through the model one sample at a time.
		`Frame based`	The signal propagates through the model in batches of samples.
Both	Minimum	`*0.0`	The minimum value to which the parameter or signal is expected to be bound.
		Any number within the minimum range of the parameter or signal. (Based on the data type and resolution of the parameter or signal.)	
Both	Maximum	`*0.0`	Maximum value to which the parameter or signal is expected to be bound. (Enter information using a dialog box.)
	Code generation options		
	Storage class		Note that an `auto` selection for a storage class tells the build process to decide how to declare and store the selected parameter or signal.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Both	Default (Custom)		Code generation decides how to declare the data object.
Both	Global (Custom)	`Global (Custom)` is the default storage class for `mpt` data objects.	Ensures that the code generator places no qualifier in the data object's declaration.
Both	Memory section	`*Default`	**Memory section** allows you to specify storage directives for the data object. `Default` ensures that the code generator places no type qualifier and no `pragma` statement with the data object's declaration.
Parameter		`MemConst`	Places the `const` type qualifier in the declaration.
Both		`MemVolatile`	Places the `volatile` type qualifier in the declaration.
Parameter		`MemConstVolatile`	Places the `const volatile` type qualifier in the declaration.
Both	Header file		Name of the file used to import or export the data object. This file contains the declaration (`extern`) to the data object. Also, you can specify this header filename between the double-quotation or angle-bracket delimiter. You can specify the delimiter with or without the `.h` extension. For example, `"object.h"` or `"object"` has the same effect. For the selected data object, this overrides the general delimiter selection in the

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
			#include file delimiter field on the Configuration Parameters dialog box.
Both	Owner	*Blank	The name of the module that owns this signal or parameter. This is used to help determine the ownership of a definition. For details, see "Ownership Settings" on page 13-10 and "Effects of Ownership Settings" on page 13-22.
Both	Definition file	*Blank	Name of the file that defines the data object.
		Any valid text string	
Both	Persistence level		The number you specify is relative to **Signal display level** or **Parameter tune level** on the **Code 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 "Code Generation Pane: Code Placement".
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 and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Parameter	Const (Custom)		Places the `const` type qualifier in the declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Both	Volatile (Custom)		Places the `volatile` type qualifier in the declaration.
Both	Header file		See above.
Both	Owner		See above.
Both	Definition file		See above.
Both	Persistence level		See above.
Parameter	ConstVolatile (Custom)		Places the `const volatile` type qualifier in declaration.
Parameter	Header file		See above.
Parameter	Owner		See above.
Parameter	Definition file		See above.
Parameter	Persistence level		See above.
Parameter	Define (Custom)		Represents parameters with a `#define` macro.
Parameter	Header file		See above.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
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.
Both	Data access	`*Direct`	Allows you to specify whether the identifier that corresponds to the selected data object stores data of a data type (`Direct`) or stores the address of the data (a pointer).
Both		`Pointer`	If you select `Pointer`, the code generator places `*` before the identifier in the generated code.
	Header file		See above.
Both	Struct (Custom)		Embeds data in a named `struct` to encapsulate sets of data.
Both	Struct name		See above.
Signal	GetSet (Custom)		Reads (gets) and writes (sets) data using functions.

Parameter and Signal Property Values (Continued)

Class: Parameter, Signal, or Both	Property	Available Property Values (* Indicates Default)	Description
Signal	Header file		See above.
Signal	Get function		Specify the Get function.
Signal	Set function		Specify the Set function.
Both	Alias	`*null`	As explained in detail in “Applying Naming Rules to Identifiers Globally” on page 12-30, for a Simulink or `mpt` data object (identifier), specifying a name in the **Alias** field overrides the global naming rule selection you make on the Configuration Parameters dialog box.
		Any valid ANSI®[1] C/C++ variable name	
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	

1. ANSI® is a registered trademark of the American National Standards Institute, Inc.

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

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 1: Parameter data objects can be declared or defined as constants. I know that the data object `GAIN` is a parameter. I want this to be declared or defined in the `.c` file as a variable. But I notice that `GAIN` is declared as a constant by the statement `const real_T GAIN = 5.0;`. Also, this statement is in the constant section of the file.	In the Model Explorer, I clicked the data object `GAIN`. I noticed that the property value for its **Memory section** property is set at `MemConst`. I changed this to `Default`.	I notice two differences. One is that now `GAIN` is declared as a variable with the statement `real_T GAIN = 5.0;`. The second difference is that the declaration now is located in the `MemConst` memory section in the `.c` or `.cpp` file.
Example 2: I notice again the declaration of `GAIN` in the `.c` file mentioned in Example 1. It appears as `real_T GAIN = 5.0;`. But I have changed my mind. I want data object `GAIN` to be `#define`.	I changed the **Storage class** selection to `Define (Custom)`.	`GAIN` is no longer declared in the `.c` file as a `MemConst` parameter. Rather, it now is defined as a `#define` macro by the code `#define GAIN 5.0`, and this is located near the top of the `.c` file with the other preprocessor directives.

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

What I noticed when inspecting the .c/.cpp file	Change I made to property value settings	What I noticed after regenerating and reinspecting the file
Example 3: I changed my mind again after doing Example 2. I do want `GAIN` defined using the `#define` preprocessor directive. But I do not want to include the `#define` in this file. I know it exists in another file and I want to reference that file.	On the Model Explorer, I notice that the property value for the **Header file** property is blank. I changed this to `filename.h`. (I chose the ANSI C/C++ double quote mechanism for the `#include`, but could have chosen the angle bracket mechanism.) Also, it is necessary that I make the user-defined `filename.h` available to the 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` file with an alias.	In the Model Explorer, I selected the data object `data_in`. I noticed that the **Alias** field is blank. I changed this to `data_in_alias`, which I know is a valid ANSI C/C++ variable name.	The identifier `data_in_alias` now appears in the `.c` file everywhere `data_in` appeared.

8

Creating and Using Custom Storage Classes

- “Introduction to Custom Storage Classes” on page 8-2
- “Resources for Defining Custom Storage Classes” on page 8-5
- “Simulink Package Custom Storage Classes” on page 8-6
- “Creating Packages that Support CSC Definitions” on page 8-8
- “Designing Custom Storage Classes and Memory Sections” on page 8-12
- “Applying CSCs to Parameters and Signals” on page 8-37
- “Generating Code with Custom Storage Classes” on page 8-58
- “Defining Advanced Custom Storage Class Types” on page 8-62
- “GetSet Custom Storage Class for Data Store Memory” on page 8-66
- “Custom Storage Class Implementation” on page 8-70
- “Custom Storage Class Limitations” on page 8-72

Introduction to Custom Storage Classes

In this section...

"Custom Storage Class Memory Sections" on page 8-3

"Registering Custom Storage Classes" on page 8-3

"Custom Storage Class Demos" on page 8-4

During the build process, the *storage class* specification of a signal, tunable parameter, block state, or data object specifies how that entity is declared, stored, and represented in generated code. Note that in the context of the build process, the term "storage class" is not synonymous with the term "storage class specifier", as used in the C language.

The Simulink Coder software defines four built-in storage classes for use with all targets: `Auto`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer`. These storage classes provide limited control over the form of the code generated for references to the data. For example, data of storage class `Auto` is typically declared and accessed as an element of a structure, while data of storage class `ExportedGlobal` is declared and accessed as unstructured global variables. For information about built-in storage classes, see "Signal Considerations" and "Defining Data Representation and Storage for Code Generation" in the Simulink Coder documentation.

The built-in storage classes are suitable for many applications, but embedded system designers often require greater control over the representation of data. Embedded Coder *custom storage classes* (CSCs) extend the built-in storage classes provided by the Simulink Coder software. CSCs can provide application-specific control over the constructs required to represent data in an embedded algorithm. For example, you can use CSCs to:

- Define structures for storage of parameter or signal data.
- Conserve memory by storing Boolean data in bit fields.
- Integrate generated code with legacy software whose interfaces cannot be modified.

- Generate data structures and definitions that comply with your organization's software engineering guidelines for safety-critical code.

Custom storage classes affect only code generated for ERT targets. When **Configuration Parameters > Code Generation > Target Selection > System target file** specifies a GRT target, the names of custom storage classes sometimes appear in dialog boxes, but selecting a CSC is functionally the same as selecting `Auto`. See "Selecting and Configuring a Target" for information about ERT and GRT targets.

Custom Storage Class Memory Sections

Every custom storage class has an associated *memory section* definition. A memory section is a named collection of properties related to placement of an object in memory; for example, in RAM, ROM, or flash memory. Memory section properties let you specify storage directives for data objects. For example, you can specify `const` declarations, or compiler-specific `#pragma` statements for allocation of storage in ROM or flash memory sections.

See "Creating and Editing Memory Section Definitions" on page 8-31 for details about using the Custom Storage Class designer to define memory sections. While memory sections are often used with data in custom storage classes, they can also be used with various other constructs. See Chapter 9, "Memory Sections" for more information about using memory sections with custom storage classes, and complete information about using memory sections with other constructs.

Registering Custom Storage Classes

CSCs are associated with Simulink data class packages (such as the `Simulink` package) and with classes within packages (such as the `Simulink.Parameter` and `Simulink.Signal` classes). The custom storage classes associated with a package are defined by a *CSC registration file*. For example, a CSC registration file exists for the `Simulink` package. This registration file provides predefined CSCs for use with the `Simulink.Signal` and `Simulink.Parameter` classes, and with subclasses derived from these classes. The predefined CSCs are sufficient for a wide variety of applications.

If you use only predefined CSCs, you do not need to be concerned with CSC registration files. You cannot add or change CSCs associated with built-in

packages and classes, but you can create your own packages and subclasses, then associate CSCs with those. See "Custom Storage Class Implementation" on page 8-70 for more information.

Custom Storage Class Demos

Three demos are available that show Custom Storage Class capabilities:

`rtwdemo_cscpredef` — Shows predefined custom storage classes and embedded signal objects

`rtwdemo_importstruct` — Shows custom storage classes used to access imported data efficiently

`rtwdemo_advsc` — Shows how custom storage classes can support data dictionary driven modeling

To launch a demo, click the demo's name above, or type its name in the MATLAB Command Window.

Resources for Defining Custom Storage Classes

The resources for working with custom storage class definitions are:

- The Simulink Data Class Designer, which you can use to create a data object package and enable the ability to define your own CSC definitions for classes contained in the package. For information about the Data Class Designer, see "Subclassing Simulink Data Classes" and "Creating Packages that Support CSC Definitions" on page 8-8.
- A set of ready-to-use CSCs. These CSCs are designed to be useful in code generation for embedded systems development. CSC functionality is integrated into the `Simulink.Signal` and `Simulink.Parameter` classes; you do not need to use special object classes to generate code with CSCs. If you are unfamiliar with the `Simulink.Signal` and `Simulink.Parameter` classes and objects, read the "Defining Data Representation and Storage for Code Generation" section in the Simulink Coder documentation.
- The Custom Storage Class Designer (`cscdesigner`) tool, which is described in this chapter. This tool lets you define CSCs that are tailored to your code generation requirements. The Custom Storage Class Designer provides a graphical user interface that you can use to implement CSCs. You can use your CSCs in code generation immediately, without any Target Language Compiler (TLC) or other programming. See "Designing Custom Storage Classes and Memory Sections" on page 8-12 for details.

Simulink Package Custom Storage Classes

The Simulink package includes a set of built-in custom storage classes. These are categorized as custom storage classes, even though they are built-in, because they:

- Extend the storage classes provided by the Simulink Coder software
- Are functionally the same as if you had defined them yourself using the CSC Designer

MathWorks provides the built-in CSCs because they provide specifications that many users need. Otherwise, all such users would have to define the needed CSCs manually, causing a needless duplication of effort.

You cannot change the CSCs built into the Simulink package, but you can subclass the package and add CSCs to the subclass, following the steps in "Resources for Defining Custom Storage Classes" on page 8-5.

Some CSCs in the Simulink package are valid for parameter objects but not signal objects and vice versa. For example, you can assign the storage class Const to a parameter but not to a signal, because signal data is not constant. The next table defines the CSCs built into the Simulink package and shows where each of the CSCs can be used.

CSC Name	Purpose	Signals?	Parameters?
BitField	Generate a struct declaration that embeds Boolean data in named bit fields.	Y	Y
CompilerFlag	Supports preprocessor conditionals defined via compiler flag. See "Generating Code for Variant Systems" on page 4-2.	N	Y
Const	Generate a constant declaration with the const type qualifier.	N	Y
ConstVolatile	Generate declaration of volatile constant with the const volatile type qualifier.	N	Y

CSC Name	Purpose	Signals?	Parameters?
`Default`	`Default` is a placeholder CSC that the code generator assigns to the `RTWInfo.CustomStorageClass` property of signal and parameter objects when they are created. You cannot edit the `Default` CSC definition.	Y	Y
`Define`	Generate `#define` directive.	N	Y
`ExportToFile`	Generate header (`.h`) file, with user-specified name, containing global variable declarations.	Y	Y
`GetSet`	Supports specialized function calls to read and write the memory associated with a Data Store Memory block. See "GetSet Custom Storage Class for Data Store Memory" on page 8-66.	Y	Y
`ImportedDefine`	Supports preprocessor conditionals defined via legacy header file. See "Generating Code for Variant Systems" on page 4-2.	N	Y
`ImportFromFile`	Generate directives to include predefined header files containing global variable declarations.	Y	Y
`Struct`	Generate a `struct` declaration encapsulating parameter or signal object data.	Y	Y
`Volatile`	Use `volatile` type qualifier in declaration.	Y	Y

Creating Packages that Support CSC Definitions

You can create a package and associate your own CSC definitions with classes contained in the package. You do this by creating a data object package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. The procedure below shows how to create and configure such a package. For additional information, see "Subclassing Simulink Data Classes".

1. Open the Simulink Data Class Designer by choosing **Tools > Data Class Designer** in the model window, or typing the following at the MATLAB command prompt:

   ```
   sldataclassdesigner
   ```

2. The Data Class Designer loads all packages that exist on the MATLAB path.

3. To create a new package, click **New** next to the **Package name** field. If desired, edit the **Package name**. Then, click **OK**.

4. In the **Parent directory** field, enter the path to the directory where you want to store the new package.

   Note Do not create class package directories under *`matlabroot`*. Packages in these directories are treated as built-in and will not be visible in the Data Class Designer.

   ---

5. Click on the **Classes** tab.

6. Create a new class by clicking **New** next to the **Class name** field. If desired, edit the **Class name**. Then, click **OK**.

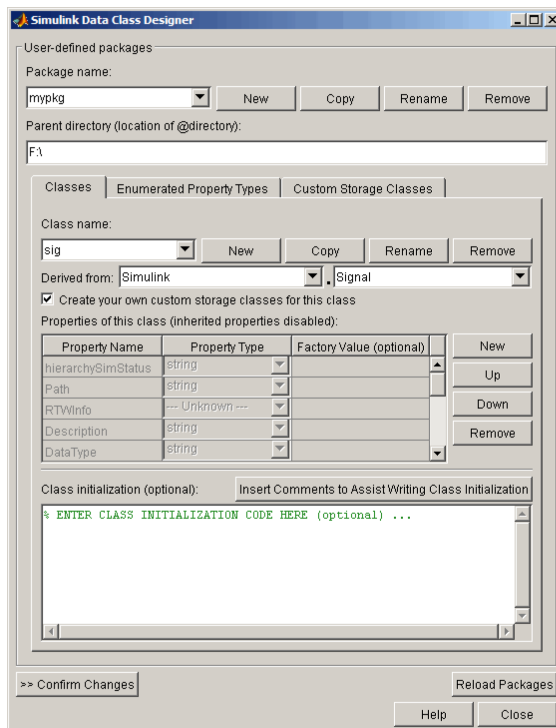
7. In the **Derived from** menus, select `Simulink.Signal` or `Simulink.Parameter`.

8. The **Create your own custom storage classes for this class** option is now enabled. This option is enabled when the selected class is derived from `Simulink.Signal` or `Simulink.Parameter`. You must select this option to create CSCs for the new class. If the **Create your own custom storage**

classes for this class option is not selected, the new class inherits the CSCs of the parent class.

Note To create a CSC registration file for a package, the **Create your own custom storage classes for this class** option must be selected for at least one of the classes in the package.

In the figure below, a new package called `mypkg` has been created. This package contains a new class, derived from `Simulink.Signal`, called `sig`. The **Create your own custom storage classes for this class** option is selected.



Complete instructions for using the Data Class Designer appear in “Subclassing Simulink Data Classes” in the Simulink documentation. See

also the instructions that appear when you click the **Custom Storage Classes** tab.

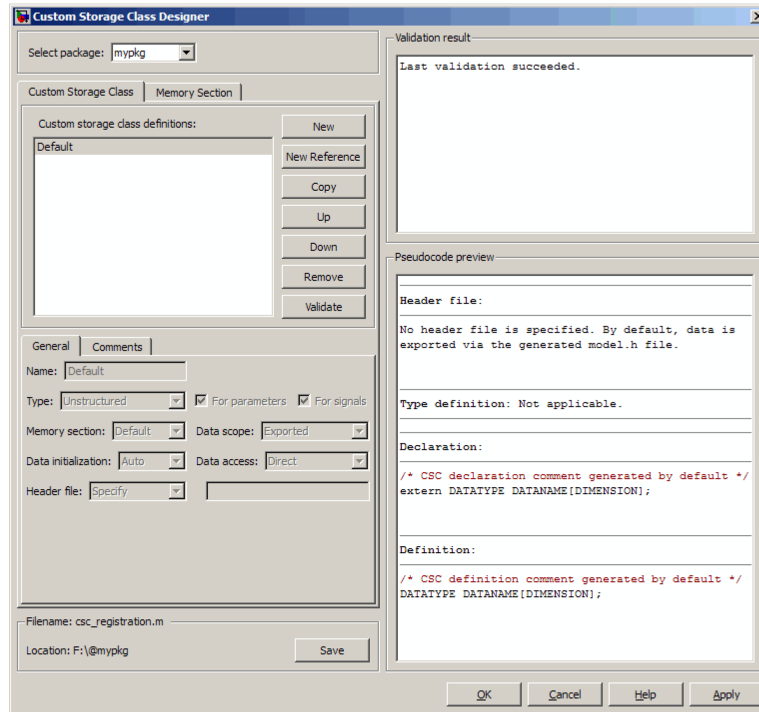
9. If desired, repeat steps 6–8 to add other derived classes to the package and associate CSCs with them.

10. Click **Confirm Changes**. In the **Confirm Changes** pane, select the package you created. Add the parent directory to the MATLAB path if necessary. Then, click **Write Selected**.

    The package directories and files, including the CSC registration file, are written out to the parent directory.

11. Click **Close**.

12. You can now view and edit the CSCs belonging to your package in the Custom Storage Class Designer, which you open with the MATLAB command `cscdesigner`. Initially, the package contains only the `Default` CSC definition, as shown in the figure below.



13 Add and edit your CSC and memory section definitions, as described in "Designing Custom Storage Classes and Memory Sections" on page 8-12. After you have created CSC definitions for your package, you can instantiate objects of the classes belonging to your package, and assign CSCs to them.

You need to restart your MATLAB session before you can use the new CSCs with objects of your new classes.

Designing Custom Storage Classes and Memory Sections

In this section...

"Using the Custom Storage Class Designer" on page 8-12
"Editing Custom Storage Class Properties" on page 8-19
"Using Custom Storage Class References" on page 8-26
"Creating and Editing Memory Section Definitions" on page 8-31
"Using Memory Section References" on page 8-34

Using the Custom Storage Class Designer

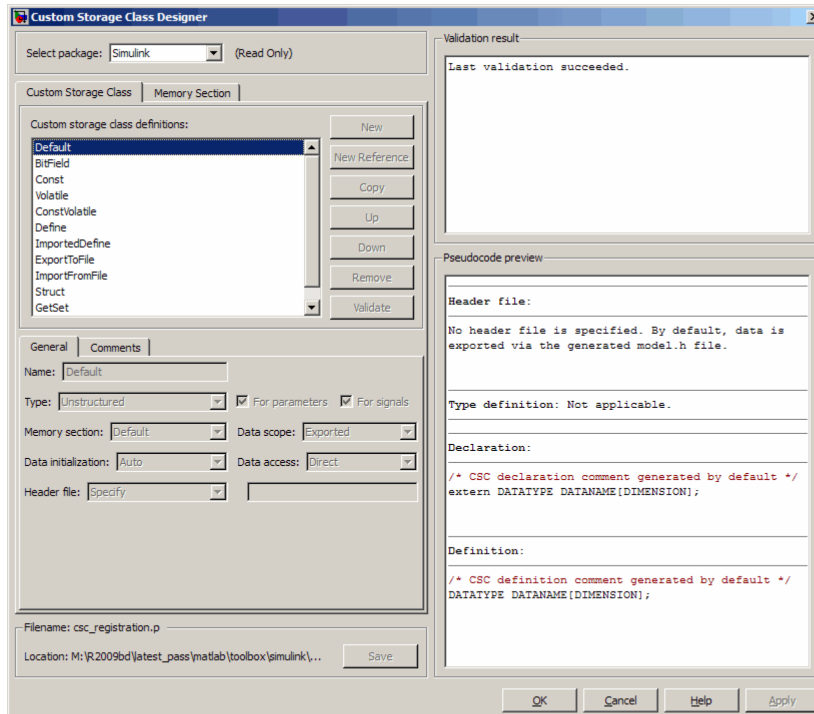
The Custom Storage Class Designer (`cscdesigner`) is a tool for creating and managing custom storage classes and memory sections. You can use the Custom Storage Class Designer to:

- Load existing custom storage classes and memory sections and view and edit their properties
- Create new custom storage classes and memory sections
- Create references to custom storage classes and memory sections defined in other packages
- Copy and modify existing custom storage class and memory section definitions
- Verify the correctness and consistency of custom storage class and memory section definitions
- Preview pseudocode generated from custom storage class and memory section definitions
- Save custom storage class and memory section definitions

To open the Custom Storage Class Designer, type the following command at the MATLAB prompt:

```
cscdesigner
```

When first opened, the Custom Storage Class Designer scans all data class packages on the MATLAB path to detect packages that have a CSC registration file. A message is displayed while scanning proceeds. When the scan is complete, the Custom Storage Class Designer window appears:



The Custom Storage Class Designer window is divided into several panels:

- **Select package**: Lets you select from a menu of data class packages that have CSC definitions associated with them. See "Selecting a Data Class Package" on page 8-14 for details.
- **Custom Storage Class / Memory Section** properties: Lets you select, view, edit, copy, verify, and perform other operations on CSC definitions or memory section definitions. The common controls in the **Custom Storage Class / Memory Section** properties panel are described in "Selecting and Editing CSCs, Memory Sections, and References" on page 8-15.

- When the **Custom Storage Class** tab is selected, you can select a CSC definition or reference from a list and edit its properties. See "Editing Custom Storage Class Properties" on page 8-19 for details.
- When the **Memory Section** tab is selected, you can select a memory section definition or reference from a list and edit its properties. See "Creating and Editing Memory Section Definitions" on page 8-31 for details.

- **Filename**: Displays the filename and location of the current CSC registration file, and lets you save your CSC definition to that file. See "Saving Your Definitions" on page 8-18 for details.
- **Pseudocode preview**: Displays a preview of code that is generated from objects of the given class. The preview is pseudocode, since the actual symbolic representation of data objects is not available until code generation time. See "Previewing Generated Code" on page 8-33 for details.
- **Validation result**: Displays any errors encountered when the currently selected CSC definition is validated. See "Validating CSC Definitions" on page 8-25 for details.

Selecting a Data Class Package

A CSC or memory section definition or reference is uniquely associated with a Simulink data class package. The link between the definition/reference and the package is formed when a CSC registration file (`csc_registration.m`) is located in the package directory.

You never need to search for or edit a CSC registration file directly: the Custom Storage Class Designer locates all available CSC registration files. The **Select package** menu contains names of all data class packages that have a CSC registration file on the MATLAB search path. At least one such package, the `Simulink` package, is always present.

When you select a package, the CSCs and memory section definitions belonging to the package are loaded into memory and their names are displayed in the scrolling list in the **Custom storage class** panel. The name and location of the CSC registration file for the package is displayed in the **Filename** panel.

If you select a user-defined package, by default you can use the Custom Storage Class Designer to edit its custom storage classes and memory sections. If you select a built-in package, you cannot edit its custom storage classes or memory sections. See “Custom Storage Class Implementation” on page 8-70 for more information.

Selecting and Editing CSCs, Memory Sections, and References

The **Custom Storage Class / Memory Section** panel lets you select, view, and (if the CSC is writable) edit CSC and memory section definitions and references. In the next figure and the subsequent examples, the selected package is `mypkg`. Instructions for creating a user-defined package like `mypkg` appear in “Creating Packages that Support CSC Definitions” on page 8-8.

The list at the top of the panel displays the definitions/references for the currently selected package. To select a definition/reference for viewing and editing, click on the desired list entry. The properties of the selected definition/reference appear in the area below the list. The number and type of properties vary for different types of CSC and memory section definitions. See:

- "Editing Custom Storage Class Properties" on page 8-19 for information about the properties of the predefined CSCs.
- "Creating and Editing Memory Section Definitions" on page 8-31 for information about the properties of the predefined memory section definitions.

The buttons to the right of the list perform these functions, which are common to both custom storage classes and memory definitions:

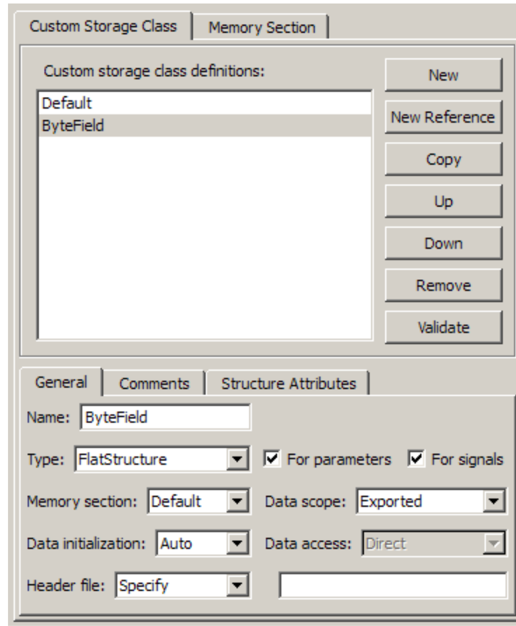
- **New**: Creates a new CSC or memory section with default values.
- **New Reference:** Creates a reference to a CSC or memory section definition in another package. The default initially has a default name and properties. See "Using Custom Storage Class References" on page 8-26 and "Using Memory Section References" on page 8-34.
- **Copy**: Creates a copy of the selected definition / reference. The copy initially has a default name using the convention:

  ```
  definition_name_n
  ```

  where `definition_name` is the name of the original definition, and `n` is an integer indicating successive copy numbers (for example: `BitField_1`, `BitField_2`, ...)
- **Up**: Moves the selected definition one position up in the list.
- **Down**: Moves the selected definition one position down in the list
- **Remove**: Removes the selected definition from the list.
- **Validate**: Performs a consistency check on the currently selected definition. Errors are reported in the **Validation result** panel.

For example, if you click **New**, a new custom storage class is created with a default name:

You can now rename the new class by typing the desired name into the **Name** field, and set other fields as needed. The changes take effect when you click **Apply** or **OK**. For example, you could set values for the new custom storage class as follows:



Saving Your Definitions

After you have created or edited a CSC or memory section definition or reference, you must save the changes to the CSC registration file. To do this, click **Save** in the **Filename** panel. When you click **Save**, the current CSC and memory section definitions that are in memory are validated, and the definitions are written out.

If errors occur, they are reported in the **Validation result** panel. The definitions are saved whether or not errors exist. However, you should correct any validation errors and resave your definitions. Trying to use definitions that were saved with validation errors can cause additional errors. Such problems can occur even it you do not try to use the specific parts of the definition that contain the validation errors, making the problems difficult to diagnose.

Restarting MATLAB After Changing Definitions

If you add, change, or delete custom storage class or memory section definitions for any user-defined class, and objects of that class already exist,

you must restart MATLAB to put the changed definitions into effect and eliminate obsolete objects. A message warning you to restart MATLAB appears when you save the changed definitions. This warning message does not affect the success of the save operation itself.

Editing Custom Storage Class Properties

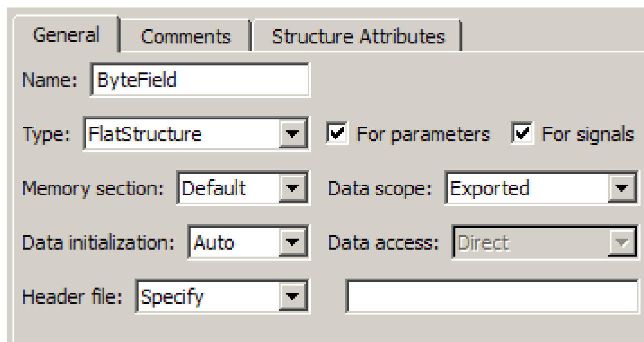
To view and edit the properties of a CSC, click the **Custom Storage Class** tab in the **Custom Storage Class / Memory Section** panel. Then, select a CSC name from the **Custom storage class definitions** list.

The CSC properties are divided into several categories, selected by tabs. Selecting a class, and setting property values for that class, can change the available tabs, properties, and values. As you change property values, the effect on the generated code is immediately displayed in the **Pseudocode preview** panel. In most cases, you can define your CSCs quickly and easily by selecting the **Pseudocode preview** panel and using the **Validate** button frequently.

The property categories and corresponding tabs are as follows:

General

Properties in the **General** category are common to all CSCs. In the next figure and the subsequent examples, the selected custom storage class is `ByteField`. Instructions for creating a user-defined custom storage class like `ByteField` appear in "Selecting and Editing CSCs, Memory Sections, and References" on page 8-15.



Properties in the **General** category, and the possible values for each property, are as follows:

- **Name**: The CSC name, selected from the **Custom storage class definitions** list. The name cannot be any TLC keyword. Violating this rule causes an error.
- **Type**: Specifies how objects of this class are stored. Values:
  - `Unstructured`: Objects of this class generate unstructured storage declarations (for example, scalar or array variables), for example:

    ```
    datatype dataname[dimension];
    ```

  - `FlatStructure`: Objects of this class are stored as members of a struct. A **Structure Attributes** tab is also displayed, allowing you to specify additional properties such as the struct name. See "Structure Attributes" on page 8-23.
  - `Other`: Used for certain data layouts, such as nested structures, that cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. If you want to generate other types of data, you can create a new custom storage class from scratch by writing the necessary TLC code. See "Defining Advanced Custom Storage Class Types" on page 8-62 for more information.
- **For parameters** and **For signals**: These options let you enable a CSC for use with only certain classes of data objects. For example, it does not make sense to assign storage class `Const` to a `Simulink.Signal` object. Accordingly, the **For signals** option for the `Const` class is deselected, while the **For parameters** is selected.
- **Memory section**: Selects one of the memory sections defined in the **Memory Section** panel. See "Creating and Editing Memory Section Definitions" on page 8-31.
- **Data scope**: Controls the scope of symbols generated for data objects of this class. Values:
  - `Auto`: Symbol scope is determined internally by code generation. If possible, symbols have `File` scope. Otherwise, they have `Exported` scope.

  - `Exported`: Symbols are exported to external code in the header file specified by the **Header File** field. If no **Header File** is specified, symbols are exported to external code in *`model`*`.h`.
  - `Imported`: Symbols are imported from external code in the header file specified by the **Header File** field. If you do not specify a header file, an `extern` directive is generated in *`model`*`_private.h`. For imported data, if the **Data initialization** value is `Macro`, a header file *must* be specified.
  - `File`: The scope of each symbol is the file that defines it. File scope requires each symbol to be used in a single file. If the same symbol is referenced in multiple files, an error occurs at code generation time.
  - `Instance specific`: Symbol scope is defined by the **Data scope** field of the `RTWInfo.CustomAttributes` property of each data object.

- **Data initialization**: Controls how storage is initialized in generated code. Values:
  - `Auto`: Storage initialization is determined internally by the code generation. Parameters have `Static` initialization, and signals have `Dynamic` initialization.
  - `None`: No initialization code is generated.
  - `Static`: A static initializer of the following form is generated:

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

  - `Dynamic`: Variable storage is initialized at runtime, in the *`model`*`_initialize` function.
  - `Macro`: A macro definition of the following form is generated:

    ```
    #define data numeric_value
    ```

    The `Macro` initialization option is available only for use with unstructured parameters. It is not available when the class is configured for generation of structured data, or for signals. If the **Data scope** value is `Imported`, a header file *must* be specified.
  - `Instance specific`: Initialization is defined by the **Data initialization** property of each data object.

Note When necessary, the code generator includes dynamic initialization code for signals and states even if the CSC has **Data initialization** set to `None` or `Static`.

- **Data access**: Controls whether imported symbols are declared as variables or pointers. This field is enabled only when **Data scope** is set to `Imported` or `Instance-specific`. Values:
  - `Direct`: Symbols are declared as simple variables, such as

    ```
    extern myType myVariable;
    ```

  - `Pointer`: Symbols are declared as pointer variables, such as

    ```
    extern myType *myVariable;
    ```

  - `Instance specific`: Data access is defined by the **Data access** property of each data object.
- **Header file**: Defines the name of a header file that contains exported or imported variable declarations for objects of this class. Values:
  - `Specify`: An edit field is displayed to the right of the property. This lets you specify a header file for exported or imported storage declarations. Specify the full filename, including the filename extension (such as `.h`). Use quotes or brackets as in C code to specify the location of the header file. Leave the edit field empty to specify no header file.
  - `Instance specific`: The header file for each data object is defined by the **Header file** property of the object. Leave the property undefined to specify no header file for that object.

  If the **Data scope** is `Exported`, specifying a header file is optional. If you specify a header file name, the custom storage class generates a header file containing the storage declarations to be exported. Otherwise, the storage declarations are exported in *`model`*`.h`.

  If the **Data scope** of the class is `Imported`, and **Data initialization** is `Macro`, you *must* specify a header file name. A `#include` directive for the header file is generated.

Comments. The **Comments** panel lets you specify comments to be generated with definitions and declarations.

Comments must conform to the ANSI C standard (`/*...*/`). Use `\n` to specify a new line.

Properties in the **Comments** tab are as follows:

- **Comment rules**: If `Specify` is selected, edit fields are displayed for entering comments. If `Default` is selected, comments are generated under control of the code generation software.
- **Type comment**: The comment entered in this field precedes the `typedef` or `struct` definition for structured data.
- **Declaration comment**: Comment that precedes the storage declaration.
- **Definition comment**: Comment that precedes the storage definition.

Structure Attributes

The **Structure Attributes** panel gives you detailed control over code generation for structs (including bitfields). The **Structure Attributes** tab

is displayed for CSCs whose **Type** parameter is set to `FlatStructure`. The following figure shows the **Structure Attributes** panel.

Structure Attributes Panel

The **Structure Attributes** properties are as follows:

- **Struct name**: If you select `Instance specific`, specify the struct name when configuring each instance of the class.

  If you select `Specify`, an edit field appears (as shown in Structure Attributes Panel on page 8-24) for entry of the name of the structure to be used in the `struct` definition. Edit fields **Type tag**, **Type token**, and **Type name** are also displayed.

- **Is typedef**: When this option is selected a `typedef` is generated for the struct definition, for example:

  ```
  typedef struct {
     ...
  } SignalDataStruct;
  ```

  Otherwise, a simple struct definition is generated.

- **Bit-pack booleans**: When this option is selected, signals and/or parameters that have Boolean data type are packed into bit fields in the generated struct.

- **Type tag**: Specifies a tag to be generated after the `struct` keyword in the struct definition.

- **Type token**: Some compilers support an additional token (which is simply another string) after the type tag. To generate such a token, enter the string in this field.

- **Type name**: Specifies the string to be used in `typedef` definitions. This field is visible if **Is typedef** is selected.

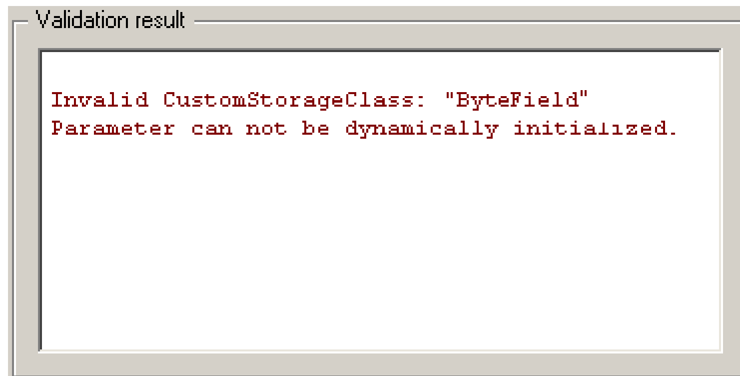
The following listing is the pseudocode preview corresponding to the **Structure Attributes** properties displayed in Structure Attributes Panel on page 8-24.

```
Header file:

No header file is specified. By default, data is
exported with the generated model.h file.


Type definition:

/* CSC type comment generated by default */

typedef struct aToken myTag {
   :
} myType;


Declaration:

/* CSC declaration comment generated by default */

extern myType MyStruct;


Definition:

/* CSC definition comment generated by default */

myType MyStruct = {...};
```

Validating CSC Definitions

To validate a CSC definition, select the definition on the **Custom Storage Class** panel and click **Validate**. The Custom Storage Class Designer then checks the definition for consistency. The **Validation result** panel displays

any errors encountered when the selected CSC definition is validated. The next figure shows the **Validation result** panel with a typical error message:



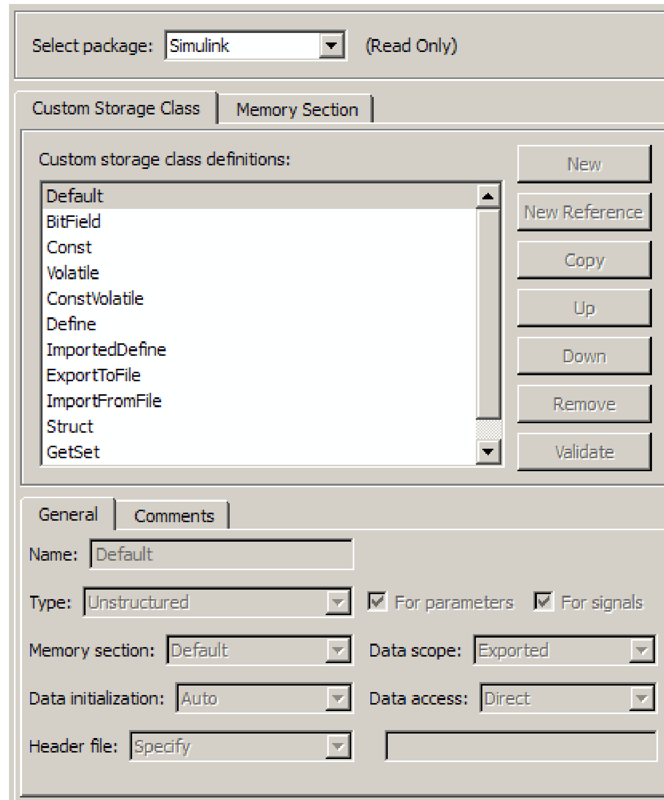
Validation is also performed whenever CSC definitions are saved. In this case, all CSC definitions are validated. (See "Saving Your Definitions" on page 8-18.)

Using Custom Storage Class References

Any package can access and use custom storage classes that are defined in any other package, including both user-defined packages and predefined packages such as `Simulink` and `mpt`. Only one copy of the storage class exists, in the package that first defined it. Other packages refer to it by pointing to it in its original location. Thus any changes to the class, including changes to a predefined class in later MathWorks® product releases, are immediately available in every referencing package.

To configure a package to use a custom storage class that is defined in another package:

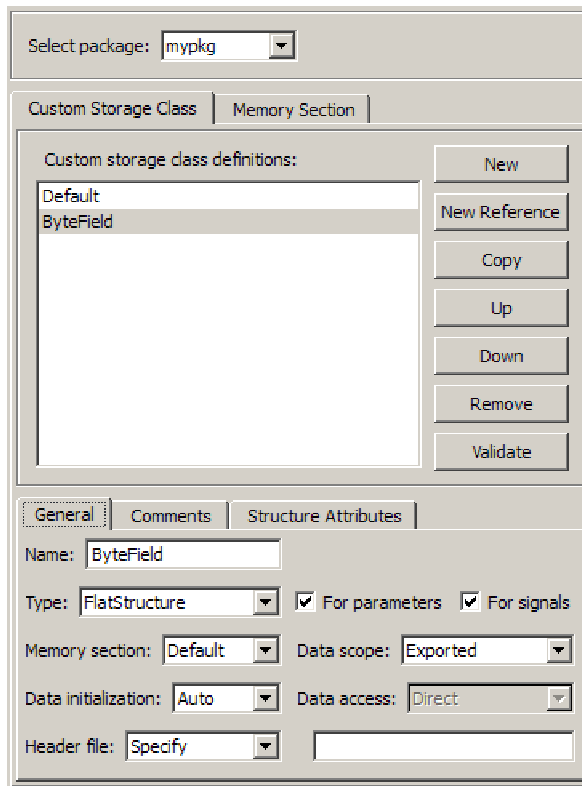
1 Type `cscdesigner` to launch the Custom Storage Class Designer. The relevant part of the designer window initially looks like this:



2 Select the **Custom Storage Class** tab.

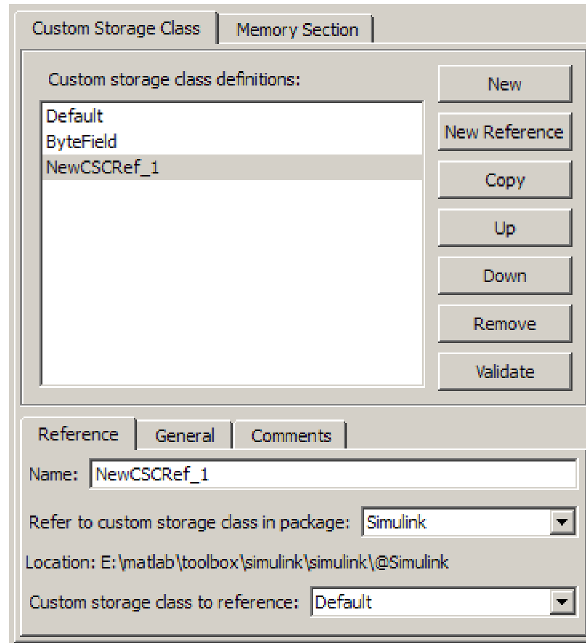
3 Use **Select Package** to select the package in which you want to reference a class or section defined in some other package. The selected package must be writable.

4 In the **Custom storage class definitions** pane, select the existing definition below which you want to insert the reference. For example:



5 Click **New Reference**.

A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties. A typical appearance is:



6 Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package. The name cannot be any TLC keyword. Violating this rule causes an error.

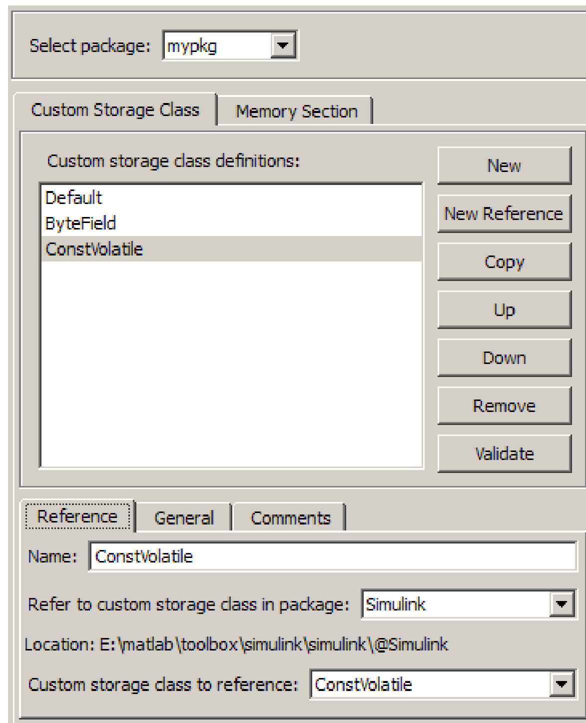
7 Set **Refer to custom storage class in package** to specify the package that contains the custom storage class you want to reference.

8 Set **Custom storage class to reference** to specify the custom storage class to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.

9 Click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 8-18 for information about saving changes permanently.

For example, the next figure shows the custom storage class `ConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name

that it has in the source package. Any other name could have been used without affecting the properties of the storage class.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage classes that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a custom storage class only in the package where it was originally defined.

Changing Existing CSC References

To change an existing CSC reference, select it in the **Custom storage class definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 8-18 for information about saving changes permanently.

Creating and Editing Memory Section Definitions

Memory section definitions add comments, qualifiers, and `#pragma` directives to generated symbol declarations. The **Memory Section** tab lets you create, view, edit, and verify memory section definitions. The steps for creating a memory section definition are essentially the same as for creating a custom storage class definition:

1. Select a writable package in the **Select package** field.

2. Select the **Memory Section** tab. In a new package, only a `Default` memory section initially appears.

3. Select the existing memory section below which you want to create a new memory section.

4. Click **New**.

   A new memory section definition with a default name appears below the selected memory section.

5. Set the name and other properties of the memory section as needed.

6. Click **OK** or **Apply**.

The next figure shows `mypkg` with a memory section called `MyMemSect`:

The **Memory section definitions** list lets you select a memory section definition to view or edit. The available memory section definitions also appear in the **Memory section name** menu in the **Custom Storage Class** panel. The properties of a memory section definition are as follows:

- **Memory section name**: Name of the memory section (displayed in **Memory section definitions** list).

- **Is const**: If selected, a `const` qualifier is added to the symbol declarations.
- **Is volatile**: If selected, a `volatile` qualifier is added to the symbol declarations.
- **Qualifier**: The string entered into this field is added to the symbol declarations as a further qualifier. Note that no verification is performed on this qualifier.
- **Memory section comment**: Comment inserted before declarations belonging to this memory section. Comments must conform to the ANSI C standard (`/*...*/`). Use `\n` to specify a new line.
- **Pragma surrounds**: Specifies whether the pragma should surround `All variables` or `Each variable`. When **Pragma surrounds** is set to `Each variable`, the `%<identifier>` token is allowed in pragmas and will be replaced by the variable or function name.
- **Pre-memory section pragma**: `pragma` directive that precedes the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.
- **Post-memory section pragma**: `pragma` directive that follows the storage definition of data belonging to this memory section. The directive must begin with `#pragma`.

Previewing Generated Code

If you click **Validate** on the **Memory Section** panel, the **Pseudocode preview** panel displays a preview of code that is generated from objects of the given class. The panel also displays messages (in blue) to highlight changes as they are made. The code preview changes dynamically as you edit the class properties. The next figure shows a code preview for the `MemConstVolatile` memory section.

Pseudocode preview

Header file: Not applicable.

Type definition: Not applicable.

Declaration:

```
extern const volatile DATATYPE DATANAME;
```

Definition:

```
/* ConstVolatile memory section */
const volatile DATATYPE DATANAME;
```

Using Memory Section References

Any package can access and use memory sections that are defined in any other package, including both user-defined packages and predefined packages such as `Simulink` and `mpt`. Only one copy of the section exists, in the package that first defined it; other packages refer to it by pointing to it in its original location. Thus any changes to the section, including changes to a predefined section in later MathWorks product releases, are immediately available in every referencing package.

To configure a package to use a memory section that is defined in another package:

1. Type `cscdesigner` to launch the Custom Storage Class Designer.
2. Select the **Memory Section** tab.
3. Use **Select Package** to select the package in which you want to reference a class or section defined in some other package.
4. In the **Memory section definitions** pane, select the existing definition below which you want to insert the reference.
5. Click **New Reference**.

A new reference with a default name and properties appears below the previously selected definition. The new reference is selected, and a **Reference** tab appears that shows the reference's initial properties.

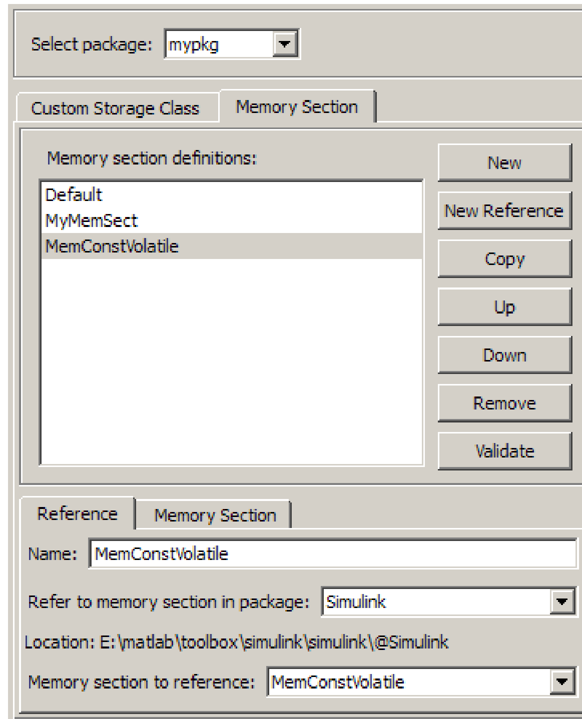
6. Use the **Name** field to enter a name for the new reference. The name must be unique in the importing package, but can duplicate the name in the source package.

7. Set **Refer to memory section in package** to specify the package that contains the memory section you want to reference.

8. Set **Memory section to reference** to specify the memory section to be referenced. Trying to create a circular reference generates an error and leaves the package unchanged.

9. Click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 8-18 for information about saving changes permanently.

For example, the next figure shows the memory section `MemConstVolatile` imported from the `Simulink` package into `mypkg`, and given the same name that it has in the source package. Any other name could have been used without affecting the properties of the memory section.



You can use Custom Storage Class Designer capabilities to copy, reorder, validate, and otherwise manage memory sections that have been added to a class by reference. However, you cannot change the underlying definitions. You can change a memory section only in the package where it was originally defined.

Changing Existing Memory Section References

To change an existing memory section reference, select it in the **Memory section definitions** pane. The **Reference** tab appears, showing the current properties of the reference. Make any needed changes, then click **OK** or **Apply** to save the changes to memory. See "Saving Your Definitions" on page 8-18 for information about saving changes permanently.

Applying CSCs to Parameters and Signals

In this section...

"About Applying Custom Storage Classes" on page 8-37
"Applying a Custom Storage Class to a Parameter" on page 8-38
"Applying a Custom Storage Class to a Signal" on page 8-40
"Applying a CSC Using a Base Workspace Signal Object" on page 8-41
"Applying a CSC Using an Embedded Signal Object" on page 8-43
"Specifying a Custom Storage Class Using the GUI" on page 8-50
"Specifying a Custom Storage Class Using the API" on page 8-53

About Applying Custom Storage Classes

You can apply a custom storage class to a parameter or a signal using the GUI or the API.

- To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace.
- To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object that is bound to the signal. You can provide this object in two ways:
  - Create the object in the base workspace, then bind it to the signal as described in "Resolving Symbols". When you save the model, you must save the object in a separate file, as with any base workspace object.
  - Use the Signal Properties dialog box to embed the object in the model on the port where the signal originates. When you save the model, Simulink automatically saves the embedded signal object as part of the model file.

Most of the GUI techniques, and most of the API techniques, are the same for parameter and signal objects, and for base workspace and embedded signal objects. Only the initial steps differ, after which you apply the same GUI or API instructions within the context that you established in the initial steps.

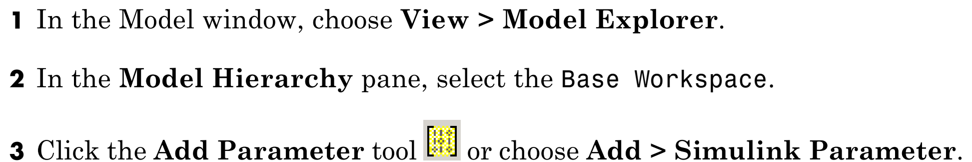
The following instructions assume that you have already created any needed packages, custom storage classes, and memory sections, as described in "Creating Packages that Support CSC Definitions" on page 8-8 and "Designing Custom Storage Classes and Memory Sections" on page 8-12.

Applying a Custom Storage Class to a Parameter

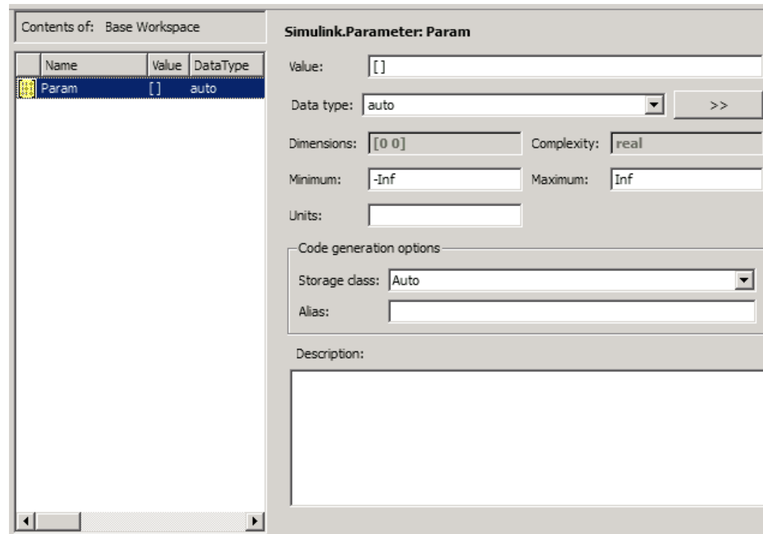
To apply a custom storage class to a parameter, you specify the storage class in the `Simulink.Parameter` object that defines the parameter in the base workspace. The instructions that begin in this section show you how to create that object using the GUI or API. Later instructions show you how to specify a custom storage class and custom attributes.

For information about using parameter objects to specify block parameter values, see "Working with Block Parameters" in the Simulink documentation. For information about parameter storage in generated code, see "Parameter Considerations" in the Simulink Coder documentation.

Providing a Parameter Object Using the GUI

1. In the Model window, choose **View > Model Explorer**.
2. In the **Model Hierarchy** pane, select the `Base Workspace`.
3. Click the **Add Parameter** tool  or choose **Add > Simulink Parameter**.

   Simulink creates a `Simulink.Parameter` object with a default name:



4. Change the parameter name as needed by editing it in the **Contents** pane. Example: `MyParam`.

5. Set parameter attributes other than **Code generation options** in the **Dialog** pane.

6. Follow the instructions in "Specifying a Custom Storage Class Using the GUI" on page 8-50.

Providing a Parameter Object Using the API

1. In the MATLAB Command Window, enter:

```
ParamName=ParamClass
```

   where *`ParamClass`* is `Simulink.Parameter` or any subclass of it that you have defined.

2. Simulink creates a *`ParamClass`* object with the specified name:

```
MyParam =

Simulink.Parameter (handle)
```

```
          Value: []
        RTWInfo: [1x1 Simulink.ParamRTWInfo]
    Description: ''
       DataType: 'auto'
            Min: -Inf
            Max: Inf
       DocUnits: ''
     Complexity: 'real'
     Dimensions: [0 0]
```

3 Set parameter attributes other than `RTWInfo`, which controls custom storage classes.

4 Follow the instructions in "Specifying a Custom Storage Class Using the API" on page 8-53.

Applying a Custom Storage Class to a Signal

To apply a custom storage class to a signal, you specify the storage class in a `Simulink.Signal` object. This object can exist in either of two locations:

- In the MATLAB base workspace
- On the port where the signal originates

The object itself is the same in either case; only its location and some of the techniques for managing it differ. The instructions that begin in this section show you how to create a signal object in either location using the GUI or API. Later instructions show you how to specify the custom storage class and custom attributes.

A given signal can be associated with at most one signal object under any circumstances. The signal can refer to the object more that once, but every reference must resolve to exactly the same object. A different signal object that has exactly the same properties will not meet the requirement for uniqueness. A compile-time error occurs if a model associates more than one signal object with any signal.

Assigning a signal to any non-`Auto` storage class automatically makes the signal a test point, overriding the setting of **Signal Properties > Logging**

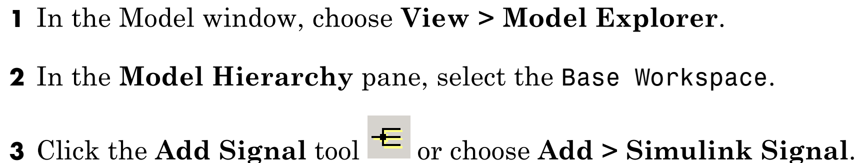
and accessibility > Test point. See “Working with Test Points”for more information.

For information about using signal objects to specify signal attributes, see “Working with Signals” in the Simulink documentation. For information about signal storage in generated code, see “Signal Considerations” in the Simulink Coder documentation.

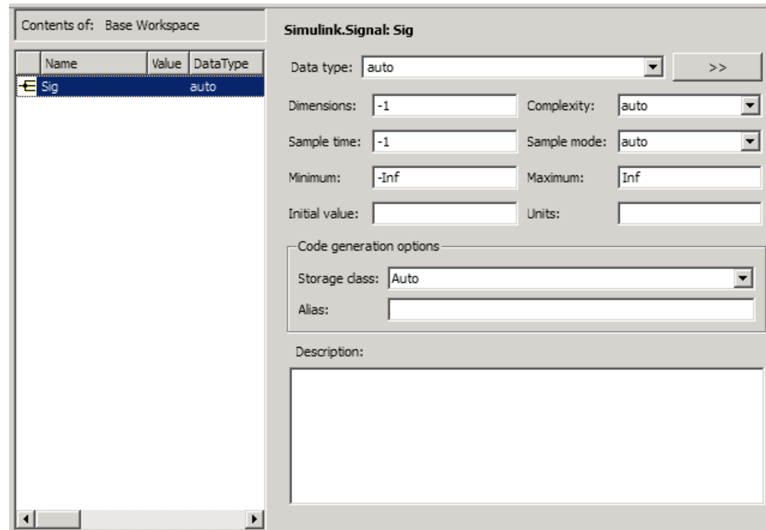
Applying a CSC Using a Base Workspace Signal Object

The first step is to create the signal object in the base workspace, after which you specify any needed signal attributes and the custom storage class and attributes.

Providing a Base Workspace Signal Object Using the GUI

1. In the Model window, choose **View > Model Explorer**.
2. In the **Model Hierarchy** pane, select the `Base Workspace`.
3. Click the **Add Signal** tool  or choose **Add > Simulink Signal**.

   Simulink creates a `Simulink.Signal` object with a default name:



4 Change the signal name as needed by editing it in the **Contents** pane. Example: `MySig`.

5 Set signal attributes other than **Code generation options** in the **Dialog** pane.

6 Give the signal the same name as the signal object, as described in "About Signal Names".

7 Arrange for the signal to resolve to the object, as described in "Resolving Symbols".

8 Follow the instructions in "Specifying a Custom Storage Class Using the GUI" on page 8-50.

Providing a Base Workspace Signal Object Using the API

1 In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

where *`SignalClass`* is `Simulink.Signal` or any subclass of it that you have defined.

2 Simulink creates a *`SignalClass`* object with the specified name:

```
MySig =

Simulink.Signal (handle)
         RTWInfo: [1x1 Simulink.SignalRTWInfo]
     Description: ''
        DataType: 'auto'
             Min: -Inf
             Max: Inf
        DocUnits: ''
      Dimensions: -1
      Complexity: 'auto'
      SampleTime: -1
    SamplingMode: 'auto'
    InitialValue: ''
```

3 Set parameter attributes other than `RTWInfo`, which controls custom storage classes.

4 Give the signal the same name as the signal object, as described in “About Signal Names”.

5 Arrange for the signal to resolve to the object, as described in “Resolving Symbols”.

6 Follow the instructions in “Specifying a Custom Storage Class Using the API” on page 8-53.

Applying a CSC Using an Embedded Signal Object

You can use the GUI or the API to apply a CSC using an embedded signal object.

- If you use the GUI, you use the Signal Properties dialog box to specify the attributes you want. The software then creates a `Simulink.Signal` object and assigns it to the output port where the signal originates.
- If you use the API, you instantiate `Simulink.Signal` or a subclass of it, set the attribute values that you want, and assign the object to the output port where the signal originates.

In either case, the effect on code generation is the same as if you had created a base workspace signal object that specified the same name, CSC, and custom attributes as the embedded signal object. See "Applying a CSC Using a Base Workspace Signal Object" on page 8-41 for details.

The advantages of using embedded signal objects are that they do not clutter the base workspace, and they do not need to be saved separately from the model, as base workspace objects do. When you save a model, Simulink saves any embedded signal objects in the model file, and reloads the objects when you later reload the model.

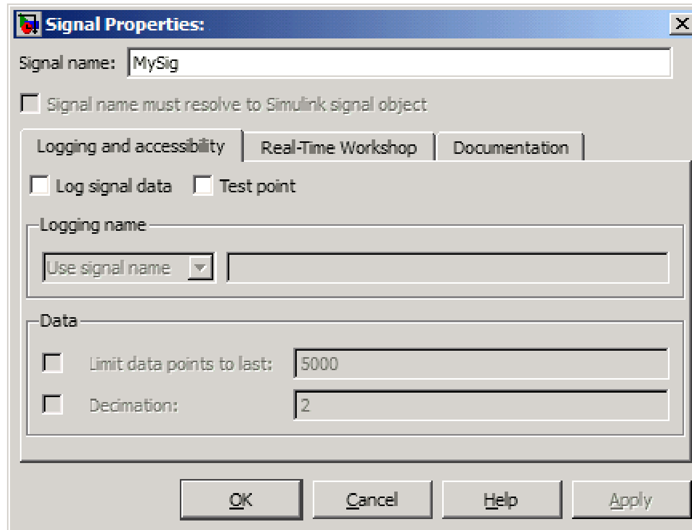
The disadvantage of embedded signal objects is that you can use such an object *only* to specify a custom storage class, custom attributes, and an alias; you must accept the default values for all other signal attributes. You cannot work around this restriction by providing additional information in a base workspace signal object on the same signal, because a signal object can have at most one associated signal object, as described in "Multiple Signal Objects".

Providing an Embedded Signal Object using the GUI

1 Give the signal a name, which must be a valid ANSI C identifier. Example: `MySig`.

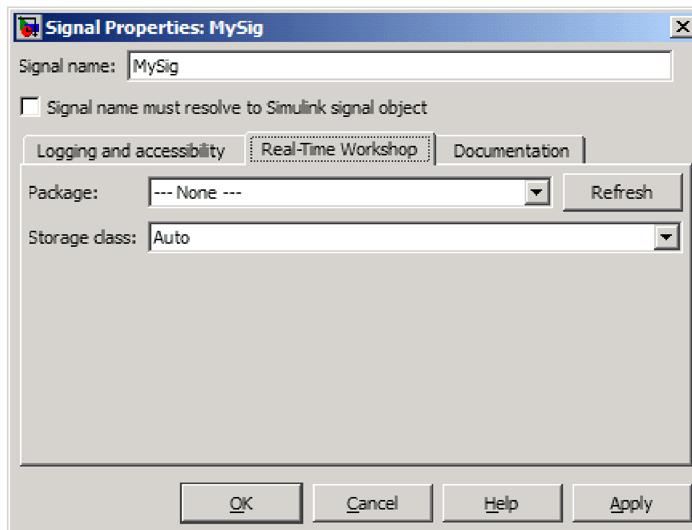
2 Right-click the signal and choose **Signal Properties** from the context menu.

   The Signal Properties dialog box opens:

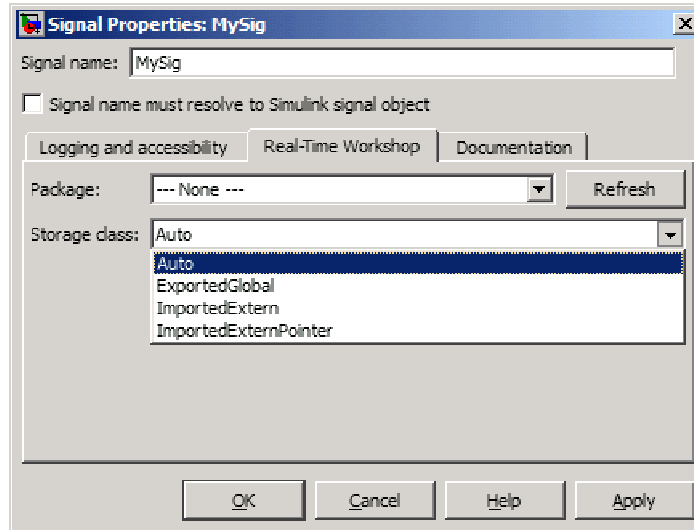


3 *Do not* select **Signal name must resolve to Simulink signal object**. Selecting it would require a base workspace signal object, which would conflict with the embedded signal object.

4 Click the **Code Generation** tab:

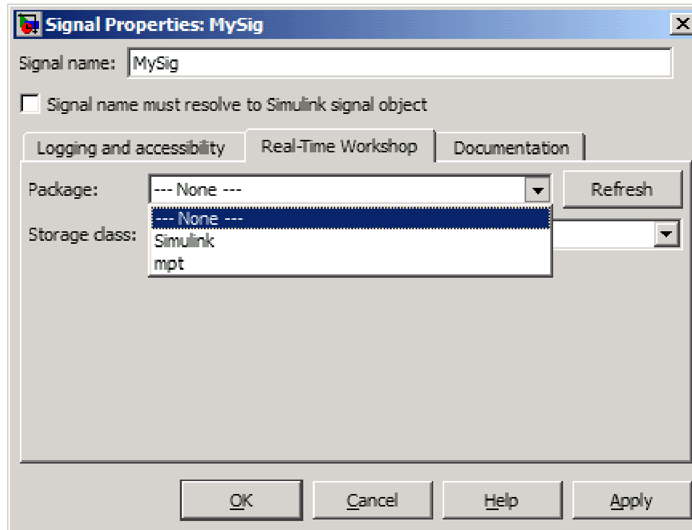


5 The **Package** is initially `---None---`. When no package is specified, only the non-custom built-in storage classes defined for both GRT and ERT targets are available:

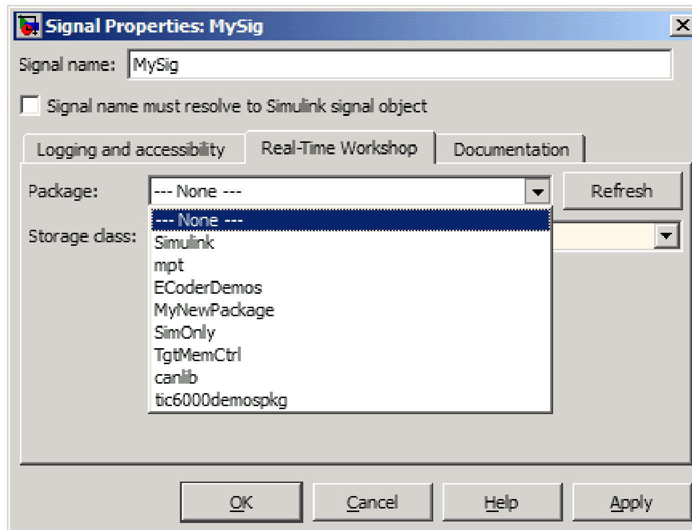


Applying a storage class when the package is `---None---` sets internal storage class attributes rather than creating an embedded signal object. For more information, see "Signal Considerations" and "Defining Data Representation and Storage for Code Generation" in the Simulink Coder documentation.

6 To apply a custom storage class, you must first specify the package where it is defined. Initially, viewing the **Package** menu displays only the built-in `Simulink` and `mpt` packages:

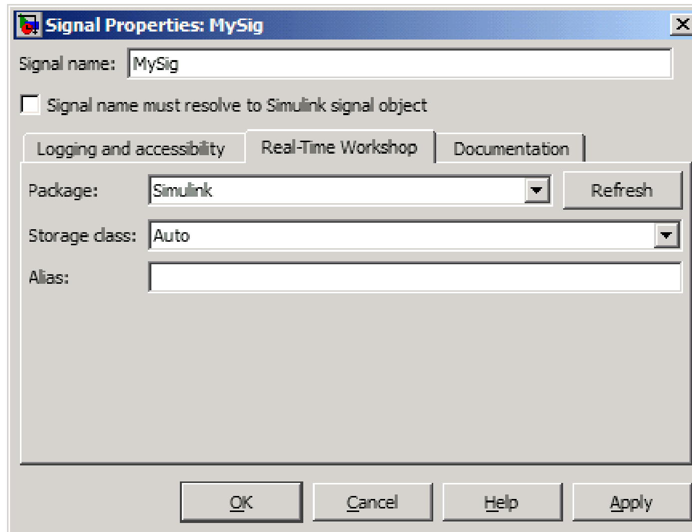


7 Click **Refresh** to load any other available packages, including user-defined packages, available on the MATLAB path. After a brief delay, a timer box tracks the progress of the package search. After the search completes, viewing the **Package** menu displays all available packages:



Once you have used **Refresh** in any Signal Properties dialog, Simulink saves the information for later use, so you do not have to click **Refresh** again during the current MATLAB session.

8. Select the package that contains the custom storage class you want to apply, e.g. `Simulink`:



9. Follow the instructions in "Specifying a Custom Storage Class Using the GUI" on page 8-50.

Deleting an Embedded Signal Object Using the GUI

To delete an embedded signal object with the GUI, delete the name of the signal to which the object applies, by editing the name in the graphical model or in the Signal Properties dialog box. Simulink automatically deletes the embedded signal object as soon as its signal has no name.

Providing an Embedded Signal Object using the API

To provide an embedded signal object using the API, you create the object, set its custom storage class and any custom attributes, then assign the object to the output port on which it will be embedded.

1. Name the signal if it does not already have a name. The name must be a valid ANSI C identifier.

2. In the MATLAB Command Window, enter:

```
SignalName=SignalClass
```

   where `SignalClass` is `Simulink.Signal` or any subclass of it that you have defined. The name of the signal object does not need to match the name of the signal to which the object will be applied.

3. Simulink creates a `SignalClass` object with the specified name. Example:

```
MySig =

Simulink.Signal (handle)
          RTWInfo: [1x1 Simulink.SignalRTWInfo]
      Description: ''
         DataType: 'auto'
              Min: -Inf
              Max: Inf
         DocUnits: ''
       Dimensions: -1
       Complexity: 'auto'
       SampleTime: -1
     SamplingMode: 'auto'
     InitialValue: ''
```

4. *Do not* set any attributes. An embedded signal object can specify *only* custom storage class information.

5. Follow the instructions in "Specifying a Custom Storage Class Using the API" on page 8-53. After specifying the custom storage class, be sure to assign the signal object to its output port, as described under "Assigning an Embedded Signal Object to an Output Port" on page 8-57.

Changing an Embedded Signal Object Using the API

To change an embedded signal object using the API, you obtain a copy of the object from the output port on which it is embedded, change the object as needed, then assign the changed object back to the port.

1 Obtain a copy of the signal object using a handle to the output port. Example:

```
hps=get_param(gcb,'PortHandles')
hp=hps.Outport(1)
MySig=get_param(hp,'SignalObject')
```

2 Change the signal object using the techniques described in "Specifying a Custom Storage Class Using the API" on page 8-53. After making the changes, be sure to copy the signal object to its output port, as described in "Assigning an Embedded Signal Object to an Output Port" on page 8-57.

Deleting an Embedded Signal Object Using the API

To delete an embedded signal object with the API, obtain a handle to the output port where the signal object is embedded, then set the port's `SignalObject` parameter to `[]`:

```
hps=get_param(gcb,'PortHandles')
hp=hps.Outport(1)
set_param(hp,'SignalObject',[])
```

Specifying a Custom Storage Class Using the GUI

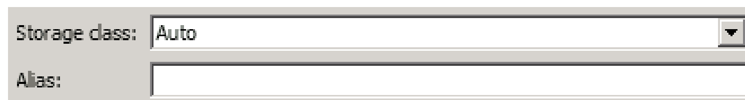
The initial steps for applying a CSC with the GUI differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- "Providing a Parameter Object Using the GUI" on page 8-38
- "Providing a Base Workspace Signal Object Using the GUI" on page 8-41
- "Providing an Embedded Signal Object using the GUI" on page 8-44

After the initial steps, applying a CSC with the GUI is the same in all three cases. The following instructions show you how to finish applying a CSC with the GUI. The instructions assume that you have completed one of the previous sets of instructions, and that the dialog you used to execute those instructions is still open. If necessary, return to the relevant section and restore the situation that existed at its end, then return to this section.

The instructions given in this section apply to all packages, but the available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the `Simulink` package.

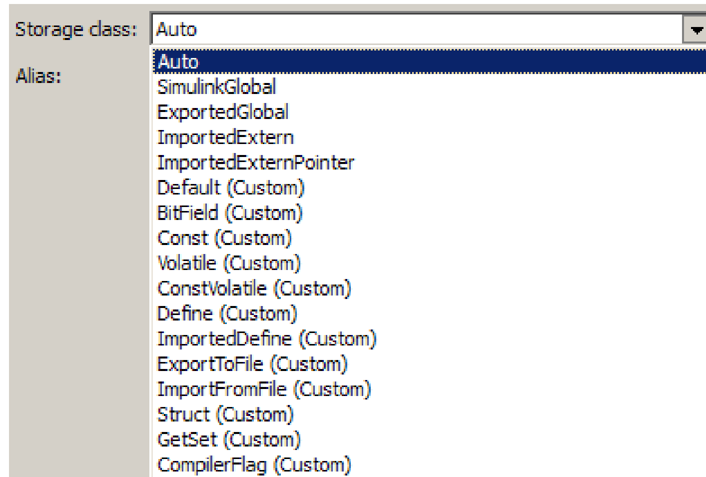
The dialog that you used to begin the process of applying a CSC with the GUI by providing an object contains two fields: one for specifying a custom storage class and one for optionally specifying an alias. These fields are the same in all three of the dialogs that you might use:



Storage class is `Auto` because that is the default storage class in the `Simulink` package. Other packages may have different defaults. You can specify an **Alias** whenever the **Storage class** is not `Auto`. If **Storage class** is `Auto`, Simulink deletes any alias you try to specify, leaving the field blank. If you specify an alias, it appears in generated code instead of the name of the object.

To specify a custom storage class and its custom attributes:

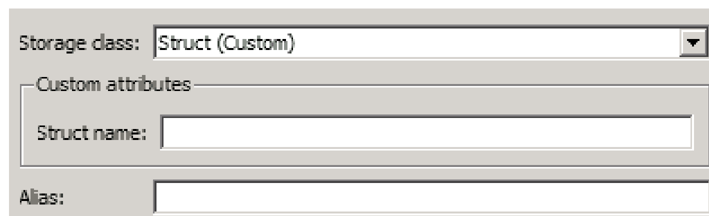
1 View the **Storage Class** menu, which looks like this for the `Simulink` package:



Each custom storage class has `(custom)` suffixed to its name. The storage classes `SimulinkGlobal`, `ExportedGlobal`, `ImportedExtern`, and `ImportedExternPointer` are the built-in non-custom storage classes described in "Signal Considerations" and "Defining Data Representation and Storage for Code Generation" in the Simulink Coder documentation.

2 Choose the desired custom storage class from **Storage class**, for example, `Struct`.

   If the storage class defines any custom attributes, fields for defining them appear:



3 Provide values for any custom attributes. `Struct` has only one, **Struct name**. For example, set **Struct name** to `MyStruct`:

4 Click **Apply**.

In generated code, all data whose storage is controlled by this custom storage class specification will appear in a structure named `MyStruct`. See "Generating Code with Custom Storage Classes" on page 8-58 for an example.

Specifying a Custom Storage Class Using the API

The initial steps for applying a CSC with the API differ depending on whether you are applying the CSC to a parameter using a base workspace object, to a signal using a base workspace object, or to a signal using an embedded object. The initial steps for each of these three cases appear in:

- "Providing a Parameter Object Using the API" on page 8-39
- "Providing a Base Workspace Signal Object Using the API" on page 8-42
- "Providing an Embedded Signal Object using the API" on page 8-48

After the initial steps, applying a CSC with the API is the same in all three cases, except for an assignment needed only by an embedded signal object. The following instructions show you how to finish applying a CSC with the API. The instructions assume that you have completed one of the previous sets of instructions, and that the resulting objects an attributes are unchanged. If necessary, return to the relevant section and restore the situation that existed at its end, then return to this section.

The instructions given in this section apply to all packages, but the available custom storage classes and custom attributes depend on the package that you select. The examples in this section assume that you are using the `Simulink` package. The examples also assume that the object for which you want to specify a custom storage class is named `MyObj`, which is a parameter or

signal object that exists in the base workspace, or a signal object that will be assigned to an output port.

The rest of this section provides information that is specific to custom storage classes in Embedded Coder. See “Simulink Package Custom Storage Classes” on page 8-6 for a list of the custom storage classes that are built into the `Simulink` package for use by Embedded Coder software.

RTWInfo Properties

Each Simulink parameter object or signal object defines properties called `RTWInfo` properties. Code generation software uses these properties to control storage class assignment in the generated code. The `RTWInfo` properties and their default values are:

```
          StorageClass: 'Auto'
                 Alias: ''
    CustomStorageClass: 'Default'
      CustomAttributes: [1x1 SimulinkCSC.AttribClass_Simulink_Default]
```

For more information about `RTWInfo` properties, see “Signal Considerations” and “Defining Data Representation and Storage for Code Generation” in the Simulink Coder documentation.

Specifying a Custom Storage Class

To specify a custom storage class using `RTWInfo` properties:

1. Set `StorageClass` to `'Custom'`.
2. Set `CustomStorageClass` to the name of the storage class.

For example, to specify the `Struct` custom storage class:

```
MyObj.RTWInfo.StorageClass='Custom'
MyObj.RTWInfo.CustomStorageClass='Struct'
```

Whenever you have specified a custom storage class other than `Auto`, you can specify an alias by setting the `Alias` attribute. If you specify an alias, it appears in generated code instead of the name of the object.

Specifying Instance-Specific Attributes

A custom storage class can have properties that define attributes that are specific to that CSC. Such properties are called **instance-specific attributes**. For example, if you specify the `Struct` custom storage class, you must specify the name of the C language structure that will store the data. That name is an instance-specific attribute of the `Struct` CSC.

Instance-specific attributes are stored in the `RTWInfo` property `CustomAttributes`. This property is initially defined as follows:

```
SimulinkCSC.AttribClass_Simulink_Default
1x1 struct array with no fields
```

When you specify a custom storage class, Simulink automatically populates `RTWInfo.CustomAttributes` with the fields necessary to represent any instance-specific attributes of that CSC. For example, if you set the `MySig` CSC to `Struct`, as described in “Specifying a Custom Storage Class” on page 8-54, then enter:

```
MyObj.RTWInfo.CustomAttributes
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
    StructName: ''
```

To specify that `StructName` is `MyStruct`, enter:

```
MyObj.RTWInfo.CustomAttributes.StructName='MyStruct'
```

MATLAB displays:

```
SimulinkCSC.AttribClass_Simulink_Struct
    StructName: 'MyStruct'
```

Simulink Package CSC Instance-Specific Properties

Class Name	Instance-Specific Property	Purpose
BitField	CustomAttributes.StructName	Name of the bitfield struct into which the code generator packs the object's Boolean data.
ExportToFile	CustomAttributes.HeaderFile	Name of header (.h) file that contains exported variable declarations and export directives for the object.
GetSet	CustomAttributes.HeaderFile	Name of header (.h) file to #include in the generated code. See "GetSet Custom Storage Class for Data Store Memory" on page 8-66.
	CustomAttributes.GetFunction	String that specifies the name of a function call to read data.
	CustomAttributes.SetFunction	String that specifies the name of a function call to write data.
ImportedDefine	CustomAttributes.HeaderFile	The header file that defines the values of code variant preprocessor conditionals. See "Generating Code for Variant Systems" on page 4-2.
ImportFromFile	CustomAttributes.HeaderFile	Name of header (.h) file containing global variable declarations the code generator imports for the object.
Struct	CustomAttributes.StructName	Name of the struct into which the code generator packs the object's data.

Assigning an Embedded Signal Object to an Output Port

If you are operating on an embedded signal object with the API, you must copy the object to the port after providing or changing its `RTWInfo` properties. For example, if `MyObj` is a signal object that you want to copy to the output port, enter:

```
hps=get_param(gcb,'PortHandles')
hp=hps.Outport(1)
set_param(hp,'SignalObject','MyObj')
```

Subsequent changes to the source object in the base workspace have no effect on the output port copy, and you can delete the source object if you have no further use for it:

```
clear ('MyObj')
```

Generating Code with Custom Storage Classes

In this section...

"Code Generation Prerequisites" on page 8-58
"Code Generation Example" on page 8-58

Code Generation Prerequisites

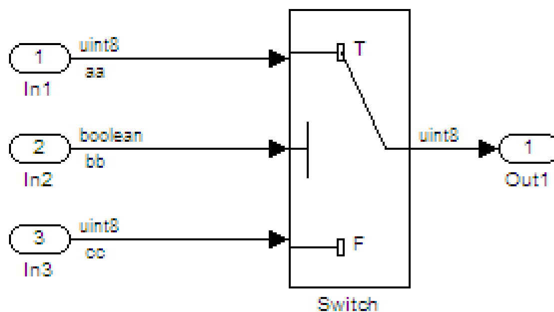
Before you generate code for a model that uses custom storage classes, set model options as follows:

- If your model assigns custom storage classes to any parameters, select **Configuration Parameters > Optimization > Signals and Parameters > Inline parameters**. Otherwise, the code generator ignores CSC specifications for parameters. This requirement also applies to models that assign built-in storage classes to parameters.
- Clear **Configuration Parameters > Code Generation > Data specification override > Ignore custom storage classes**.

  Otherwise, the code generator ignores all CSC specifications, and treats all data objects as if their **Storage class** were `Auto`.

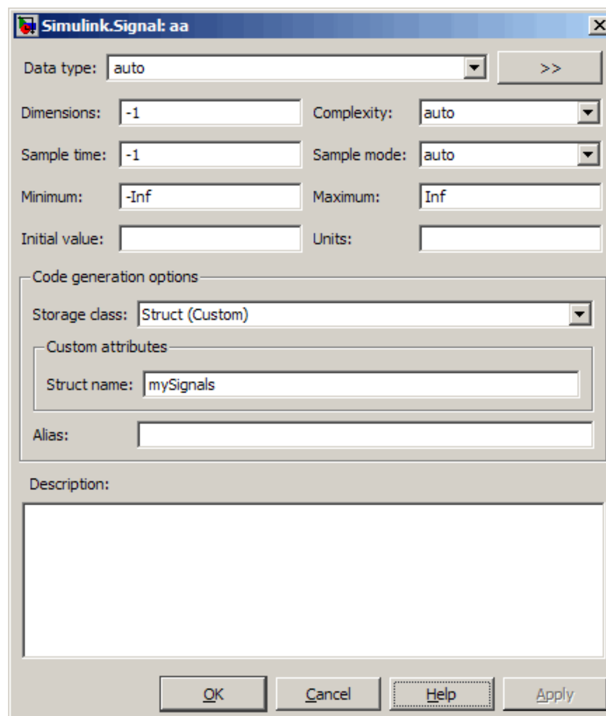
Code Generation Example

This section presents an example of code generation with CSCs, based on this model:



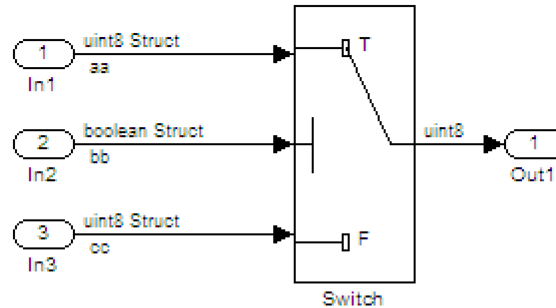
The model contains three named signals: `aa`, `bb`, and `cc`. Using the predefined `Struct` custom storage class, the example generates code that packs these signals into a `struct` named `mySignals`. The `struct` declaration is then exported to externally written code.

To specify the `struct`, you provide `Simulink.Signal` objects that specify the `Struct` custom storage class, and associate the objects with the signals as described in "Applying CSCs to Parameters and Signals" on page 8-37. All three objects have the same properties. This figure shows the signal object properties for `aa`:



The association between identically named model signals and signal objects is formed as described in "Resolving Symbols". In this example, the symbols `aa`, `bb`, and `cc` resolve to the signal objects `aa`, `bb`, and `cc`, which have custom storage class `Struct`. In the generated code, storage for the three signals will be allocated within a `struct` named `mySignals`.

You can display the storage class of the signals in the block diagram by selecting **Port/Signal Display > Storage Class** from the Simulink model editor **Format** menu. The figure below shows the block diagram with signal data types and signal storage classes displayed.



With the model configured as described in "Code Generation Prerequisites" on page 8-58, and the signal objects defined and associated with the signals, you can generate code that uses the custom storage classes to generate the desired data structure for the signals. After code generation, the relevant definitions and declarations are located in three files:

- `model_types.h` defines the following `struct` type for storage of the three signals:

  ```
  typedef struct MySignals_tag {
    boolean_T cc;
    uint8_T bb;
    uint8_T aa;
  } mySignals_type;
  ```

- `model.c` (or `.cpp`) defines the variable `mySignals`, as specified in the object's instance-specific `StructName` attribute. The variable is referenced in the code generated for the Switch block:

  ```
  /* Definition for Custom Storage Class: Struct */

  mySignals_type mySignals = {
  /* cc */
  ```

```
FALSE,
/* bb */
0,
/* aa */
  0
};
...
/*  Switch: '<Root>/Switch1'  */
  if(mySignals.cc) {
    rtb_Switch1 = mySignals.aa;
  } else {
    rtb_Switch1 = mySignals.bb;
  }
```

- `model.h` exports the `mySignals Struct` variable:

```
/* Declaration for Custom Storage Class: Struct */

extern mySignals_type mySignals;
```

Grouped Custom Storage Classes

A custom storage class that results in multiple data objects being referenced with a single variable in the generated code, in the previous example, is called a *grouped custom storage class*. In the `Simulink` package, `Bitfield` and `Struct` (shown in the preceding example) are grouped CSCs. Data grouped by a CSC is referred to as *grouped data*.

Defining Advanced Custom Storage Class Types

In this section...

"Introduction" on page 8-62

"Create Your Own Parameter and Signal Classes" on page 8-62

"Create a Custom Attributes Class for Your CSC (Optional)" on page 8-62

"Write TLC Code for Your CSC" on page 8-63

"Register Custom Storage Class Definitions" on page 8-63

Introduction

Certain data layouts, such as nested structures, cannot be generated using the standard `Unstructured` and `FlatStructure` custom storage class types. You can define an *advanced custom storage class* if you want to generate other types of data. Creating advanced CSCs requires understanding TLC programming and using a special advanced mode of the Custom Storage Class Designer. This sections explain how to define advanced CSC types.

Create Your Own Parameter and Signal Classes

The first step is to use the Simulink Data Class Designer to create your own package containing classes derived from `Simulink.Parameter` or `Simulink.Signal`. This procedure is described in "Creating Packages that Support CSC Definitions" on page 8-8.

Add your own object properties and class initialization if desired. For each of your classes, select the **Create your own custom storage classes for this class** option.

Create a Custom Attributes Class for Your CSC (Optional)

If you have instance-specific properties that are relevant only to your CSC, you should use the Simulink Data Class Designer to create a *custom attributes class* for the package. A custom attributes class is a subclass of `Simulink.CustomStorageClassAttributes`. The name, type, and default

value properties you set for the custom attributes class define the user view of instance-specific properties.

For example, the `ExportToFile` custom storage class requires that you set the `RTWInfo.CustomAttributes.HeaderFile` property to specify a `.h` file used for exporting each piece of data. See "Simulink Package Custom Storage Classes" on page 8-6 for further information on instance-specific properties.

Write TLC Code for Your CSC

The next step is to write TLC code that implements code generation for data of your new custom storage class. A template TLC file is provided for this purpose. To create your TLC code, follow these steps:

1. Create a `tlc` directory inside your package's @directory (if it does not already exist). The naming convention to follow is

   ```
   @PackageName/tlc
   ```

2. Copy `TEMPLATE_v1.tlc` (or another CSC template) from *`matlabroot`*`/toolbox/rtw/targets/ecoder/csc_templates` into your `tlc` directory to use as a starting point for defining your custom storage class.

3. Write your TLC code, following the comments in the CSC template file. Comments describe how to specify code generation for data of your custom storage class (for example, how data structures are to be declared, defined, and whether they are accessed by value or by reference).

Alternatively, you can copy a custom storage class TLC file from another existing package as a starting point for defining your custom storage class.

Register Custom Storage Class Definitions

After you have created a package for your new custom storage class and written its associated TLC code, you must register your class definitions with the Custom Storage Class Designer, using its advanced mode.

The advanced mode supports selection of an additional storage class **Type**, designated `Other`. The `Other` type is designed to support special CSC types that cannot be accommodated by the standard `Unstructured` and

`FlatStructure` custom storage class types. The `Other` type cannot be assigned to a CSC except when the Custom Storage Class Designer is in advanced mode.

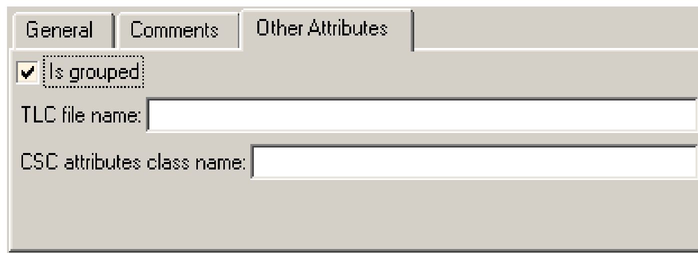
To register your class definitions:

1 Launch the Custom Storage Class Designer in advanced mode by typing the following command at the MATLAB prompt:

```
cscdesigner -advanced
```

2 Select your package and create a new custom storage class.

3 Set the **Type** of the custom storage class to `Other`. Note that when you do this, the **Other Attributes** pane is displayed. This pane is visible only for CSCs whose **Type** is set to `Other`.



If you specify a customized package, additional options, as defined by the package, also appear on the **Other Attributes** pane.

4 Set the properties shown on the **Other Attributes** pane. The properties are:

- **Is grouped**: Select this option if you intend to combine multiple data objects of this CSC into a single variable in the generated code. (for example, a struct).
- **TLC file name**: Enter the name of the TLC file corresponding to this custom storage class. The location of the file is assumed to be in the `/tlc` subdirectory for the package, so you should not enter the path to the file.
- **CSC attributes class name**: (optional) If you created a custom attributes class corresponding to this custom storage class, enter the full

name of the custom attributes class. (see "Create a Custom Attributes Class for Your CSC (Optional)" on page 8-62).

5. Set the remaining properties on the **General** and **Comments** panes based on the layout of the data that you wish to generate (as defined in your TLC file).

GetSet Custom Storage Class for Data Store Memory

In this section...

"Overview" on page 8-66
"GetSet CSC Properties" on page 8-66
"Using the GetSet CSC" on page 8-67
"GetSet CSC Restrictions" on page 8-67
"GetSet Custom Storage Class Example" on page 8-68

Overview

`GetSet` is a built-in advanced custom storage class that generates specialized function calls to read from (get) and write to (set) the memory associated with a Data Store Memory block that is read and written many times in a single model. See "Working with Data Stores" for information about data stores and the Data Store Memory block, and for information about advanced CSCs.

The `GetSet` custom storage class is designed primarily for use with the state of the Data Store Memory block. However, `GetSet` is capable of handling signals other than data stores, and is supported for the outputs of most built-in blocks provided by MathWorks. For more about the definition of the `GetSet` storage class, look at its associated TLC code in the file:

```
matlabroot\toolbox\simulink\simulink\@Simulink\tlc\GetSet.tlc
```

GetSet CSC Properties

The next table summarizes the instance-specific properties of the `GetSet` storage class:

Property	Description
GetFunction	String that specifies the name of a function call to read data.
SetFunction	String that specifies the name of a function call to write data.
HeaderFile (optional)	String that specifies the name of a header (.h) file to add as an #include in the generated code.

For example, if the GetFunction of signal X is specified as 'get_X' then the generated code calls get_X() wherever the value of X is used. Similarly, if the SetFunction of signal X is specified as 'set_X' then the generated code calls set_X(value) wherever the value of X is assigned.

Using the GetSet CSC

The GetSet storage class cannot be represented by the standard Unstructured or FlatStructure custom storage class types, so it is an advanced CSC, as described in "Defining Advanced Custom Storage Class Types" on page 8-62. To access the CSC definition for GetSet, you must launch Custom Storage Class designer in advanced mode:

```
cscdesigner -advanced
```

If you omit the HeaderFile property for a GetSet data object, you must specify a header file by an alternative means, such as the **Header file** field of the **Code Generation > Custom Code** pane of the Configuration Parameters dialog box. Otherwise, the generated code might not compile or might function improperly.

For wide signals, an additional index argument is passed, so the calls to the get and set functions are get_X(idx) and set_X(idx, value) respectively.

GetSet CSC Restrictions

- The GetSet supports only signals of noncomplex data types.
- Some built-in blocks do not directly support GetSet.
- User-written S-functions do not directly support GetSet.

To use `GetSet` with a nonsupporting built-in block or a user-written S-function:

1. Insert a Signal Conversion block at the outport of the block or function.
2. Select the Signal Conversion Block's **Exclude this block from 'Block reduction' optimization** property.
3. Assign the `GetSet` storage class to the output of the Signal Conversion block.

GetSet Custom Storage Class Example

The model below contains a Data Store Memory block that resolves to the Simulink signal object `X`:

The following specifications configure the signal object `X` to use the `GetSet` custom storage class:

```
X = Simulink.Signal;
X.RTWInfo.StorageClass                    = 'Custom';
X.RTWInfo.CustomStorageClass              = 'GetSet';
X.RTWInfo.CustomAttributes.GetFunction = 'get_X';
X.RTWInfo.CustomAttributes.SetFunction = 'set_X';
X.RTWInfo.CustomAttributes.HeaderFile  = 'user_file.h';
```

The `GetSet` CSC appears as follows in the code generated for the model:

```
/* Includes for objects with custom storage classes. */
   #include "user_file.h"

   void getset_csc_step(void)
```

```
{
  /* local block i/o variables */
  real_T rtb_DSRead_o;

  /* DataStoreWrite: '<Root>/DSWrite' incorporates:
   *   Inport: '<Root>/In1'
   */
  set_X(getset_csc_U.In1);

  /* DataStoreRead: '<Root>/DSRead' */
  rtb_DSRead_o = get_X();

  /* Outport: '<Root>/Out1' */
  getset_csc_Y.Out1 = rtb_DSRead_o;
}
```

Note that the code uses a local variable `rtb_DSRead_o` rather than multiple calls to the `get_X` function. This technique increases code efficiency and ensures that the value does not change within a simulation step.

Custom Storage Class Implementation

You can skip this section unless you want to ship custom storage class definitions in an uneditable format, or you intend to bypass the Custom Storage Class designer and work directly with files that contain custom storage class definitions.

The file that defines a package's custom storage classes is called a *CSC registration file*. The file is always named `csc_registration` and resides in the *@package* directory that defines the package. A CSC registration file can be a P-file (`csc_registration.p`) or a MATLAB file (`csc_registration.m`). A built-in package defines custom storage classes in both a P-file and a functionally equivalent MATLAB file. A user-defined package initially defines custom storage classes only in a MATLAB file.

P-files take precedence over MATLAB files, so when MATLAB looks for a package's CSC registration file and finds both a P-file and a MATLAB file, MATLAB loads the P-file and ignores the MATLAB file. All capabilities and tools, including the Custom Storage Class Designer, then use the CSC definitions stored in the P-file. P-files cannot be edited, so all CSC Designer editing capabilities are disabled for CSCs stored in a P-file. If no P-file exists, MATLAB loads CSC definitions from the MATLAB file. MATLAB files are editable, so all CSC Designer editing capabilities are enabled for CSCs stored in a MATLAB file.

Because CSC definitions for a built-in package exist in both a P-file and a MATLAB file, they are uneditable. You can make the definitions editable by deleting the P-file, but MathWorks strongly discourages modifying built-in CSC registration files or any other files under `matlabroot`. The preferred technique is to create user-defined packages, data classes, and custom storage classes, as described in "Subclassing Simulink Data Classes" and this chapter.

The CSC Designer saves CSC definitions for user-defined packages in a MATLAB file, so the definitions are editable. You can make the definitions uneditable by using the `pcode` function to create an equivalent P-file, which will then shadow the MATLAB file. However, you should preserve the MATLAB file if you may need to make further changes, because you cannot modify CSC definitions that exist only in a P-file.

You can also use tools or techniques other than the Custom Storage Class Designer to create and edit MATLAB files that define CSCs. However, MathWorks discourages this practice, which is vulnerable to syntax errors and can give unexpected results. When MATLAB finds an older P-file that shadows a newer MATLAB file, it displays a warning in the MATLAB Command Window.

Custom Storage Class Limitations

- Data objects cannot have a CSC and a multi-word data type.
- The Fcn block does not support parameters with custom storage class in code generation.
- For CSCs in models that use referenced models (see "Referencing a Model"):
  - If data is assigned a grouped CSC, such as `Struct` or `Bitfield`, the CSC's **Data scope** property must be `Imported` and the data declaration must be provided in a user-supplied header file. See "Grouped Custom Storage Classes" on page 8-61 for more information about grouped CSCs.
  - If data is assigned an ungrouped CSC, such as `Const`, and the data's **Data scope** property is `Exported`, its **Header file** property must be unspecified. This results in the data being exported with the standard header file, *`model`*`.h`. Note that for ungrouped data, the **Data scope** and **Header file** properties are either specified by the selected CSC, or as one of the data object's instance-specific properties.

9

Memory Sections

- “Introduction to Memory Sections” on page 9-2
- “Requirements for Defining Memory Sections” on page 9-4
- “Defining Memory Sections” on page 9-7
- “Configuring Memory Sections” on page 9-11
- “Declaring Constant Data as Volatile” on page 9-12
- “Applying Memory Sections” on page 9-15
- “Examples of Generated Code with Memory Sections” on page 9-23
- “Model-Level Data Structures” on page 9-25
- “Memory Section Limitation” on page 9-28

Introduction to Memory Sections

In this section...

"Overview" on page 9-2
"Memory Sections Demo" on page 9-2
"Additional Information" on page 9-2

Overview

The Embedded Coder software provides a memory section capability that allows you to insert comments and pragmas and qualify constants as `volatile` in generated code for

- Data in custom storage classes
- Model-level functions
- Model-level internal data
- Subsystem functions
- Subsystem internal data

Pragmas inserted into generated code can surround

- A contiguous block of function or data definitions
- Each function or data definition separately

When pragmas surround each function or data definition separately, the text of each pragma can contain the name of the definition to which it applies.

Memory Sections Demo

To see a demo of memory sections, type `rtwdemo_memsec` in the MATLAB Command Window.

Additional Information

See the following for additional information relating to memory sections:

- Simulink data types, packages, data classes, and data objects — “Working with Data” in the Simulink documentation
- Simulink Coder data structures and storage classes — “Defining Data Representation and Storage for Code Generation” in the Simulink Coder documentation
- Chapter 8, “Creating and Using Custom Storage Classes”
- Fine-tuned optimization of generated code for functions or data — *Simulink Coder Target Language Compiler* documentation

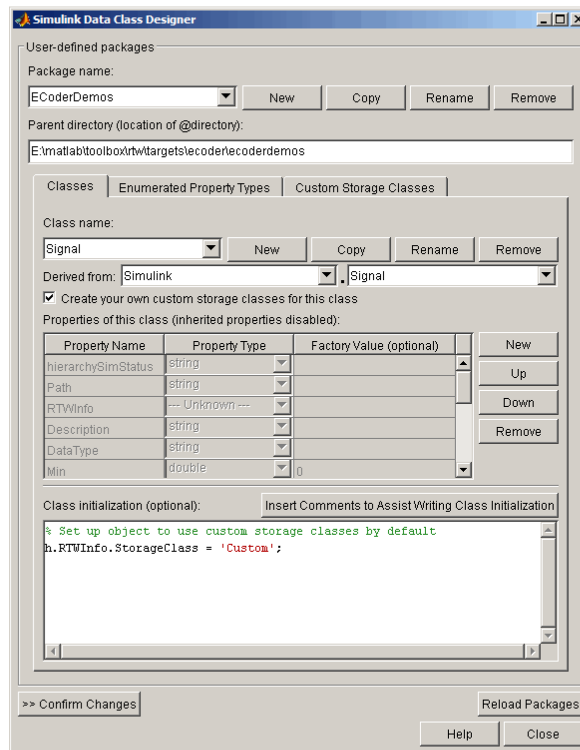
Requirements for Defining Memory Sections

Before you can define memory sections, you must do the following:

1. Set the Simulink model's code generation target to an embedded target such as `ert.tlc`.

2. If you need to create packages, specify package properties, or create classes, including custom storage classes, choose **Tools > Data Class Designer** in the model window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the Simulink Data Class Designer appears. The Data Class Designer initially looks like this:



Complete instructions for using the Data Class Designer appear in "Subclassing Simulink Data Classes" in the Simulink documentation. See also the instructions that appear when you click the **Custom Storage Classes** tab.

3 If you need to specify custom storage class properties,

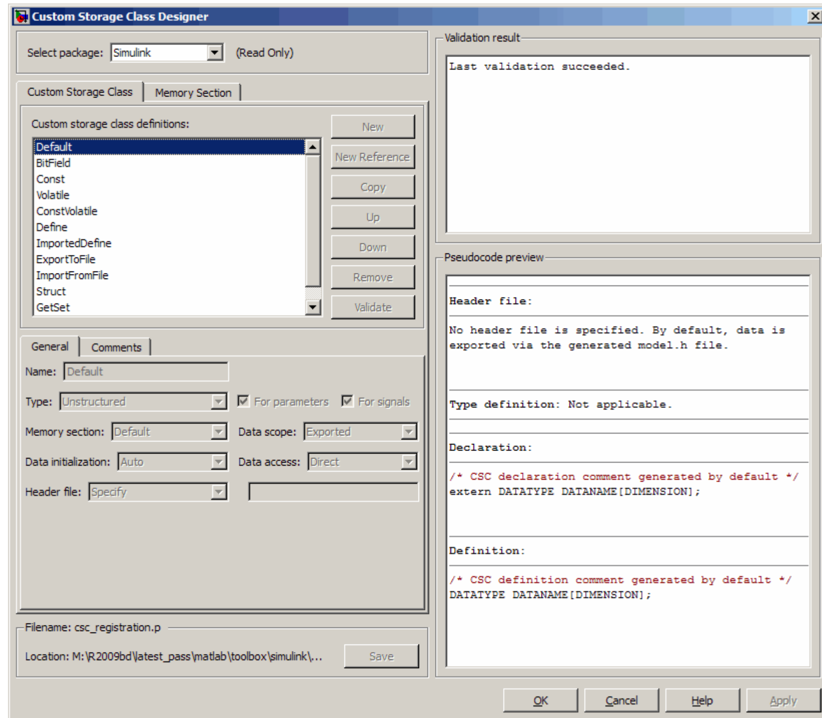
   a Choose **View > Model Explorer** in the model window.

      The Model Explorer appears.

   b Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

      A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

   c Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



d Use the **Custom Storage Class** pane as needed to select a writable package and specify custom storage class properties. Instructions for using this pane appear in “Designing Custom Storage Classes and Memory Sections” on page 8-12.

Defining Memory Sections

In this section...

"Editing Memory Section Properties" on page 9-7

"Specifying the Memory Section Name" on page 9-9

"Specifying a Qualifier for Custom Storage Class Data Definitions" on page 9-9

"Specifying Comment and Pragma Text" on page 9-9

"Surrounding Individual Definitions with Pragmas" on page 9-9

"Including Identifier Names in Pragmas" on page 9-10

Editing Memory Section Properties

After you have satisfied the requirements in "Requirements for Defining Memory Sections" on page 9-4, you can define memory sections and specify their properties. To create new memory sections or specify memory section properties,

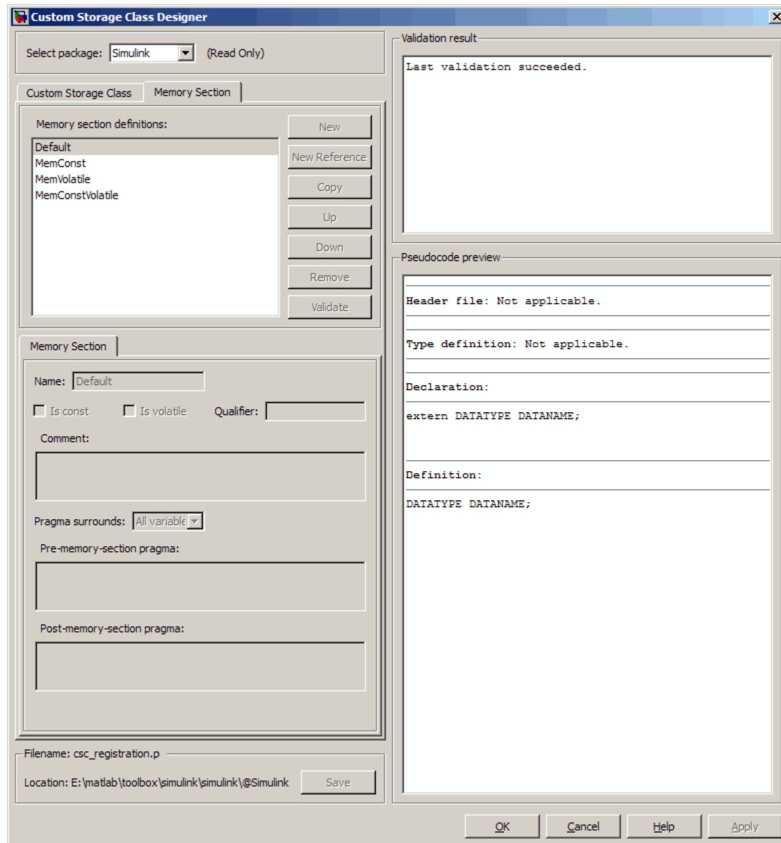
1. Choose **View > Model Explorer** in the model window.

   The Model Explorer appears.

2. Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

3. Click the **Memory Section** tab of the Custom Storage Class Designer. The **Memory Section** pane initially looks like this:



4 If you intend to create or change memory section definitions, use the **Select package** field to select a writable package.

The rest of this section assumes that you have selected a writable package, and describes the use of the **Memory section** subpane on the lower left. For descriptions of the other subpanes, instructions for validating memory section definitions, and other information, see "Defining Memory Sections" on page 9-7.

Specifying the Memory Section Name

To specify the name of a memory section, use the **Name** field. A memory section name must be a legal MATLAB identifier.

Specifying a Qualifier for Custom Storage Class Data Definitions

To specify a qualifier for custom storage class data definitions in a memory section, enter the components of the qualifier below the **Name** field.

- To specify `const`, check **Is const**.
- To specify `volatile`, check **Is volatile**.
- To specify anything else (e.g., `static`), enter the text in the **Qualifier** field.

The qualifier will appear in generated code with its components in the same left-to-right order in which their definitions appear in the dialog box. A preview appears in the **Pseudocode preview** subpane on the lower right.

Note Specifying a qualifier affects only custom storage class data definitions. The code generator omits the qualifier from any other category of definition.

Specifying Comment and Pragma Text

To specify a comment, prepragma, or postpragma for a memory section, enter the text in the appropriate edit boxes on the left side of the Custom Storage Class Designer. These boxes accept multiple lines separated by ordinary Returns.

Surrounding Individual Definitions with Pragmas

If the **Pragma surrounds** field for a memory section specifies `Each variable`, the code generator will surround each definition in a contiguous block of definitions with the comment, prepragma, and postpragma defined for the section. This behavior occurs with all categories of definitions.

If the **Pragma surrounds** field for a memory section specifies `All variables`, the code generator will insert the comment and prepragma for the

section before the first definition in a contiguous block of custom storage class data definitions, and the postpragma after the last definition in the block.

Note Specifying `All variables` affects only custom storage class data definitions. For any other category of definition, the code generator surrounds each definition separately regardless of the value of **Pragma surrounds**.

Including Identifier Names in Pragmas

When pragmas surround each separate definition in a contiguous block, you can include the string `%<identifier>` in a pragma. The string must appear without surrounding quotes.

- When `%<identifier>` appears in a prepragma, the code generator will substitute the identifier from the subsequent function or data definition.
- When `%<identifier>` appears in a postpragma, the code generator will substitute the identifier from the previous function or data definition.

You can use `%<identifier>` with pragmas *only* when pragmas to surround each variable. The `Validate` phase will report an error if you violate this rule.

Note Although `%<identifier>` looks like a TLC variable, it is not: it is just a keyword that directs the code generator to substitute the applicable data definition identifier when it outputs a pragma. TLC variables cannot appear in pragma specifications in the **Memory Section** pane.

Configuring Memory Sections

You configure memory sections by using the **Code Generation > Memory Sections** pane of the Configuration Parameters dialog box.

To...	Select...
Specify the package that contains memory sections that you want to apply	The name of a package for **Package**. Click **Refresh package list** to refresh the list of available packages in your configuration.
Apply memory sections to initialize/start and terminate functions	A value for **Initialize/Terminate**.
Apply memory sections to step, run-time initialization, derivative, enable, and disable functions	A value for **Execution**.
Apply memory sections to constant parameters, constant, block I/O, or zero representation	A value for **Constants**.
Apply memory sections to root inputs or outputs	A value for **Inputs/Outputs**.
Apply memory sections to block I/O, Dwork vectors, run-time models, zero-crossings	A value for **Internal data**.
Apply memory sections to parameters	A value for **Parameters**.

The interface checks whether the specified package is on the MATLAB path and that the selected memory sections are in the package. The results of this validation appear in the field **Validation results**.

Declaring Constant Data as Volatile

In the C language, the value of data declared with the storage type qualifier, `volatile`, can be read from memory when needed and written back to memory when changed without compiler control or detection. Examples of use include variables for initialization at system power-up or for system clock updates.

You can add the `volatile` qualifier to type definitions generated in code for model constant block I/O, constant parameters, and ground data (zero representation).

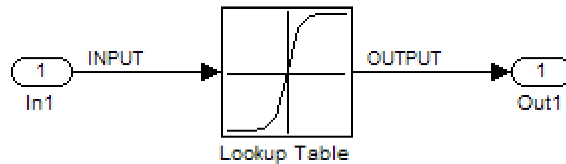
To add the `volatile` qualifier to type definitions, you must configure your model as follows:

- Enable inline parameters
- Specify an ERT target
- Set the memory section for constant data to `MemVolatile` or `MemConstVolatile`

If you choose to add the `volatile` qualifier to type definitions in your generated code, note the following:

- If constant data that is qualified with `volatile` is passed by pointer, the code generator casts away the volatility. This occurs because generated functions assume that data values do not change during execution and, therefore, pass their arguments as `const *` (not `const volatile *`).
- If a variable must be declared `const` and you specify `MemVolatile`, no warning occurs and the code generator declares the variable with the `const` and `volatile` qualifiers.
- If you set **Constants** to `MemConst` or `MemConstVolatile`, and a variable cannot be declared as constant data, a TLC warning appears and the code generator does not qualify the variable with `const`.

Consider the following simple lookup table model.



1. On the Configuration Parameters dialog box, in the **Optimization > Signals and Parameters** pane, select **Inline parameters**.

2. In the **Code Generation** pane, set **System target file** to `ert.tlc`.

3. In the **Code Generation > Memory Sections** pane, set **Package** to `Simulink` or `mpt`, and **Constants** to `MemConstVolatile`.

4. Open the Signal Properties dialog box for signal `INPUT`. On the **Code Generation** tab, set the **Package** to `Simulink` or `mpt` and the **Storage class** to `ExportedGlobal` for storing state in a global variable.

5. Generate code. You should see the `volatile` qualifier in the generated files *`model`*`_data.c` and *`model`*`.h`.

   `model``_data.c`

```
/* Constant parameters (auto storage) */
/* ConstVolatile memory section */
const volatile ConstParam_simple_lookup simple_lookup_ConstP = {
  /* Expression: [-5:5]
   * Referenced by: '<Root>/Lookup Table'
   */
  { -5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },

  /* Expression: tanh([-5:5])
   * Referenced by: '<Root>/Lookup Table'
   */
  { -0.99990920426259511, -0.999329299739067,
    -0.99505475368673046, -0.9640275800758169,
    -0.76159415595576485, 0.0, 0.76159415595576485,
    0.9640275800758169, 0.99505475368673046,
    0.999329299739067, 0.99990920426259511 }
```

```
};
```

model.h

```
/* Real-time Model Data Structure */
struct RT_MODEL_simple_lookup {
  const char_T * volatile errorStatus;
};

/* Constant parameters (auto storage) */
extern const volatile ConstParam_simple_lookup simple_lookup_ConstP;
```

Also note in the *model*.c file that a typecast is inserted in the `rt_Lookup` function call, removing the `volatile` qualifier.

```
/* Lookup: '<Root>/Lookup Table' incorporates:
 *  Inport: '<Root>/In1'
 */
OUTPUT = rt_Lookup(((const real_T*)
  &simple_lookup_ConstP.LookupTable_XData[0]), 11, INPUT, ((
  const real_T*) &simple_lookup_ConstP.LookupTable_YData[0]));
```

Applying Memory Sections

In this section...

"Assigning Memory Sections to Custom Storage Classes" on page 9-15

"Applying Memory Sections to Model-Level Functions and Internal Data" on page 9-17

"Applying Memory Sections to Atomic Subsystems" on page 9-19

Assigning Memory Sections to Custom Storage Classes

To assign a memory section to a custom storage class,

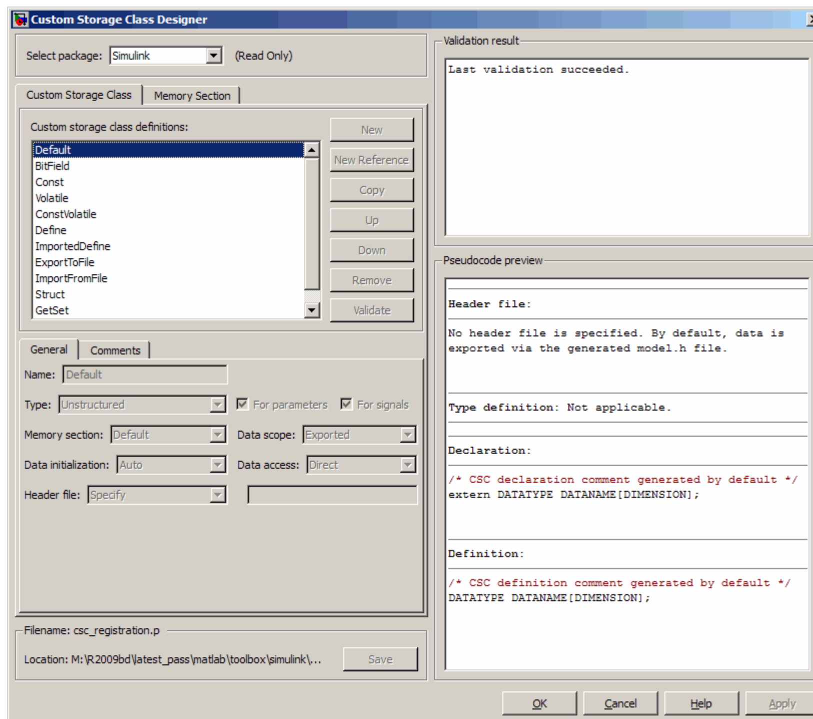
1. Choose **View > Model Explorer** in the model window.

   The Model Explorer appears.

2. Choose **Tools > Custom Storage Class Designer** in the Model Explorer window.

   A notification box appears that states **Please Wait ... Finding Packages**. After a brief pause, the notification box closes and the Custom Storage Class Designer appears.

3. Select the **Custom Storage Class** tab. The **Custom Storage Class** pane initially looks like this:



4 Use the **Select package** field to select a writable package. The rest of this section assumes that you have selected a writable package.

5 Select the desired custom storage class in the **Custom storage class definitions** pane.

6 Select the desired memory section from the **Memory section** pull-down.

7 Click **Apply** to apply changes to the open copy of the model; **Save** to apply changes and save them to disk; or **OK** to apply changes, save changes, and close the Custom Storage Class Designer.

Generated code for all data definitions in the specified custom storage class will be enclosed in the pragmas of the specified memory section. The pragmas can surround contiguous blocks of definitions or each definition separately, as described in "Surrounding Individual Definitions with Pragmas" on page 9-9.

For more information, see "Creating Packages that Support CSC Definitions" on page 8-8.

Note The code generator does not generate a `pragma` around definitions or declarations for data that has the following built-in storage classes:

- `ExportedGlobal`
- `ImportedExtern`
- `ImportedExternPointer`

The code generator treats data with these built-in storage classes like custom storage classes with no memory section specified.

Applying Memory Sections to Model-Level Functions and Internal Data

When using code generation software, you can apply memory sections to the following categories of model-level functions:

Function Category	Function Subcategory
Initialize/Terminate functions	Initialize/Start
	Terminate
Execution functions	Step functions
	Run-time initialization
	Derivative
	Enable
	Disable

When using code generation software, you can apply memory sections to the following categories of internal data:

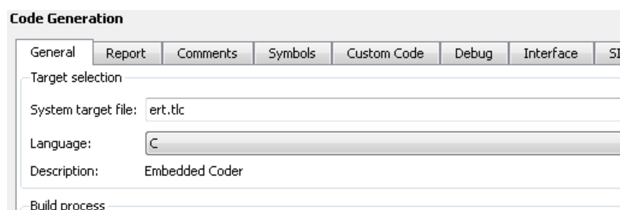
Data Category	Data Definition	Data Purpose
Constants	*model*_cP	Constant parameters
	*model*_cB	Constant block I/O
	*model*_Z	Zero representation
Input/Output	*model*_U	Root inputs
	*model*_Y	Root outputs
Internal data	*model*_B	Block I/O
	*model*_D	D-work vectors
	*model*_M	Run-time model
	*model*_Zero	Zero-crossings
Parameters	*model*_P	Parameters

Memory section specifications for model-level functions and internal data apply to the top level of the model and to all subsystems *except* atomic subsystems that contain overriding memory section specifications, as described in "Applying Memory Sections to Atomic Subsystems" on page 9-19.

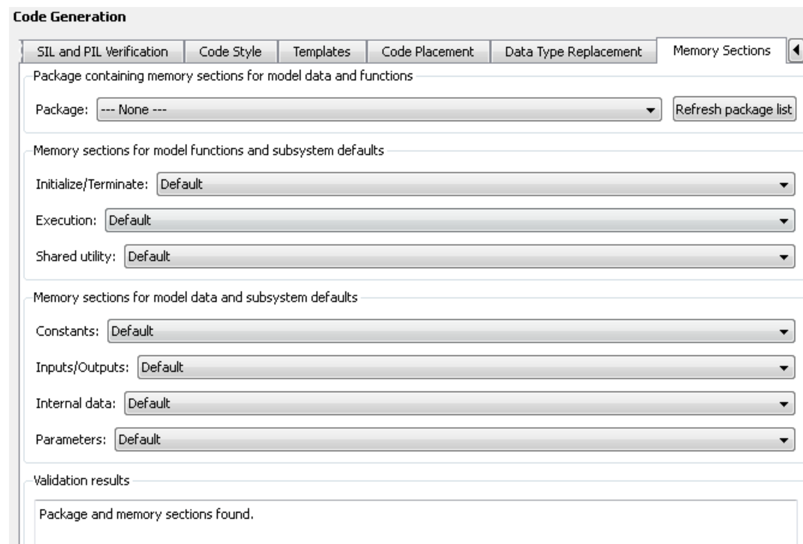
To specify memory sections for model-level functions or internal data,

1 Open the Model Explorer and select **Configuration (Active)** > **Code Generation** > **General**. (Alternatively, choose **Simulation > Configuration Parameters** in the model window.)

2 Ensure that the **System target file** is an ERT target, such as `ert.tlc` .



3 Select the **Memory Sections** tab. The **Memory Sections** pane looks like this:



4 Initially, the **Package** field specifies `---None---` and the pull-down lists only built-in packages. If you have defined any packages of your own, click **Refresh package list**. This action adds all user-defined packages on your search path to the package list.

5 In the **Package** pull-down, select the package that contains the memory sections that you want to apply.

6 In the pull-down for each category of internal data and model-level function, specify the memory section (if any) that you want to apply to that category. Accepting or specifying `Default` omits specifying any memory section for that category.

7 Click **Apply** to save any changes to the package and memory section selections.

Applying Memory Sections to Atomic Subsystems

For any atomic subsystem whose generated code format is `Function` or `Reusable Function`, you can specify memory sections for functions and

internal data that exist in that code format. Such specifications override any model-level memory section specifications. Such overrides apply only to the atomic subsystem itself, not to any subsystems within it. Subsystems of an atomic subsystem inherit memory section specifications from the containing model, *not* from the containing atomic subsystem.

To specify memory sections for an atomic subsystem,

1. Right-click the subsystem in the model window.

2. Choose **Subsystem Parameters** from the context menu. The Function Block Parameters: `Subsystem` dialog box appears.

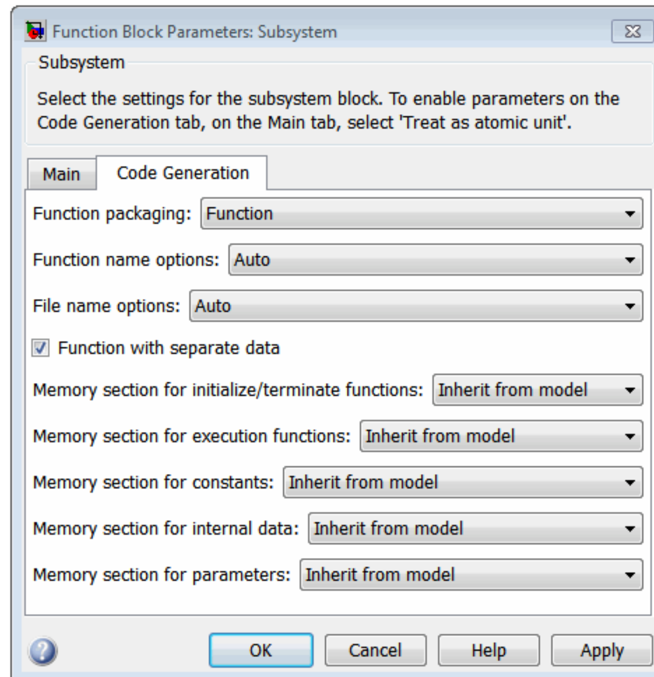
3. Ensure that **Treat as atomic unit** is checked. Otherwise, you cannot specify memory sections for the subsystem.

   For an atomic system, on the **Code Generation** tab, you can use the **Function packaging** field to control the format of the generated code.

4. Ensure that **Function packaging** is `Function` or `Reusable function`. Otherwise, you cannot specify memory sections for the subsystem.

5. If the code format is `Function` and you want separate data, check **Function with separate data**.

   The **Code Generation** tab now shows all applicable memory section options. The available options depend on the values of **Function packaging** and the **Function with separate data** check box. When the former is `Function` and the latter is checked, the pane looks like this:



6 In the pull-down for each available definition category, specify the memory section (if any) that you want to apply to that category.

- Selecting `Inherit from model` inherits the corresponding selection (if any) from the model level (not any parent subsystem).
- Selecting `Default` specifies that the category has no associated memory section, overriding any model-level specification for that category.

7 Click **Apply** to save changes, or **OK** to save changes and close the dialog box.

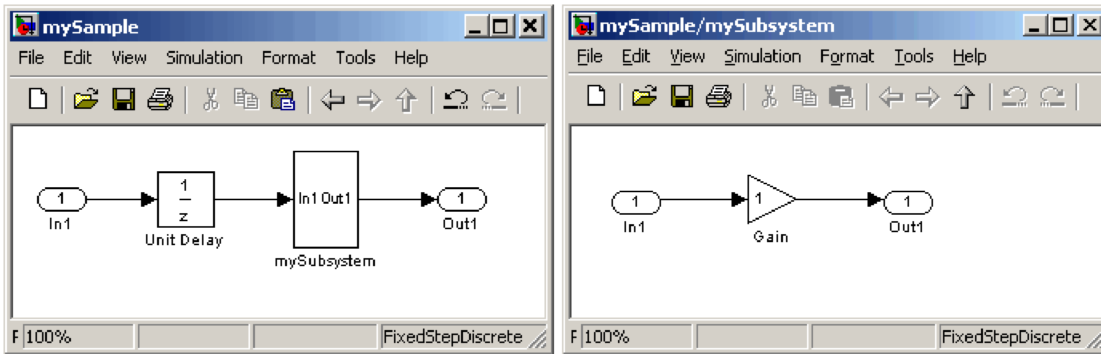
Caution If you use **Build Subsystem** to generate code for an atomic subsystem that specifies memory sections, the code generator ignores the subsystem-level specifications and uses the model-level specifications instead. The generated code is the same as if the atomic subsystem specified `Inherit from model` for every category of definition. For information about **Build Subsystem**, see "Generating Code and Executables from Subsystems".

It is not possible to specify the memory section for a subsystem in a library. However, you can specify the memory section for the subsystem after you have copied it into a Simulink model. This is because in the library it is unknown what code generation target will be used. You can copy a library block into many different models with different code generation targets and different memory sections available.

Examples of Generated Code with Memory Sections

Sample ERT-Based Model with Subsystem

The next figure shows an ERT-based Simulink model that defines one subsystem, and the contents of that subsystem.



Assume that the subsystem is atomic. On the **Code Generation** tab, the **Function packaging** parameter is `Reusable function`. Memory sections have been created and assigned as shown in the next two tables, and all data memory sections specify **Pragma surrounds** to be `Each variable`.

Model-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Input/Output	`MemSect1`	Prepragma	`#pragma IO_begin`
		Postpragma	`#pragma IO-end`
Internal data	`MemSect2`	Prepragma	`#pragma InData-begin(%<identifier>)`
		Postpragma	`#pragma InData-end`
Parameters	`MemSect3`	Prepragma	`#pragma Parameters-begin`
		Postpragma	`#pragma Parameters-end`

Model-Level Memory Section Assignments and Definitions (Continued)

Section Assignment	Section Name	Field Name	Field Value
Initialize/ Terminate	MemSect4	Prepragma	#pragma InitTerminate-begin
		Postpragma	#pragma InitTerminate-end
Execution functions	MemSect5	Prepragma	#pragma ExecFunc-begin(%<identifier>)
		Postpragma	#pragma ExecFunc-begin(%<identifier>)

Subsystem-Level Memory Section Assignments and Definitions

Section Assignment	Section Name	Field Name	Field Value
Execution functions	MemSect6	Prepragma	#pragma DATA_SEC(%<identifier>, "FAST_RAM")
		Postpragma	

Given the preceding specifications and definitions, the code generator would create the following code, with minor variations depending on the current version of the Target Language Compiler.

Model-Level Data Structures

```
#pragma IO-begin
ExternalInputs_mySample mySample_U;
#pragma IO-end

#pragma IO-begin
ExternalOutputs_mySample mySample_Y;
#pragma IO-end

#pragma InData-begin(mySample_B)
BlockIO_mySample mySample_B;
#pragma InData-end

#pragma InData-begin(mySample_DWork)
D_Work_mySample mySample_DWork;
#pragma InData-end

#pragma InData-begin(mySample_M_)
RT_MODEL_mySample mySample_M_;
#pragma InData-end

#pragma InData-begin(mySample_M)
RT_MODEL_mySample *mySample_M = &mySample_M_;
#pragma InData-end

#pragma Parameters-begin
Parameters_mySample mySample_P = {
  0.0 , {2.3}
};
#pragma Parameters-end
```

Model-Level Functions

```
#pragma ExecFunc-begin(mySample_step)
void mySample_step(void)
{
  real_T rtb_UnitDelay;
  rtb_UnitDelay = mySample_DWork.UnitDelay_DSTATE;
  mySubsystem(rtb_UnitDelay, &mySample_B.mySubsystem;,
```

```
    (rtP_mySubsystem *) &mySample_P.mySubsystem);
  mySample_Y.Out1_o = mySample_B.mySubsystem.Gain;
  mySample_DWork.UnitDelay_DSTATE = mySample_U.In1;
}
#pragma ExecFunc-end(mySample_step)

#pragma InitTerminate-begin
void mySample_initialize(void)
{
  rtmSetErrorStatus(mySample_M, (const char_T *)0);
  {
    ((real_T*)&mySample_B.mySubsystem.Gain)[0] = 0.0;
  }
  mySample_DWork.UnitDelay_DSTATE = 0.0;
  mySample_U.In1 = 0.0;
  mySample_Y.Out1_o = 0.0;
  mySample_DWork.UnitDelay_DSTATE = mySample_P.UnitDelay_X0;
}
#pragma InitTerminate-end
```

Subsystem Function

Because the subsystem specifies a memory section for execution functions that overrides that of the parent model, subsystem code looks like this:

```
/* File: mySubsystem.c */

#pragma DATA_SEC(mySubsystem,  FAST_RAM )
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
  localB->Gain = rtu_In1 * localP->Gain_Gain;
}
```

If the subsystem had not defined its own memory section for execution functions, but inherited that of the parent model, the subsystem code would have looked like this:

```
/* File: mySubsystem.c */
```

```
#pragma ExecFunc-begin(mySubsystem)
void mySubsystem(real_T rtu_In1,
rtB_mySubsystem *localB,
rtP_mySubsystem *localP)
{
  localB->Gain = rtu_In1 * localP->Gain_Gain;
}
#pragma ExecFunc-end(mySubsystem)
```

Memory Section Limitation

Memory sections cannot be applied to shared utility functions, such as lookup table functions, data type conversion functions, and fixed-point functions. For information about shared utilities, see “Setting Up Runtime Logging to MAT-Files”, “Supporting Shared Utility Folders in the Build Process”, and “Supporting the Shared Utilities Folder” in the Simulink Coder documentation.

10

Optimizing Buses for Code Generation

- “Introduction” on page 10-2
- “Setting Bus Diagnostics” on page 10-3
- “Optimizing Virtual and Nonvirtual Buses” on page 10-4
- “Using Single-Rate and Multi-Rate Buses” on page 10-7
- “Setting Bus Signal Initial Values” on page 10-11
- “Buses and Atomic Subsystems” on page 10-16

Introduction

When you use buses in a model for which you intend to generate code:

- Setting appropriate diagnostic configuration parameters can add to the ease of development.
- The bus implementation techniques used can affect the speed, size, and clarity of that code.
- Some bus implementation techniques that can be useful are not immediately obvious.

This chapter contains guidelines that you can use to improve the results when you work with buses. The guidelines describe techniques for:

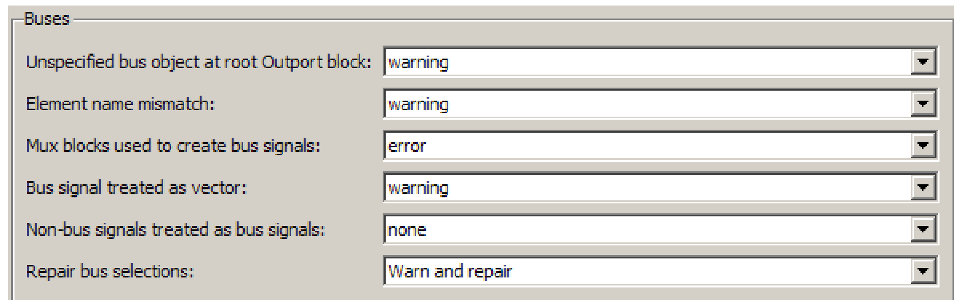
- Simplifying the layout of the model
- Increasing the efficiency of generated code
- Defining data structures for function/subsystem interfaces
- Defining data structures that match existing data structures in external C code

Some trade-offs inevitably exist among speed, size, and clarity. For example, the code for nonvirtual buses is easier to read because the buses appear in the code as structures, but the code for virtual buses is faster because virtual buses do not require copying signal data. The applicability of some guidelines can therefore depend on where you are in the application development process.

This chapter focuses on optimizations that are appropriate for final production code. Before you read this chapter, be sure that you have read "Using Composite Signals". This chapter assumes that you understand all the concepts and procedures described in that one, including the blocks used for creating and manipulating buses.

Setting Bus Diagnostics

Simulink provides diagnostics that you can use to optimize bus usage. MathWorks recommends setting the following values on the **Configuration parameters > Diagnostics > Connectivity** pane:



Bus signal treated as vector is enabled only when **Mux blocks used to create bus signals** is set to `error`. Setting **Mux blocks used to create bus signals** to `none` disables both diagnostics. Temporarily disabling the two mux and bus diagnostics allows you to debug other bus problems before addressing mux and bus mixtures. You can then enable the last two diagnostics and use them to eliminate any such mixtures. When you build existing models, the diagnostic settings should be as shown at all times. See "Avoiding Mux/Bus Mixtures" for more information.

Optimizing Virtual and Nonvirtual Buses

In this section...

"Use Virtual Buses Wherever Possible" on page 10-4

"Avoid Nonlocal Nested Buses in Nonvirtual Buses" on page 10-5

Use Virtual Buses Wherever Possible

Virtual buses are graphical conveniences that do not affect generated code. As a result, the code generation engine is able to fully optimize the signals in the bus. You should therefore use virtual rather than nonvirtual buses wherever possible. You can convert between virtual and nonvirtual buses as needed using Signal Conversion blocks. In many cases, Simulink automatically converts a virtual bus to a nonvirtual bus when required. For example, a virtual bus input to a Model block becomes a nonvirtual bus with no need for explicit conversion. See for more information.

When are Virtual and Nonvirtual Buses Required?

In some cases, Simulink requires the use of nonvirtual buses:

- For non-auto storage classes
- Inports and Outports of Model blocks
- To generate a specific structure from the bus
- Root level Inport or Outport blocks when the bus has mixed data types

In one case, Simulink requires the use of virtual buses:

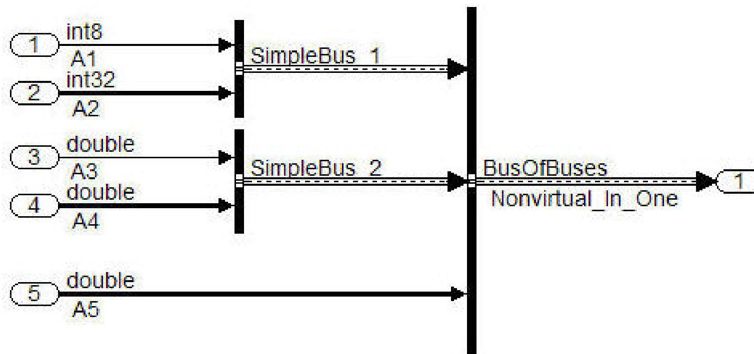
- Only virtual buses can be used for bundling function call signals.

Avoid Nonlocal Nested Buses in Nonvirtual Buses

Buses can contain subordinate buses. The storage class of any subordinate bus should be `auto`, which results in a local signal. Setting a subordinate bus to a non-`auto` storage class has two undesirable results:

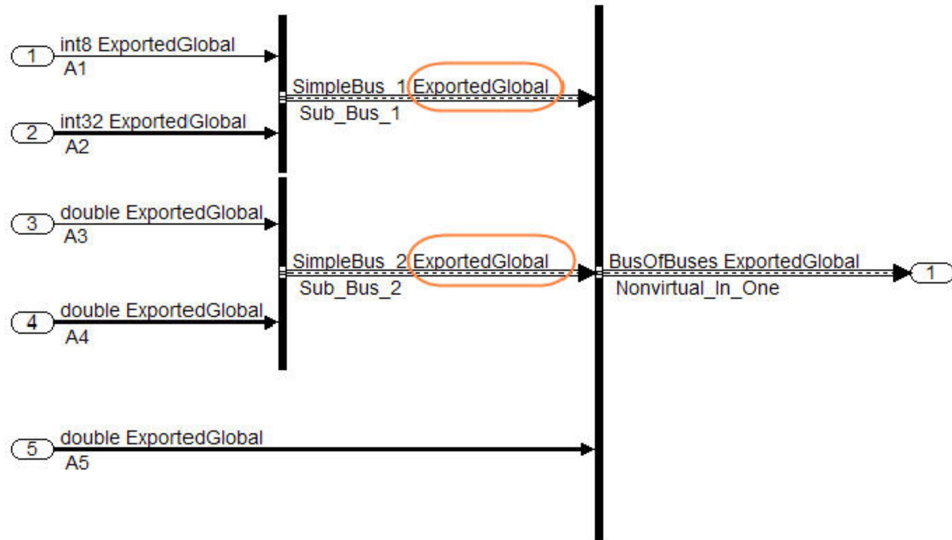
- Allocation of redundant memory (memory for the subordinate bus object and memory for the final bus object)
- Additional copy operations (first copying to the subordinate bus and then copying from the subordinate bus to the final bus)

In the following example, the final bus is created from local scoped subordinate elements. The resulting assignment operations are relatively efficient:




```
34  void bus_in_steps_a_step(void)
35  {
36    Nonvirtual_In_One.Simp_1.enableFlag = A1;
37    Nonvirtual_In_One.Simp_1.calValues[0] = A2[0];
38    Nonvirtual_In_One.Simp_1.calValues[1] = A2[1];
39    Nonvirtual_In_One.Simp_2.Entry_1 = A3;
40    Nonvirtual_In_One.Simp_2.Entry_2_Array[0] = A4[0];
41    Nonvirtual_In_One.Simp_2.Entry_2_Array[1] = A4[1];
42    Nonvirtual_In_One.A_Vector[0] = A5[0];
43    Nonvirtual_In_One.A_Vector[1] = A5[1];
44    Nonvirtual_In_One.A_Vector[2] = A5[2];
45  }
```

By contrast in the next example the subordinate elements `sub_bus_1` and `sub_bus_2` are global in scope. First the assignment to the subordinate bus occurs (lines 54 – 59) then the copy of the subordinate bus to the main bus (lines 60 – 61). In most cases, this is not an efficient implementation:




```
52    void bus_in_steps_b_step(void)
53    {
54      Sub_bus_1.enableFlag = A1;
55      Sub_bus_2.Entry_1 = A3;
56      Sub_bus_1.calValues[0] = A2[0];
57      Sub_bus_2.Entry_2_Array[0] = A4[0];
58      Sub_bus_1.calValues[1] = A2[1];
59      Sub_bus_2.Entry_2_Array[1] = A4[1];
60      Nonvirtual_In_Steps.Simp_1 = Sub_bus_1;
61      Nonvirtual_In_Steps.Simp_2 = Sub_bus_2;
62      Nonvirtual_In_Steps.A_Vector[0] = A5[0];
63      Nonvirtual_In_Steps.A_Vector[1] = A5[1];
64      Nonvirtual_In_Steps.A_Vector[2] = A5[2];
65    }
```

Using Single-Rate and Multi-Rate Buses

In this section...

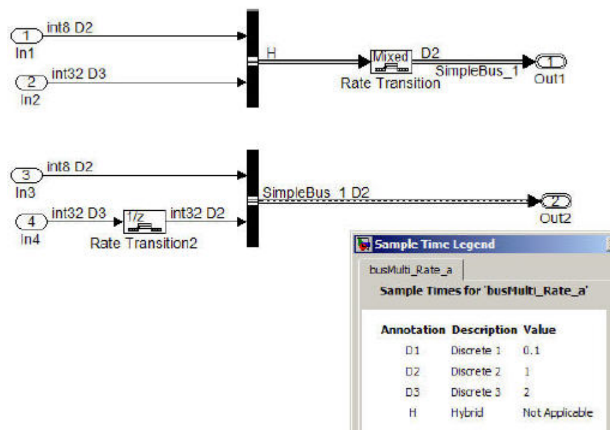
"Introduction" on page 10-7
"Techniques for Combining Multiple Rates" on page 10-7
"Larger Buses and Multiple Rates" on page 10-9
"Specifying Sample Time Rates" on page 10-10

Introduction

Nonvirtual buses do not support multiple rates. Virtual buses support multiple rates as long as the bus does not cross any root level inport or outport. The best techniques for optimizing a bus that contains signals that initially have different rates can depend on the type of the bus and the number of signals.

Techniques for Combining Multiple Rates

The simplest bus contains only two signals. The next figure shows two examples of two-element buses. The first example shows a virtual bus created from two signals that have different rates. The second example shows a nonvirtual bus created from the same two signals. The Sample Time Legend shows the different signal rates:

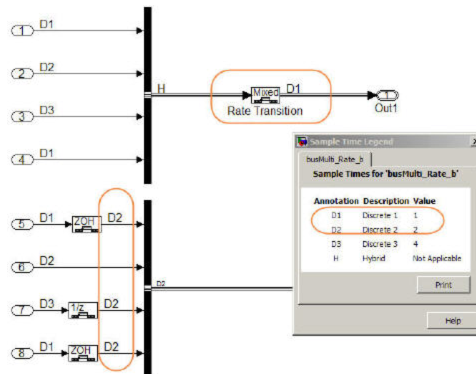


The signals with different rates in the first example can be combined into a virtual bus, because virtual buses support multiple rates. However, a multirate virtual bus cannot connect to a root-level output port. The bus therefore passes through a Rate Transition block that converts it to a single-rate bus, then connects to the Outport. This technique is preferable only for virtual buses that contain no more than two signals. See “Larger Buses and Multiple Rates” on page 10-9.

The signals with different rates in the second example cannot initially be combined into a nonvirtual bus, because nonvirtual buses do not support multiple rates. One of the signals therefore passes through a Rate Transition block, which converts it to have the same rate as the other signal, then connects to the Bus Creator block. The signals can then combine into a single-rate nonvirtual bus, which can connect to the root-level outport without further conversion.

Larger Buses and Multiple Rates

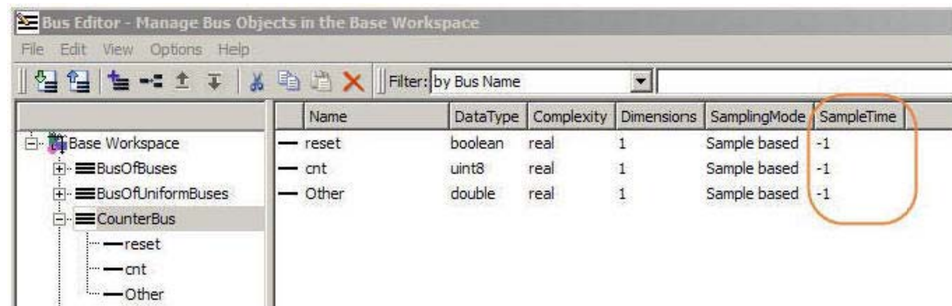
When you create a multirate virtual bus that contains more than two signals, you can convert the bus to single-rate by applying a Rate Transition block to the output of the Bus Creator block. However, MathWorks recommends using a Rate Transition block on each input signal to give full control over the output rate. As the next figure shows, when a single Rate Transition block is used the block sets all of the signals to the fastest rate (D1):



Note that the preferred techniques for a virtual bus with more than two signals, and the required technique for a nonvirtual bus with any number of signals, are the same. Note also that, in the preceding figure, the blocks that perform rate transition are not actual Rate Transition blocks, but other blocks that can change the signal rate as part of some other operation. The identity of the blocks that perform rate transition makes no difference. All that matters is that the signal rates match when required.

Specifying Sample Time Rates

The sample time for buses should be specified through the signals that define the bus. If the sample times do not match, use Rate Transition (or equivalent) blocks to create a uniform rate, as shown in the previous figures. The signal rates should *not* be set by specifying **Sample Time** values in a Bus Creator block's bus object. Instead, set the sample time for each signal before inputting it to the Bus Creator, and set each **Sample Time** in the corresponding bus object to -1 (inherited), as shown in the next figure:



Setting Bus Signal Initial Values

In this section...

"Introduction" on page 10-11

"Initializing Bus Signals in Simulink" on page 10-11

"Bus Initialization in Stateflow" on page 10-12

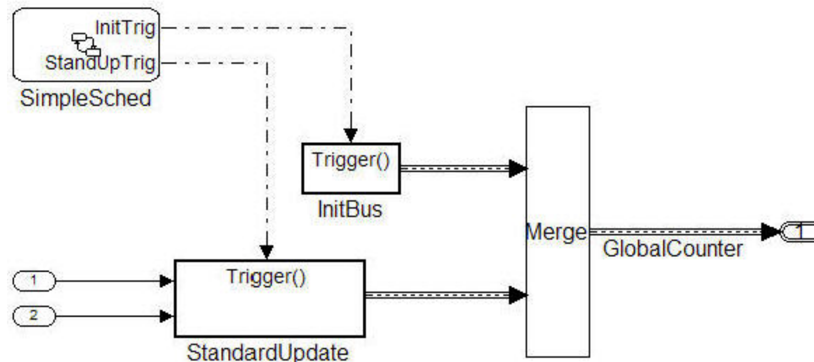
"Creating a Bus of Constants" on page 10-14

Introduction

Unlike scalar and vector signals, buses provide no direct way to initialize signals. This section describes techniques for initializing bus signals using Simulink, Stateflow, and MATLAB functions.

Initializing Bus Signals in Simulink

In Simulink, you can set initial values on a bus by using a set of conditionally executed subsystems, such as Function-Call subsystems, and a Merge block, as shown in this example:

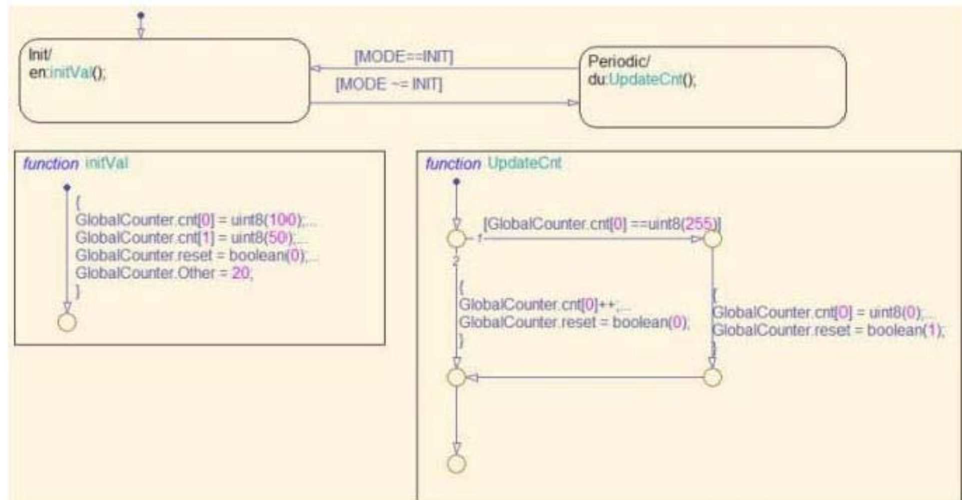


Both subsystems (`InitBus` and `StandardUpdate`) create a bus signal of type `CounterBus`. However, the assignment to the variable `GlobalCounter` is controlled by the Merge block. See "Function-Call Subsystems" for more information.

This technique is limited because the `StandardUpdate` subsystem does not use the initial values from the `InitBus` subsystem. If the calculations depend on past information from the bus, consider using Stateflow or MATLAB functions to initialize bus signals.

Bus Initialization in Stateflow

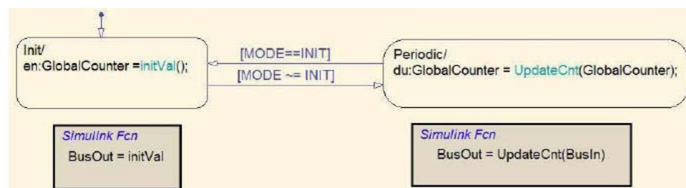
Stateflow and MATLAB functions allow for conditional execution internally. In the following example, the `init` and `update` code are Functions in the Stateflow diagram. This technique simplifies the presentation in the generated code:



In the generated code, you can see that the `UpdateCnt` function uses the past value of `GlobalCounter.cnt`:

```
static void initBus_4_Stateflow_Arr_initVal(void)
{
  GlobalCounter.cnt[0] = 100U;
  GlobalCounter.cnt[1] = 50U;
  GlobalCounter.reset = false;
  GlobalCounter.Other = 20.0;
}

static void initBus_4_Stateflow_A_UpdateCnt(void)
{
  if (GlobalCounter.cnt[0] == 255) {
    GlobalCounter.cnt[0] = 0U;
    GlobalCounter.reset = true;
  } else {
    GlobalCounter.cnt[0] = (uint8_T)(GlobalCounter.cnt[0] + 1);
    GlobalCounter.reset = false;
  }
}
```

The previous example used Stateflow Graphical functions to initialize and update the buses. Alternatively, you can use MATLAB functions or Simulink subsystems embedded in a Stateflow diagram. The next figure illustrates this technique:



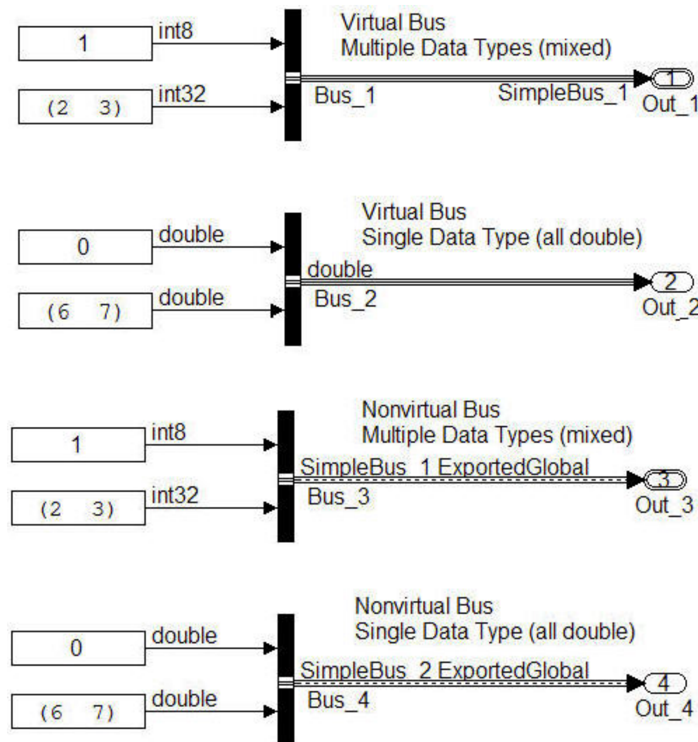
The Simulink subsystems are the same as those used in the earlier Simulink-only example.

Creating a Bus of Constants

The code for specifying a bus of constant values will appear in either the `Init` or the `Step` function of the model. The code location depends on the configuration of the bus. In most cases the code appears in the `Step` function. However if the following conditions hold the code will be placed in the `Init` function:

- The bus is a virtual bus
- All signals have the same data type
- The signals in the bus are all constants

In the next figure, only the bus named `Bus_2` meets all the requirements:



The code for `Bus_2` therefore appears in the `Init` function. The code for the other buses appears in the `Step` function:

```
SimpleBus_2 Bus_4;
SimpleBus_1 Bus_3;
ExternalOutputs_busOfConstants_ busOfConstants_A_Y;
RT_MODEL_busOfConstants_A busOfConstants_A_M_;
RT_MODEL_busOfConstants_A *busOfConstants_A_M = &busOfConstants_A_M_;
void busOfConstants_A_step(void)
{
  busOfConstants_A_Y.Out_1.enableFlag = 1;
  Bus_3.enableFlag = 1;
  Bus_4.Entry_1 = 0.0;
  busOfConstants_A_Y.Out_1.calValues[0] = 2;
  Bus_3.calValues[0] = 2;
  Bus_4.Entry_2_Array[0] = 6.0;
  busOfConstants_A_Y.Out_1.calValues[1] = 3;
  Bus_3.calValues[1] = 3;
  Bus_4.Entry_2_Array[1] = 7.0;
}

void busOfConstants_A_initialize(void)
{
  busOfConstants_A_Y.Out_2[0] = 0.0;
  busOfConstants_A_Y.Out_2[1] = 6.0;
  busOfConstants_A_Y.Out_2[2] = 7.0;
}
```

To avoid repeatedly updating a bus of constants, place the bus code into a function-call subsystem, as described in "Initializing Bus Signals in Simulink" on page 10-11. When you use this technique, make sure the function-call subsystem is called at the start of execution. See "Function-Call Subsystems" for more information.

Buses and Atomic Subsystems

In this section...

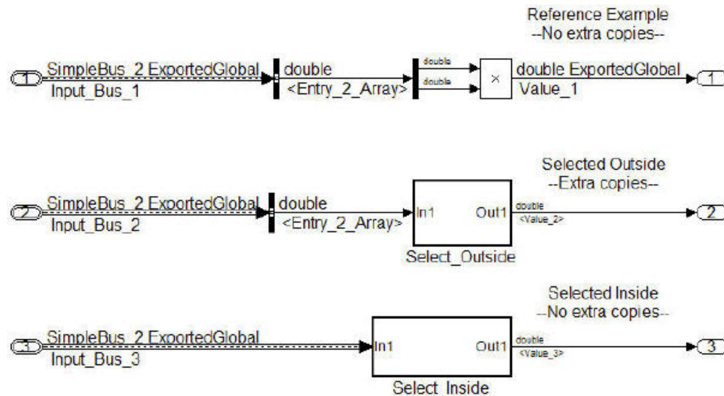
"Extract Nonvirtual Bus Signals Inside of Atomic Subsystems" on page 10-16

"Virtual Bus Signals Crossing Atomic Boundaries" on page 10-17

"Atomic Subsystems and Buses of Constants" on page 10-19

Extract Nonvirtual Bus Signals Inside of Atomic Subsystems

Selecting signals from a nonvirtual bus can result in unnecessary data copies when those signals cross an atomic boundary. In the following example the same code, a simple multiplication of two elements in a vector, is executed three times:

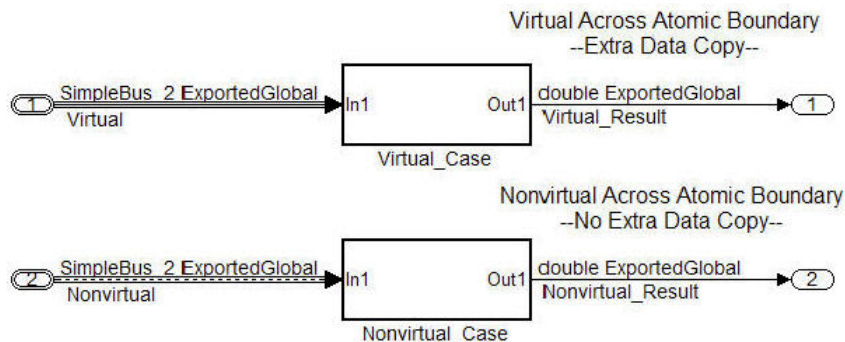


In the second instance when the bus signals are selected outside of the atomic subsystem an unnecessary copy of the bus data is created.

Although this example shows only signals with global scope, both global and local signals show the same behavior: the selection of the signals outside of the model results in an unnecessary copy, while the internal selection does not.

Virtual Bus Signals Crossing Atomic Boundaries

Virtual buses that cross atomic boundaries can result in the creation of unnecessary data copies. The following example shows the data copy that occurs when a virtual bus crosses an atomic boundary:

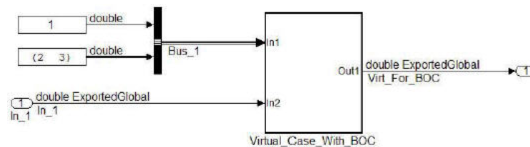


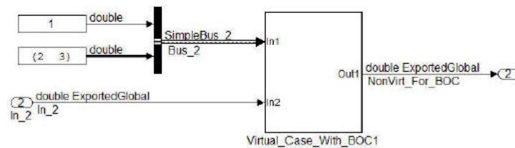

```
12   void virtualAcrossBo_Nonvirtual_Case(void)
13   {
14     Nonvirtual_Result = Nonvirtual.Entry_2_Array[0] * Nonvirtual.Entry_2_Array[1];
15   }
16
17   void virtualAcrossBound_Virtual_Case(void)
18   {
19     Virtual_Result = virtualAcrossBoundary_B.Entry_2_Array[0] *
20       virtualAcrossBoundary_B.Entry_2_Array[1];
21   }
22
23   void virtualAcrossBoundary_step(void)
24   {
25     virtualAcrossBoundary_B.Entry_2_Array[0] = Virtual.Entry_2_Array[0];
26     virtualAcrossBoundary_B.Entry_2_Array[1] = Virtual.Entry_2_Array[1];
27     virtualAcrossBound_Virtual_Case();
28     virtualAcrossBo_Nonvirtual_Case();
29   }
```

Lines 25–26 show the signals being selected out of the bus before they are used in the function on lines 19–20. By comparison the nonvirtual bus does not require the use of temporary variables.

Atomic Subsystems and Buses of Constants

If the bus passed into an atomic subsystem consists exclusively of constants, using a virtual bus is more efficient, because Simulink is able to inline the constant values into the code:






```
void virtualAc_Virtual_Case_With_BOC(void)
{
  Virt_For_BOC = 6.0 * In_1;
}

void virtualA_Virtual_Case_With_BOC1(void)
{
  NonVirt_For_BOC = virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[0] *
    virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[1] * In_2;
}

void virtualAcrossBoundaryBOC_step(void)
{
  virtualAc_Virtual_Case_With_BOC();
  virtualAcrossBoundaryBOC_B.Bus_2.Entry_1 = 1.0;
  virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[0] = 2.0;
  virtualAcrossBoundaryBOC_B.Bus_2.Entry_2_Array[1] = 3.0;
  virtualA_Virtual_Case_With_BOC1();
}
```

11

Renaming and Replacing Data Types

- “Defining Application-Specific Data Types Based On Built-In Types” on page 11-2
- “Code Generation with User-Defined Data Types” on page 11-4

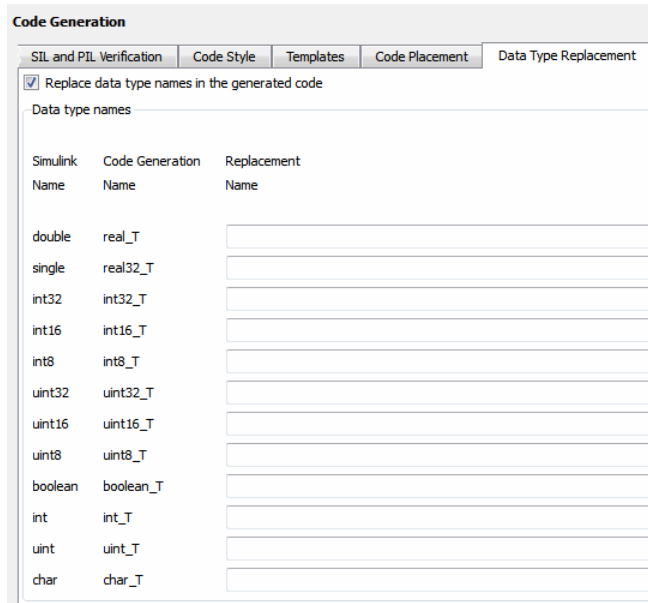
Defining Application-Specific Data Types Based On Built-In Types

You can replace built-in data type names with user-defined replacement data type names in the generated code for a model.

To configure replacement data types,

1 For a referenced model, set the **Simulation mode** parameter for the corresponding Model block to `Normal`, `Software-in-the-loop (SIL)`, or `Processor-in-the-loop (PIL)`.

2 In the Configuration Parameters dialog box, click **Code Generation > Data Type Replacement > Replace data type names in the generated code**. A **Data type names** table appears. The table lists each Simulink built-in data type name with its corresponding code generation data type name.



3 Fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type.

   For each replacement data type you enter, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.

   - For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, and `uint8`, the replacement data type's `BaseType` must match the built-in data type.
   - For `boolean`, the replacement data type's `BaseType` must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.
   - For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

   An error occurs if a replacement data type specification is inconsistent. For more information, see "Replacing Built-In Data Type Names in Generated Code" on page 12-52.

Code Generation with User-Defined Data Types

In this section...

"Overview" on page 11-4

"Specifying Type Definition Location for User-Defined Data Types" on page 11-5

"Using User-Defined Data Types for Code Generation" on page 11-6

Overview

The Embedded Coder software supports use of user-defined data type objects in code generation. These include objects of the following classes:

- `Simulink.AliasType`
- `Simulink.Bus`
- `Simulink.NumericType`
- `Simulink.StructType`

For information on the properties and usage of these data object classes, see `Simulink.AliasType`, `Simulink.Bus`, `Simulink.NumericType`, and `Simulink.StructType` in the "Simulink Classes" section of the Simulink Reference documentation. For general information on creating and using data objects, see the "Working with Data Objects" section of the Simulink documentation

In code generation, you can use user-defined data objects to

- Map your own data type definitions to Simulink built-in data types, and specify that your data types are to be used in generated code.
- Optionally, generate `#include` directives specifying your own header files, containing your data type definitions. This technique lets you use legacy data types in generated code.

In general, code generated from user-defined data objects conforms to the properties and attributes of the objects as defined for use in simulation. When generating code from user-defined data objects, the name of the object

is the name of the data type that is used in the generated code. Exception: for `Simulink.NumericType` objects whose `IsAlias` property is false, the name of the functionally equivalent built-in or fixed-point Simulink data type is used instead.

Note The names of data types defined using `Simulink.AliasType` objects are preserved in the generated code only for installations with a Embedded Coder license.

Specifying Type Definition Location for User-Defined Data Types

When a model uses `Simulink.DataType` and `Simulink.Bus` objects, corresponding `typedefs` are needed in code. Both `Simulink.DataType` and `Simulink.Bus` objects have a `HeaderFile` property that controls the location of the object's `typedef`. Setting a `HeaderFile` is optional and affects code generation only.

Omitting a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is left empty, a generated `typedef` for the object appears in the generated file *`model`*`_types.h`.

Example. For a `Simulink.NumericType` object named `myfloat` with a `Category` of `double` and no `HeaderFile` property specified, *`model`*`_types.h` in the generated code contains:

```
typedef real_T myfloat;
```

Specifying a HeaderFile Value

If the `HeaderFile` property for a `Simulink.DataType` or `Simulink.Bus` object is set to a string value,

- The string must be the name of a header file that contains a `typedef` for the object.

- The generated file *model*_types.h contains a #include that gives the header file name.

You can use this technique to include legacy or other externally created typedefs in generated code. When the generated code compiles, the specified header file must be accessible on the build process include path.

HeaderFile Property Syntax. The HeaderFile property should include the desired preprocessor delimiter ("" or '<>'), as in the following examples.

This example:

```
myfloat.HeaderFile = '<legacy_types.h>'
```

generates the directive:

```
#include <legacy_types.h>
```

This example:

```
myfloat.HeaderFile = '"legacy_types.h>"'
```

generates the directive:

```
#include "legacy_types.h"
```

Using User-Defined Data Types for Code Generation

To specify and use user-defined data types for code generation:

1. Create a user-defined data object and configure its properties, as described in the “Working with Data Objects” section of the Simulink documentation.

2. If you specified the HeaderFile property, copy the header file to the appropriate directory.

3. Set the output data type of selected blocks in your model to the user-defined data object. To do this, set the **Data type** parameter of the block to Specify with dialog. Then, enter the object name in the **Output data type** parameter.

4. The specified output data type propagates through the model and variables of the user-defined type are declared as required in the generated code.

12

Managing Data Definitions and Declarations With the Data Dictionary

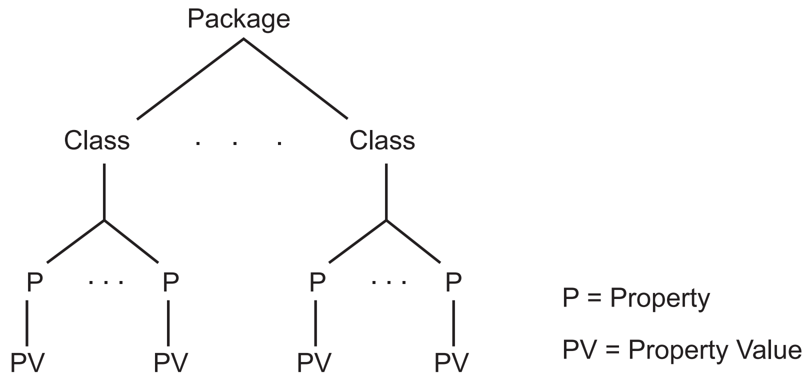
- “Overview of the Data Dictionary” on page 12-2
- “Creating Simulink and mpt Data Objects” on page 12-4
- “Creating a Data Dictionary for a Model” on page 12-19
- “Defining All Global Data Objects in a Separate File” on page 12-26
- “Defining a Specific Global Data Object in Its Own File” on page 12-28
- “Saving and Loading Data Objects” on page 12-29
- “Applying Naming Rules to Identifiers Globally” on page 12-30
- “Creating User Data Types” on page 12-38
- “Selecting User Data Types for Signals and Parameters” on page 12-43
- “Registering mpt User Object Types” on page 12-48
- “Replacing Built-In Data Type Names in Generated Code” on page 12-52
- “Customizing Data Object Wizard User Packages” on page 12-60

Overview of the Data Dictionary

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

The values of data object properties can affect where the code generator places a parameter or signal in the generated file. This is because some property values are associated with different template symbols. The location of a symbol in a template determines where the associated parameter or signal is located in the generated file. For details about templates and symbols, see "Configuring Templates for Customizing Code Organization and Format" on page 17-23.

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



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

Creating Simulink and mpt Data Objects

In this section...

"Overview" on page 12-4
"Creating Simulink Data Objects with Data Object Wizard" on page 12-5
"Creating mpt Data Objects with Data Object Wizard" on page 12-11
"Comparing Simulink and mpt Data Objects" on page 12-12
"Creating Data Objects Based on an External Data Dictionary" on page 12-16

Overview

The Embedded Coder software provides the `mpt` (module packaging tool) data object, which contains all the properties of Simulink data objects plus properties that provide additional control over module packaging. For a comparison of the properties of Simulink and `mpt` data objects, see "Comparing Simulink and mpt Data Objects" on page 12-12.

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

- One-by-one, either using the MATLAB command line or using the Model Explorer **Add** menu and selecting **Simulink Parameter**, **Simulink Signal**, **MPT Parameter**, or **MPT Signal**. For more information, see "Working with Data Objects" in the Simulink documentation.
- All at once, invoking Data Object Wizard for an existing model. For more information and examples, see "Data Object Wizard" in the Simulink documentation and "Creating mpt Data Objects with Data Object Wizard" on page 12-11.
- Creating data objects based on an external data dictionary. You can do this manually item by item, or all at once automatically using a script. For more information, see "Creating Data Objects Based on an External Data Dictionary" on page 12-16.

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

Creating Simulink Data Objects with Data Object Wizard

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

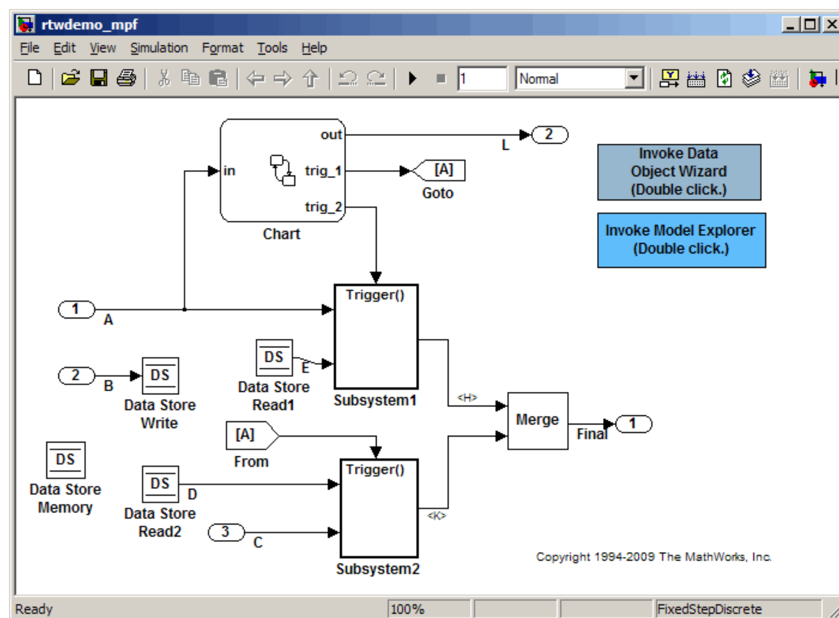
Data Object Wizard is especially useful for creating multiple data objects for

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

Creating Simulink Data Objects

This procedure creates Simulink data objects using Data Object Wizard.

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



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

The **Model name** field displays the name of the model. You could specify a different model by editing the field or by selecting the model using the adjacent **Browse** button. When the **Model name** field is nonempty, the **Find** button is enabled.

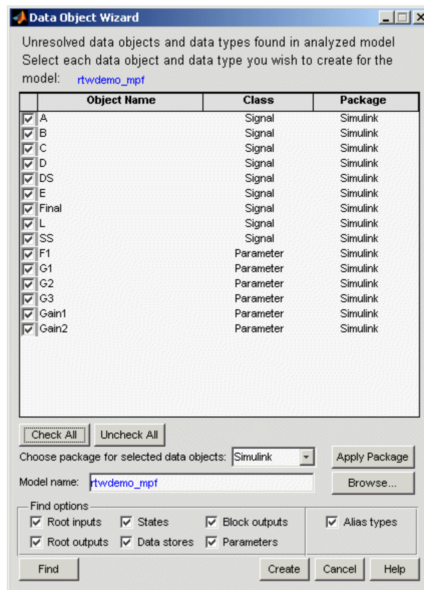
3 In the **Find options** pane, select the desired check boxes. For descriptions of each check box, see "Data Object Wizard" in the Simulink documentation. Be sure to check the **Alias types** option. This finds all user-registered data types in the `sl_customization.m` file plus all data type replacements specified for the model in the **Data Type Replacement** pane of the Configuration Parameters dialog box. Data Object Wizard can create `Simulink.AliasType` objects from these.

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

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

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

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



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

7 Click **Cancel**. The Data Object Wizard dialog box disappears.

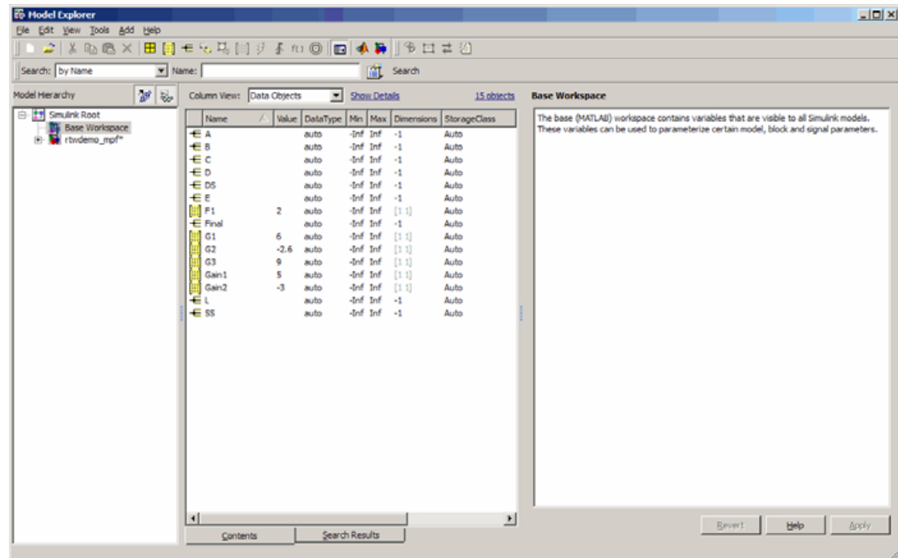
Now you can set property values for the data objects.

Setting Property Values for Simulink Data Objects

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

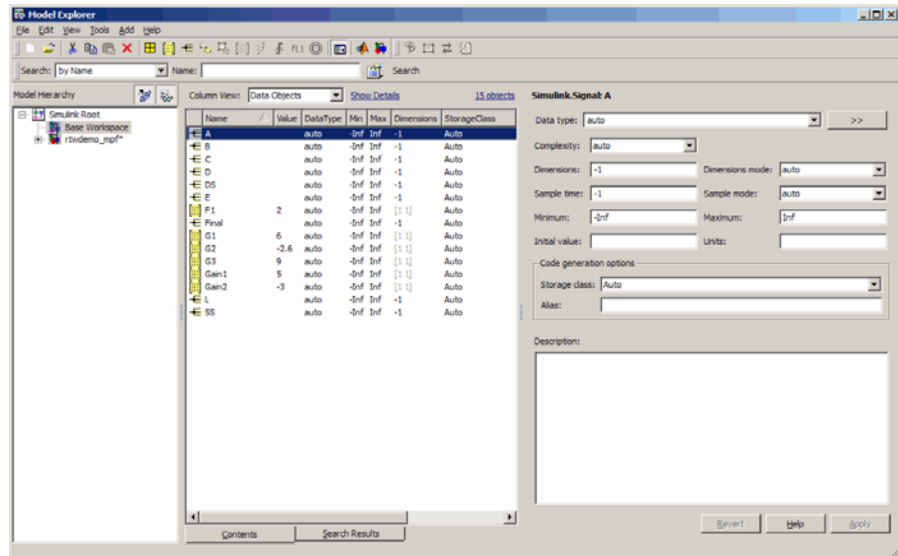
1 Type `daexplr` on the MATLAB command line, and press **Enter**. The Model Explorer appears.

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

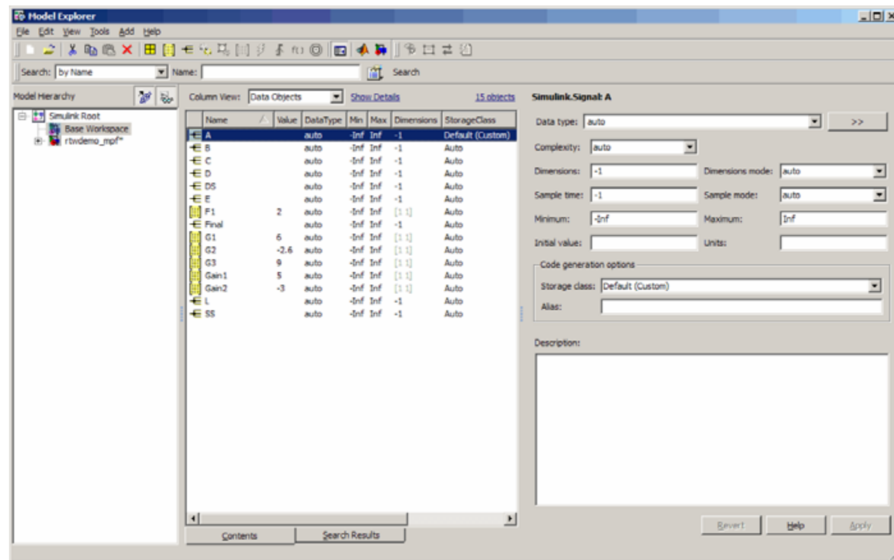


If the objects that you see do not appear in the order shown, click the **Name** column header in the middle pane to sort the objects in ascending order by name.

3 To see the properties of a Simulink data object, select a data object in the middle pane. The right pane displays the property names for that object. These property names also appear as column headings in the middle pane. Reshape the middle pane as needed to see all the headings. For example, if you select signal data object A in the middle pane, the Model Explorer looks like this:



4 You can change the values specified for the properties of the selected object. For example, with `A` selected, change its **StorageClass** property from `Auto` to `Default (Custom)`, then click **Apply**. The property changes as shown below:



You can use Control-Right-Click to select multiple objects in the center pane, then edit any property value. The wizard applies the new value to all selected objects. For descriptions of object properties and their values, see Parameter and Signal Property Values on page 7-2.

Generating and Inspecting Code

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

1 In the Configuration Parameters dialog box, click **Code Generation** in the left pane.

2 In the **Report** pane, select the **Create code generation report** check box.

Note When you select the **Create code generation report** check box, the code generation software automatically selects two check boxes on the pane: **Launch report automatically** and **Code-to-model**. For large models, you may find that HTML report generation (step 4 below) takes longer than you want. In this case, consider clearing the **Code-to-model** check box (and the **Model-to-code** check box if selected). The report will be generated faster.

---

3 In the **Code Generation** pane, select the **Generate code only** check box. The **Build** button changes to **Generate code**.

Note The generate code process generates the `.c/.cpp` and `.h` files. The build process adds compiling and linking to generate the executable. For details on build, see “Understanding the Build Process” in the Simulink Coder documentation.

---

4 Click the **Generate code** button. After a moment, the HTML code generation report appears, listing the generated files on the left pane.

5 Select and review files in the HTML code generation report. See “Generating an HTML Code Generation Report” on page 20-4 for more information.

Creating mpt Data Objects with Data Object Wizard

Create `mpt` data objects using Data Object Wizard the same way you did for Simulink data objects, as explained in “Creating Simulink Data Objects with Data Object Wizard” on page 12-5, except select `mpt` as the package instead of `Simulink`.

Set the property values for the `mpt` data objects the same way you set them for Simulink data objects, as explained in “Setting Property Values for Simulink Data Objects” on page 12-8, with the following exceptions:

- Accept the default custom storage class for `mpt` data objects, `Global(Custom)`

- For data objects `A` and `F1`, type `mydefinitionfile` in the **Definition file** field on the Model Explorer.

Then generate and inspect the code.

Note The **Alias** field is related to "Applying Naming Rules to Identifiers Globally" on page 12-30.

Comparing Simulink and mpt Data Objects

The `mpt` data object contains all the properties of Simulink data objects plus properties that provide additional control over module packaging. The differences between Simulink and `mpt` data objects can be illustrated by comparing

- "Signal and Parameter Properties" on page 12-13
- "Configuration Parameters" on page 12-14
- "Generated Code" on page 12-15

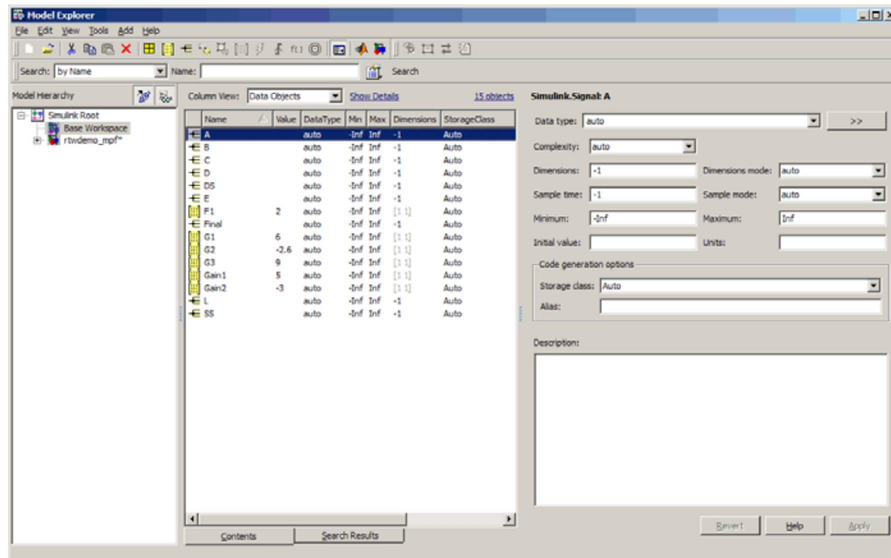
Key differences include the following:

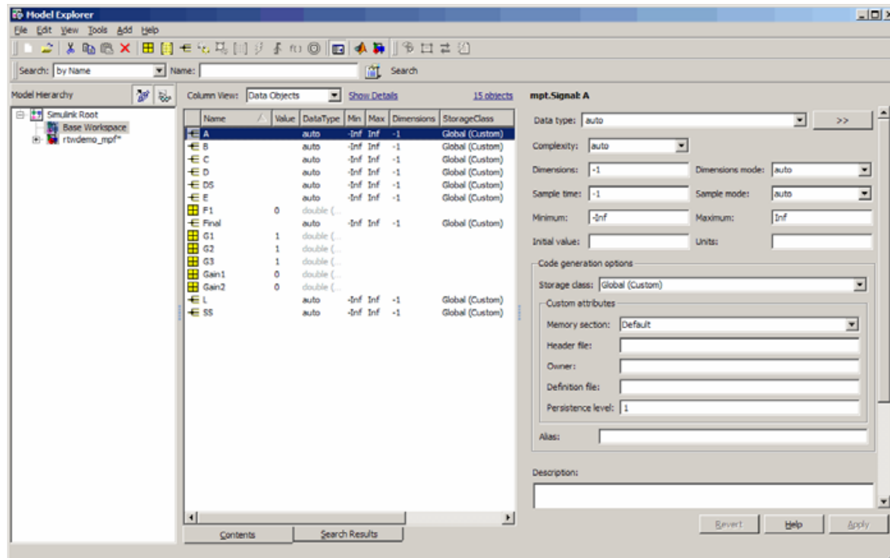
- Different custom storage classes displayed in the Model Explorer for `mpt` data objects provide more control over the appearance of the generated code.
- Additional custom attributes (owner, definition file, persistence level, memory section) for `mpt` data objects provide more control over data packaging in the generated code.
- On the **Comments** pane of the Configuration Parameters dialog box, the **Custom comments (MPT objects only)** option allows you to add a comment just above a signal or parameter's identifier in the generated code.
- On the **Data Placement** pane of the Configuration Parameters dialog box, in the **Global data placement (MPT data objects only)** subpane:
  - The **Module naming** parameter allows you to name the module that owns the model
  - The **Signal display level** parameter allows you to specify whether or not the code generator declares a signal data object as global data

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

Signal and Parameter Properties

The properties that appear in Model Explorer when `mpt` is the package include all the properties that appear when `Simulink` is the package plus additional properties. Notice this by comparing the next two figures. (For descriptions of all properties in Model Explorer, see Parameter and Signal Property Values on page 7-2.)





Configuration Parameters

The following configuration parameters relate to module packaging features. These parameters are available in the Configuration Parameters dialog box and Model Explorer when the system target file selected for a Simulink model is `ert.tlc` (or a system target file derived from an `ert.tlc`):

- **Custom comments (MPT objects only)** option on the **Code Generation > Comments** pane
- In the **Global data placement (MPT data objects only)** subpane on the **Code Generation > Data Placement** pane:
  - **Module naming** parameter
  - **Signal display level** parameter
  - **Parameter tune level** parameter

Generated Code

In the example used in "Setting Property Values for Simulink Data Objects" on page 12-8, you selected `Default (Custom)` in the **Storage class** field for signal `A` and parameter `F1`. You selected the default `Auto` in the **Storage class** field for the remaining data objects. But for the `mpt` data objects you used the default `Global (Custom)` in the **Storage class** field for all data objects. When you generated code, these selections resulted in the definitions and declarations shown in the table below.

<table>
<tr><th>Simulink Data Object with Auto Storage Class</th><th>Simulink Data Object with Default (Custom) Storage Class</th><th>mpt Data Object with Global (Custom) Storage Class and Definition File Named mydefinitionfile</th></tr>
<tr><td>In rtwdemo_mpf.c:

/* For signal A */
ExternalInputs rtU;

/* For parameter F1 */
if(rtU.A * 2.0 > 10.0) {...

In rtwdemo_mpf.h:

/* For signal A */
typedef struct {
real_T A;
} ExternalInputs;

extern ExternalInputs rtU;</td><td>In global.c:

real_T A;
real_T F1 = 2.0;

In global.h:

extern real_T A;
extern real_T F1;</td><td>In mydefinitionfile.c:

real_T A;
real_T F1 = 2.0;

In global.h:

extern real_T A;
extern real_T F1;</td></tr>
</table>

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

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

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

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

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

For more details, see "Introduction to Custom Storage Classes" on page 8-2 and "Summary of Signal Storage Class Options" in the Simulink Coder documentation.

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

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

Creating Data Objects Based on an External Data Dictionary

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

Manually Creating Objects to Represent External Data

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

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

2. Determine all of the data in this file that correspond to the parameters and signals in the model. In the code generation data dictionary, parameters in the external file belong to the Simulink parameter class and signals belong to the Simulink signal class.

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

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

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

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

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

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

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

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

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

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

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

Automatically Creating Objects to Represent External Data

You can create data objects (and their properties) all at once, based on an external data dictionary by creating and running a `.m` file. This file contains the same MATLAB commands you could use for creating data objects one-by-one on the command line, as explained in "Working with Data Objects" in the Simulink documentation. But instead of using the command line, you place the MATLAB commands in the `.m` file for *all* of the desired data in the external file:

1. Create a new `.m` file.

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

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

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

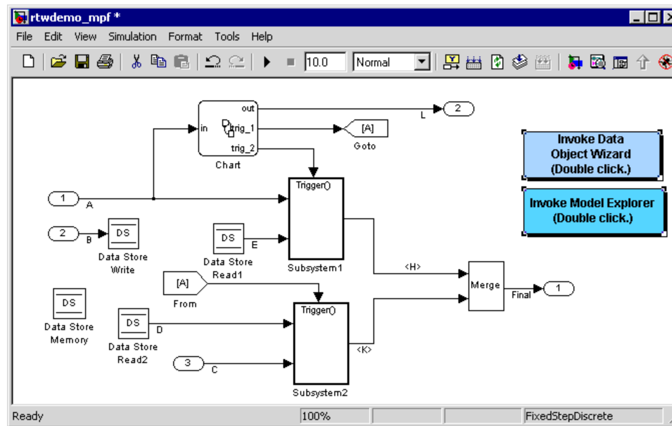
Note If you want to import data from an external data dictionary, you can write functions that read the information, convert these to data objects, and load them into the MATLAB workspace. Among available MATLAB functions that you can use for this process are `xmlread`, `xmlwrite`, `xlsread`, `xlswrite`,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

Creating a Data Dictionary for a Model

In this procedure, you create a data dictionary for a model using Data Object Wizard, inspect the data dictionary, and generate code. Definitions for the data objects in the dictionary are generated into the model source file (`model.c`).

Using Data Object Wizard

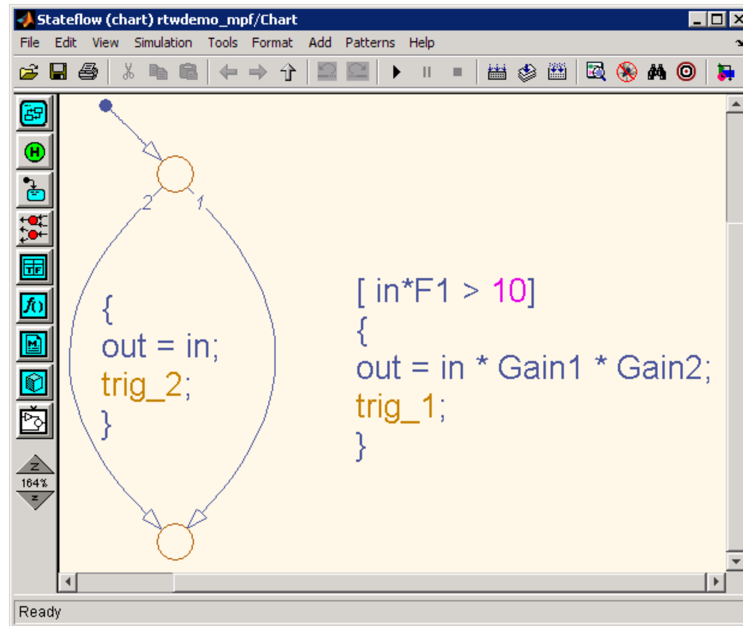
1 Open the demo model `rtwdemo_mpf` by clicking the link or by typing `rtwdemo_mpf` in the MATLAB Command Window.



In this model,

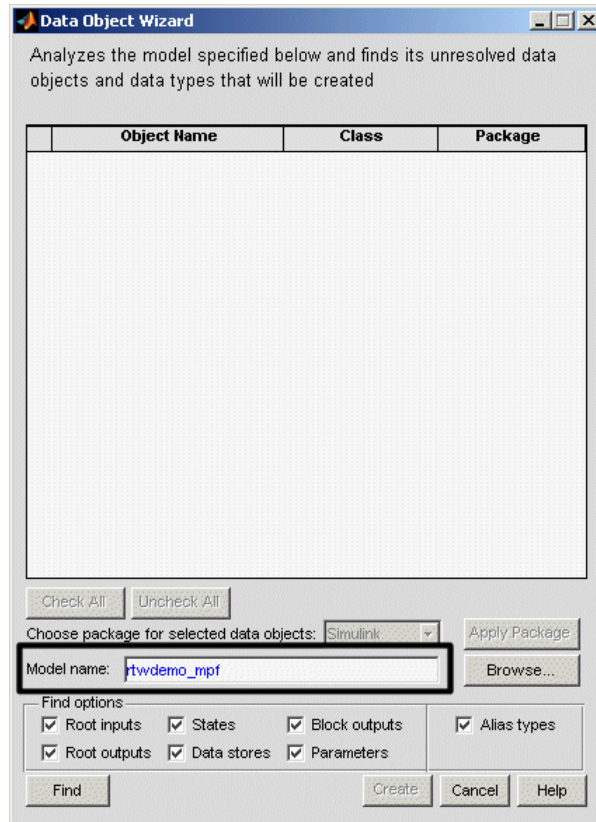
- `A`, `B`, and `C` are input signals, and `L` and `Final` are output signals.
- Subsystem1 receives inputs `A` and `E` and contains constants `G1` and `G2`. Signal `E` is an output from Data Store Read1.
- Subsystem2 receives inputs `C` and `D`. Signal `D` is an output from Data Store Read2. There is a constant in Subsystem2 named `G3`. Also, this subsystem has a Unit Delay block whose state name is `SS`.

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



3 Change to a work folder that is not on an installation path and save the model in that work folder. The code generation software does not allow you to generate code from an installation folder.

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

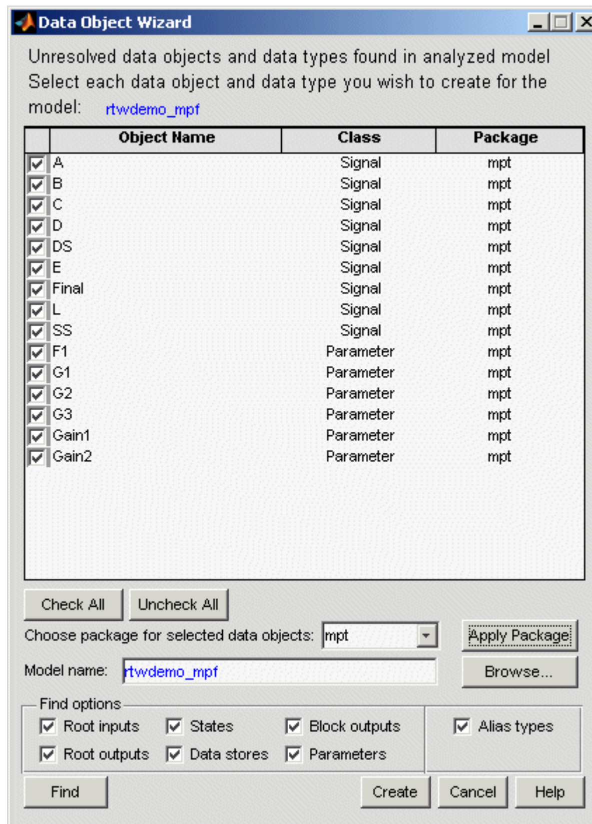


5 Click **Find** on Data Object Wizard. After a moment, the model's parameters and signals appear in Data Object Wizard. These "data objects" make up the data dictionary.

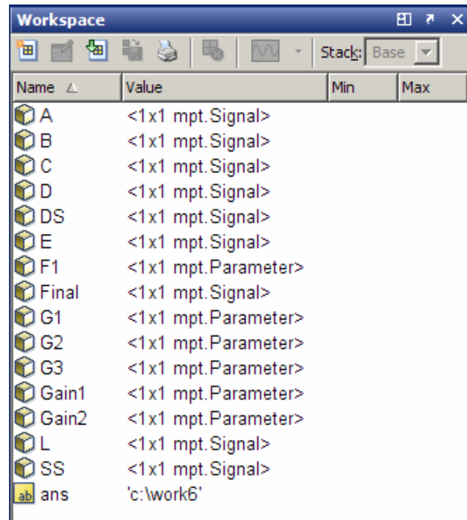
6 Click **Check All**, to select all data objects for the data dictionary.

7 In the **Choose package for selected objects** field, select `mpt`. For an explanation of "package," see "Overview of the Data Dictionary" on page 12-2.

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



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



10 Close Data Object Wizard.

Inspect the Data Dictionary

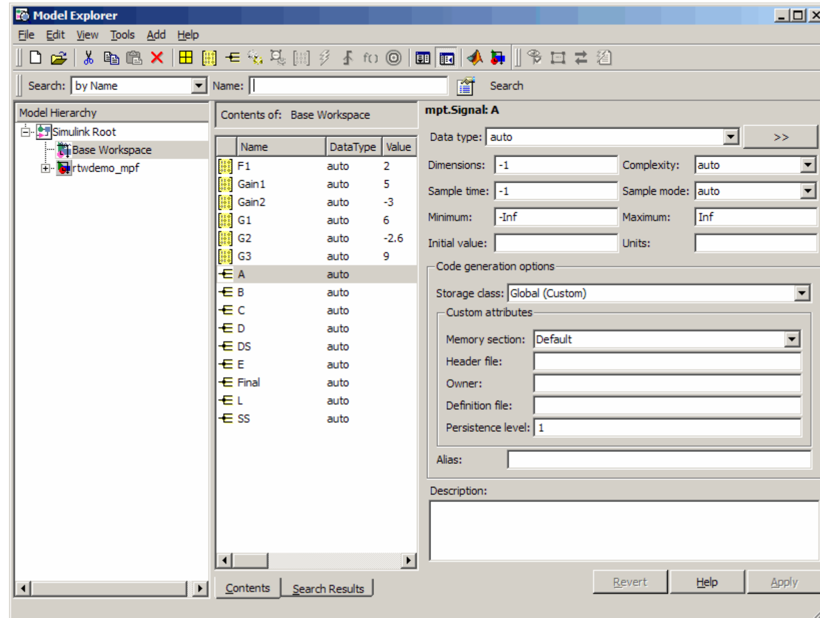
You can verify that each data object you selected in Data Object Wizard is in the data dictionary, using the Model Explorer:

1 If you have not already done so, complete the steps in “Using Data Object Wizard” on page 12-19 .

2 Open the Model Explorer.

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

4 In the middle pane, select data objects one at a time, and notice their property values in the right pane. The figure below shows this for signal `A`. All of the data objects have default property values. Note that for an `mpt` data object, the default in the **Storage class** field is `Global (Custom)`. For descriptions of the properties on the Model Explorer, see “Setting Property Values for Simulink Data Objects” on page 12-8.



Generate and Inspect Code

1 If you have not already done so, complete the steps in “Using Data Object Wizard” on page 12-19 and “Inspect the Data Dictionary” on page 12-23.

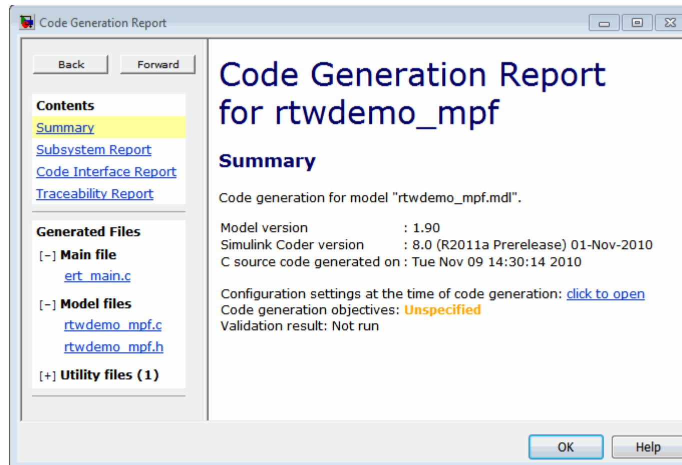
2 In the left pane of the Model Explorer, expand the **rtwdemo_mpf** node.

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

4 In the center pane, click **Code Generation**. The active configuration parameters appear in the right pane.

5 In the **Report** tab, select **Create code generation report**

6 Select the **General** tab. Select **Generate code only**, and then click **Generate code**. After a few moments, the names of the generated files are listed on the Code Generation Report on the left pane.



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

```
real_T F1 = 2.0;
real_T G1 = 6.0;
real_T G2 = -2.6;
real_T G3 = 9.0;
real_T Gain1 = 5.0;
real_T Gain2 = -3.0;
```

Defining All Global Data Objects in a Separate File

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

1. Configure the model's generated code to include all Simulink data objects (signal and parameter) in a separate definition file. Set **Diagnostics > Data Validity > Signal resolution** to `Explicit and implicit`.

2. Specify that data be defined in a separate file. Set **Code Generation > Data Placement > Data definition** to `Data defined in single separate source file`. Accept the default for **Data definition filename**, `global.c`

3. Specify that data be declared in a separate file. Set **Data declaration** to `Data declared in a single separate header file` and accept the default for **Data declaration filename**, `global.h`. Then, click **Apply**.

4. Click **Generate code**. Notice that the code generation report lists `global.c` and `global.h` files.

5. Inspect the code generation report. Notice that
   - The data objects formerly initialized in `rtwdemo_mpf.c` now are initialized in `global.c`.

- The file `rtwdemo_mpf.c` includes `rtwdemo_mpf.h`.
- The file `rtwdemo_mpf.h` includes `global.h`.

Defining a Specific Global Data Object in Its Own File

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

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

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

3. On the **Code Generation > General** pane, click **Generate code**. The code generation report still lists `global.c` and `global.h`, but adds `finalsig.c` and `finalsig.h`.

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

Saving and Loading Data Objects

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

Applying Naming Rules to Identifiers Globally

In this section...

"Overview" on page 12-30

"Changing Names of Identifiers" on page 12-31

"Specifying Simulink Data Object Naming Rules" on page 12-34

"Defining Rules That Change All Signal Names" on page 12-35

"Defining Rules That Change All Parameter Names" on page 12-35

"Defining Rules That Change All #defines" on page 12-36

Note The capabilities described in this section apply both to Simulink and `mpt` data objects.

Overview

Signal and parameter names appear on a Simulink model. The same names appear as data objects on the Model Explorer. By default, these names are replicated exactly in the generated code. For example, `"Speed"` on the model (and workspace) appears as the identifier `"Speed"` in the code, by default. But you can change how they appear in the code. For example, if desired, you can change `"Speed"` to `SPEED` or `speed`. Or, you can choose to use a different name altogether in the generated code, like `MPH`. The only restriction is that you follow ANSI C[2]/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 the procedure in this section. This procedure makes selections on the Configuration Parameters dialog box to change *all* of the names when code generation occurs, according to the same rule. Or, you can change the names *individually* by following the steps described in "Setting Property Values for Simulink Data Objects" on page 12-8. The relevant field in that procedure is **Alias** on the Model Explorer.

2. ANSI® is a registered trademark of the American National Standards Institute, Inc.

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

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

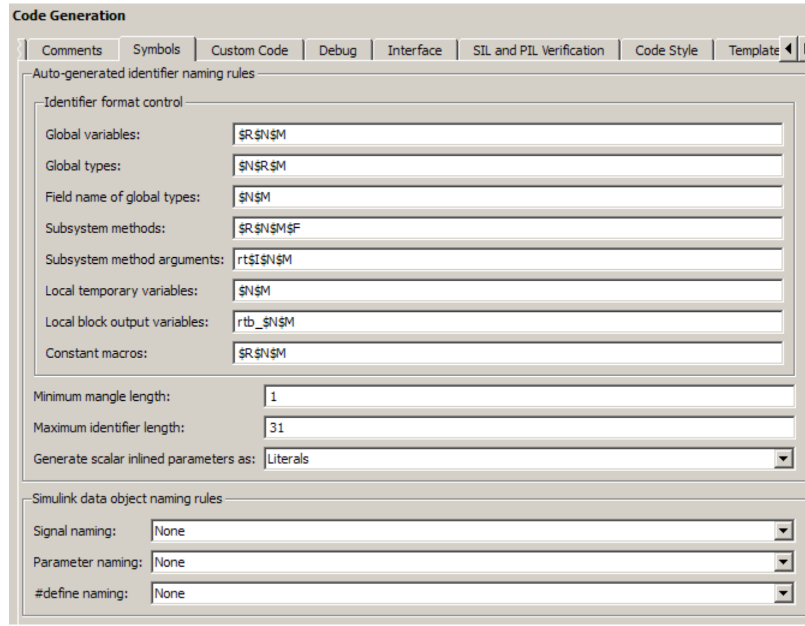
Name of Data Object in Model	Name in Alias Field	Package	Result in Generated Code
A		`Simulink` or `mpt`	a
A	D	`Simulink` or `mpt`	D

Changing Names of Identifiers

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

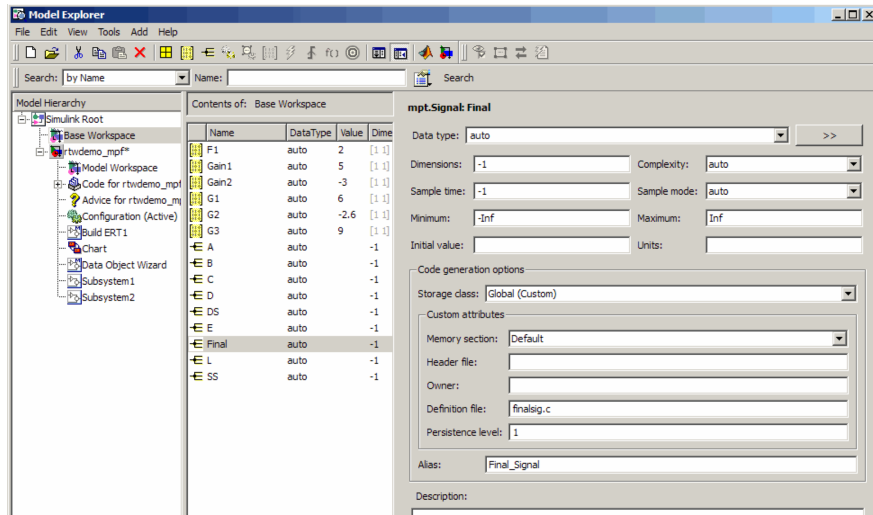
1 Open the **Code Generation > Symbols** pane of the Configuration Parameters dialog.

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



3 Display the base workspace and select `Final`.

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



5 On the **Code Generation > General** pane, click **Generate code** . Now, the signal identifiers in `global.c` and `global.h` appear with lowercase letters.

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

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

```
real T Final_Signal;
```

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

```
extern real_T Final_Signal;
```

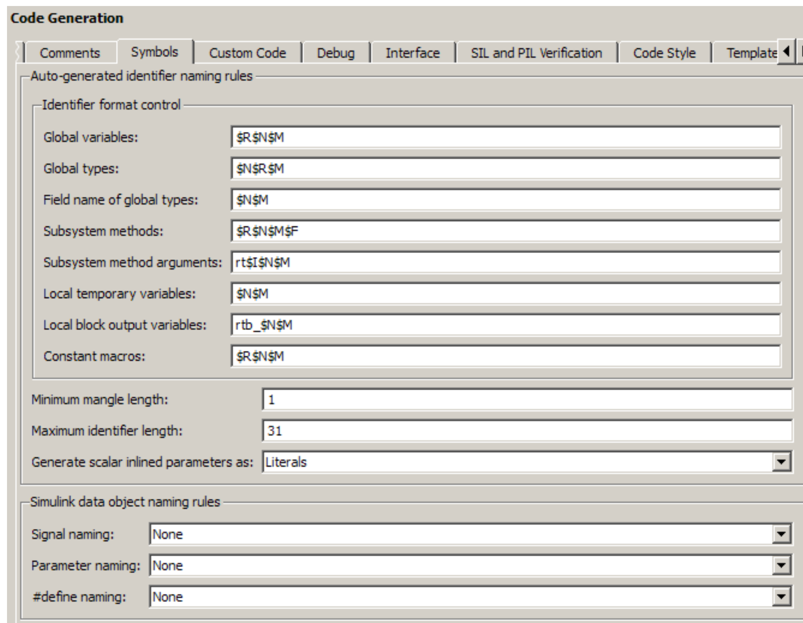
Specifying Simulink Data Object Naming Rules

You specify Simulink data object naming rules on the **Code Generation > Symbols** pane of the Configuration Parameters dialog box. To access that pane,

1 Open an ERT-based model.

2 Open the Configuration Parameters dialog box from the **Simulation** menu or Model Explorer.

3 Open the **Code Generation > Symbols** pane. See the subpane **Simulink data object naming rules**.



Notice the preconfigured settings on this pane. If all of these are acceptable as is, proceed to "Creating User Data Types" on page 12-38. Otherwise, follow the procedures below, as desired, to change signal names, parameter names, or parameter names that you want to use in a `#define` preprocessor directive.

"Code Generation Pane: Symbols" in the Simulink Coder documentation describes all fields on this pane and their possible settings for these procedures.

- "Defining Rules That Change All Signal Names" on page 12-35
- "Defining Rules That Change All Parameter Names" on page 12-35
- "Defining Rules That Change All #defines" on page 12-36

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. In the **Signal naming** menu, click the desired selection. ("Signal naming" explains the possible selections.) The default is `None`. If you selected `Custom M-function`, go to the next step. Otherwise, click **Apply** and then generate and inspect code.
2. Write a MATLAB function 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 12-35.)
3. Save the function as a `.m` file in any folder that is in the MATLAB path.
4. In the **M-function** field under **Signal naming**, type the name of the file you saved in the previous step.
5. Click **Apply** and then generate and inspect code.

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. ("Parameter naming" explains the possible selections.) 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 12-35.

2 Write a MATLAB function 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 #defines

This procedure allows you to change all of the model's parameter names whose storage class you selected as `Define` in "Creating mpt Data Objects with Data Object Wizard" on page 12-11, using the same rule. The new names will appear as identifiers in the generated code:

1 In **#define naming**, click the desired selection. ("#define naming" explains the possible selections.) 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 12-35.

2. Write a MATLAB function that changes all occurrences of the parameter name whose storage class you specified as `Define` in “Creating mpt Data Objects with Data Object Wizard” on page 12-11 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.

Creating User Data Types

In this section...

"Overview" on page 12-38

"Registering User Data Types Using sl_customization.m" on page 12-39

"Example User Data Type Customization Using sl_customization.m" on page 12-41

Note The capabilities described in this section apply both to Simulink and `mpt` data objects.

Overview

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

Code generation software automatically associates the MathWorks C/C++ data types with the equivalent ANSI[3] C/C++ data types. If you want to use only the default MathWorks C/C++ data types, you do not need to register and create your own data types.

To register user data types, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

3. ANSI® is a registered trademark of the American National Standards Institute, Inc.

Once you have registered your user data types using sl_customization.m, you must create the Simulink.AliasType objects corresponding to your user data types. If your model references a user data type either directly (for example, in the output data type of a block) or indirectly (for example, a Simulink.Signal object data type is set to the user data type), you must create the corresponding Simulink.AliasType object before updating the model, running a simulation, or generating code. To create the Simulink.AliasType objects, you can:

- Invoke the MATLAB function ec_create_type_obj to programmatically create all the required Simulink.AliasType objects
- Create Simulink.AliasType objects one at a time by selecting **Add > Simulink.AliasType** in the Model Explorer
- Create Simulink.AliasType objects one at a time by entering the MATLAB command *userdatatype* = Simulink.AliasType, where *userdatatype* is a user data type registered in sl_customization.m

Registering User Data Types Using sl_customization.m

To register user data type customizations, you create an instance of sl_customization.m and include it on the MATLAB path of the Simulink installation that you want to customize. The sl_customization function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the sl_customization function must first get the default (factory) customizations, using the following assignment statement:

```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering Simulink user data type customizations:

- addUserDataType(hObj, userName, builtinName, userHeader)

  addUserDataType(hObj, userName, builtinName)

`addUserDataType(hObj, userName, aliasTypeObj)`

`addUserDataType(hObj, userName, numericTypeObj)`

`addUserDataType(hObj, userName, fixdtString)`

Registers the specified user-defined data type and adds it to the top of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer.

- `userName` — Name of the user data type
- `builtinName` — MathWorks C/C++ data type to which `userName` is mapped
- `userHeader` — Name of the user header file that includes the definition of the user data type
- `aliasTypeObj`, `numericTypeObj`, or `fixdtString` — `Simulink.AliasType`, `Simulink.NumericType`, or `fixdt` to which `userName` is mapped

---

Note If a `Simulink.AliasType` or `Simulink.NumericType` object of the same name as your registered user data type is already defined in the base workspace, the definitions of the existing object and the registered user data type must be consistent or a consistency warning will be displayed.

- `moveUserDataTypeToTop(hObj, userName)`

  Moves the specified user-defined data type to the top of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer.

- `moveUserDataTypeToEnd(hObj, userName)`

  Moves the specified user-defined data type to the end of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer.

- `removeUserDataType(hObj, userName)`

  Removes the specified user-defined data type from the data type list.

Your instance of the `sl_customization` function should use these methods to register user data types for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart your Simulink session or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Example User Data Type Customization Using sl_customization.m

The `sl_customization.m` file shown in Example 1: sl_customization.m for User Data Type Customizations on page 12-41 uses the following methods:

- `addUserDataType` to register the user-defined data types `MyInt16`, `MyInt32`, `MyBool`, and `fixdt(1,8)`
- `moveUserDataTypeToTop` to move `MyBool` to the top of the data type list, as displayed in the **Data type** pull-down list in the Model Explorer
- `removeUserDataType` to remove the built-in data types `boolean` and `double` from the data type list

Example 1: sl_customization.m for User Data Type Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.slDataObjectCustomizer;

% Add custom types
hObj.addUserDataType('MyInt16', 'int16_T', '<mytypes.h>');
hObj.addUserDataType('MyInt32', 'int32_T', '<mytypes.h>');
hObj.addUserDataType('MyBool','boolean_T');
hObj.addUserDataType('fixdt(1,8)');

% Make MyBool first in the list
hObj.moveUserDataTypeToTop('MyBool');

% Remove built-in boolean and double from the list
hObj.removeUserDataType('boolean');
hObj.removeUserDataType('double');
```

```
end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Model Explorer. For example, you could view the customizations as follows:

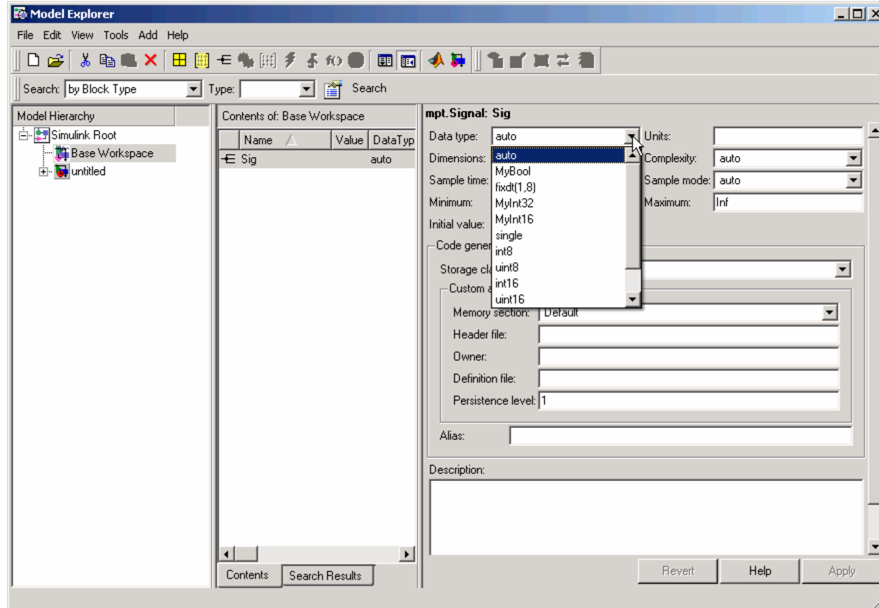
1 Start a MATLAB session.

2 Open Model Explorer, for example, by entering the MATLAB command `daexplr`.

3 Select **Base Workspace**.

4 Add an `mpt` signal, for example, by selecting **Add > MPT Signal**.

5 In the right-hand pane display for the added `mpt` signal, examine the **Data type** drop-down list, noting the impact of the changes specified in Example 1: sl_customization.m for User Data Type Customizations on page 12-41.



Selecting User Data Types for Signals and Parameters

In this section...

"Preparing User Data Types" on page 12-43
"Selecting the User Data Types" on page 12-45

Preparing User Data Types

You can use user-defined data types for Simulink signals and parameters and their corresponding identifiers in generated code. This is true whether or not a signal or parameter has a Simulink data object associated with it.

Before you can select a user data type for a signal or parameter, you must:

1. Create a user data type (alias), as explained in the description of `Simulink.AliasType` in the Simulink documentation. For the example in "Selecting the User Data Types" on page 12-45 demonstrating how to select user data types for signals and parameters, create the alias data type `f32`.

2. Register the user data type so that it is associated with the corresponding MathWorks C/C++ data type, as explained in "Creating User Data Types" on page 12-38. For the example, register the data type `f32` so that it is associated with type `real32_T`. The call to function `addUserDataType` in the `sl_customization.m` file you use for the registration must specify:

   - `f32` as the user data type
   - `real32_T` as the built-in data type
   - `<userdata_types.h>` as the user header file that is to include the user data type definition

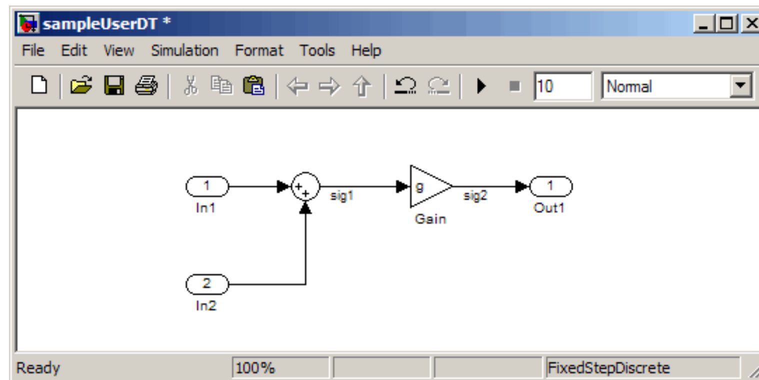
   For example,

   ```
   function sl_customization(cm)

   hObj = cm.slDataObjectCustomizer;

   addUserDataType(hObj, 'f32', 'real32_T', '<userdata_types.h>');
   ```

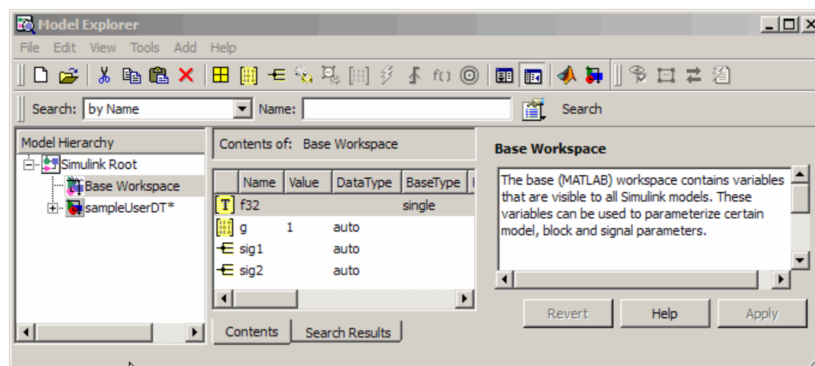
```
end
```

3 If you have not already done so, add the directory containing the `sl_customization.m` file that you created or modified in step 1 to the MATLAB search path.

4 Open a model. The example uses the following model.



5 Create a data dictionary for the model, as explained in "Creating Simulink and mpt Data Objects" on page 12-4, to associate signals and parameters with data objects. For the example, the Model Explorer display must appear as shown below. The three data objects that appear, `sig1` , `sig2`, and `g`, and the registered user data type, `f32`, appear in the middle pane. The "**T**" indicates that `f32` is an alias data type.



For the selection procedure and to continue with the example, continue to "Selecting the User Data Types" on page 12-45.

Selecting the User Data Types

After completing the preparation explained in "Preparing User Data Types" on page 12-43, you can use the user-defined data types for Simulink signals and parameters and for their corresponding identifiers in the generated code. You can use user-defined data types with signals and parameters whether or not they have Simulink objects associated with them.

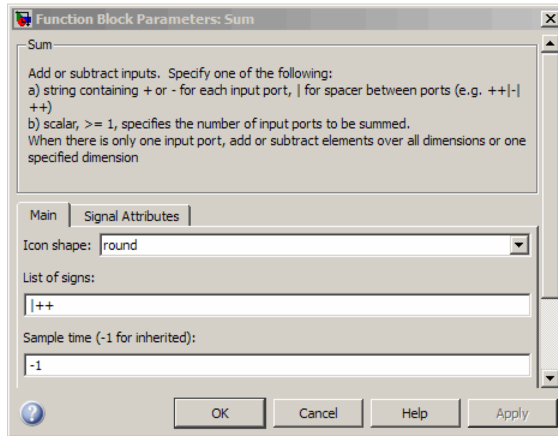
1. For an `mpt` object that is associated with a signal or parameter in your model, in the **Data type** field, select the user data type that you want. For example, select `f32`, for `sig1`.

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

   To continue with the example, type `f32` into the **Data type** field for `sig1`.

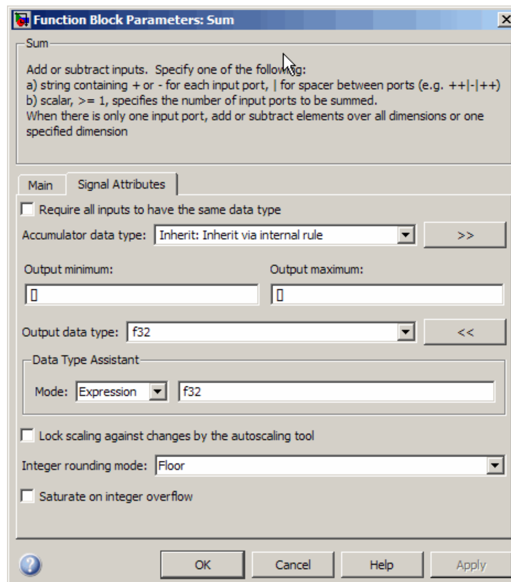
2. Select the model and double-click the signal or parameter source block. (The source block of a model signal or parameter controls the signal's or parameter's data type.) For example, in the example model the Sum block is the source block for `sig1`. Double-click the Sum block.

   The Function Block Parameters dialog box opens.

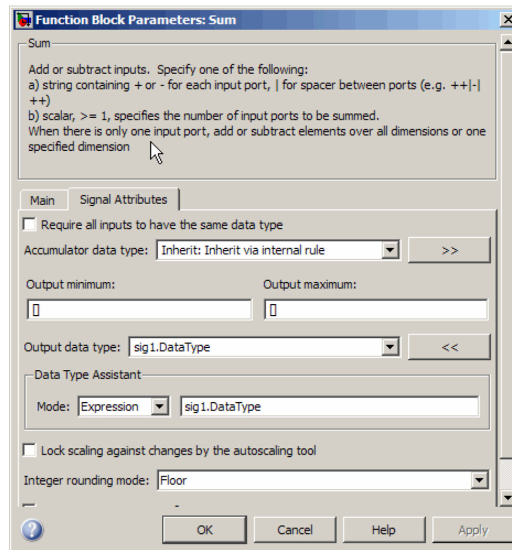


3 Select the **Signal Attributes** tab.

4 In the **Output data type** or **Parameter data type** field, type the name of the user data type that you specified for the data object in step 1, and click **Apply**. The user data type of the signal in the model and that of the signal object are now the same.



Alternatively, you can use dictionary-driven data typing. In the **Output data type** field, specify *`object`*`.DataType`, where *`object`* is the case-sensitive object name. For example, you can specify `sig1.DataType` instead of `f32`.



The advantage of using the alternative is that subsequent user data type changes for the object in the dictionary automatically change the user data type of the corresponding model signal or parameter.

5 Repeat steps 1 through 4 for each remaining model signal and parameter that has a corresponding signal object for which you selected a user data type.

6 Save the model and save all of the data objects in the MATLAB base workspace in a `.mat` file for reuse later. Generated code for `sig1` in the example model (with default MPF settings) would appear as follows:

In `sampleUserDT.c`	`f32 sig1;`
In `sampleUserDT_types.h`	`#include <userdata_types.h>`

Registering mpt User Object Types

In this section...

"Introduction" on page 12-48

"Registering mpt User Object Types Using sl_customization.m" on page 12-48

"Example mpt User Object Type Customization Using sl_customization.m" on page 12-50

Introduction

Embedded Coder software allows you to create custom `mpt` object types and specify properties and property values to be associated with them (see "Creating mpt Data Objects with Data Object Wizard" on page 12-11). Once created, a user object type can be applied to data objects displayed in Model Explorer. When you apply a user object type to a data object, by selecting a type name in the **User object type** pull-down list in Model Explorer, the data object is automatically populated with the properties and property values that you specified for the user object type.

To register `mpt` user object type customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

Registering mpt User Object Types Using sl_customization.m

To register `mpt` user object type customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the `sl_customization` function must first get the default (factory) customizations, using the following assignment statement:

```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering `mpt` user object type customizations:

- `addMPTObjectType(hObj, objectTypeName, classtype, propName1, propValue1, propName2, propValue2, ...)`

  `addMPTObjectType(hObj, objectTypeName, classtype, {propName1, propName2, ...}, {propValue1, propValue2, ...})`

  Registers the specified user object type, along with specified values for object properties, and adds the object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

  - `objectTypeName` — Name of the user object type
  - `classType` — Class to which the user object type applies: `'Signal'`, `'Parameter'`, or `'Both'`
  - `propName` — Name of a property of an `mpt` or `mpt`-derived data object to be populated with a corresponding `propValue` when the registered user object type is selected
  - `propValue` — Specifies the value for a corresponding `propName`

- `moveMPTObjectTypeToTop(hObj, objectTypeName)`

  Moves the specified user object type to the top of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `moveMPTObjectTypeToEnd(hObj, objectTypeName)`

  Moves the specified user object type to the end of the user object type list, as displayed in the **User object type** pull-down list in the Model Explorer.

- `removeMPTObjectType(hObj, objectTypeName)`

Removes the specified user object type from the user object type list.

Your instance of the `sl_customization` function should use these methods to register `mpt` object type customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart your MATLAB session to effect the changes.

Example mpt User Object Type Customization Using sl_customization.m

The `sl_customization.m` file shown in Example 2: sl_customization.m for mpt Object Type Customizations on page 12-50 uses the `addMPTObjectType` method to register the user signal types `EngineType` and `FuelType` for `mpt` objects.

Example 2: sl_customization.m for mpt Object Type Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.slDataObjectCustomizer;

% Add commonly used signal types
hObj.addMPTObjectType(...
    'EngineType','Signal',...
    'DataType', 'uint8',...
    'Min', 0,...
    'Max', 255,...
    'DocUnits','m/sec');

hObj.addMPTObjectType(...
    'FuelType','Signal',...
    'DataType', 'int16',...
    'Min', -12,...
    'Max', 3000,...
    'DocUnits','mg/hr');
```

```
end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Model Explorer. For example, you could view the customizations as follows:

1. Start a MATLAB session.
2. Open Model Explorer, for example, by entering the MATLAB command `daexplr`.
3. Select **Base Workspace**.
4. Add an `mpt` signal, for example, by selecting **Add > MPT Signal**.
5. In the right-hand pane display for the added `mpt` signal, examine the **User object type** drop-down list, noting the impact of the changes specified in Example 2: sl_customization.m for mpt Object Type Customizations on page 12-50.
6. From the **User object type** drop-down list, select one of the registered user signal types, for example, `FuelType`, and verify that the displayed settings are consistent with the arguments specified to the `addMPTObjectType` method in `sl_customization.m`.

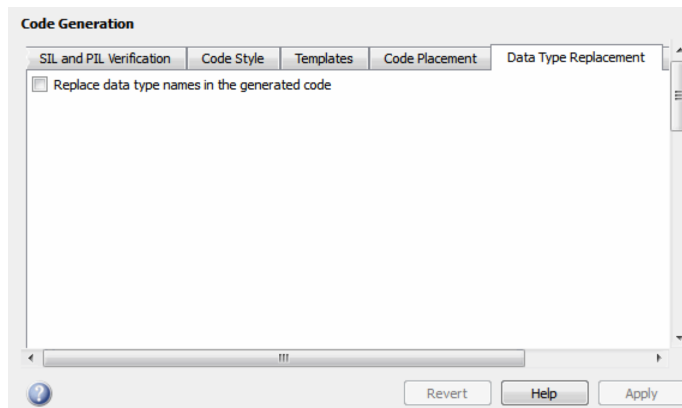
Replacing Built-In Data Type Names in Generated Code

In this section...

"Replacing Built-In Data Type Names" on page 12-52
"Replacing `boolean` with an Integer Data Type" on page 12-57
"Data Type Replacement Limitations" on page 12-59

Replacing Built-In Data Type Names

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



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

- Performs consistency checks to ensure that your specified data type replacements are consistent with your model's data types.
- Allows many-to-one data type replacement, the ability to map multiple built-in data types to one replacement data type in generated code. For

example, built-in data types `uint8` and `boolean` could both be replaced in your generated code by a data type `U8` that you have previously defined.

Note For limitations that apply, see "Data Type Replacement Limitations" on page 12-59.

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

Code Generation

SIL and PIL Verification | Code Style | Templates | Code Placement | Data Type Replacement

☑ Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	
single	real32_T	
int32	int32_T	
int16	int16_T	
int8	int8_T	
uint32	uint32_T	
uint16	uint16_T	
uint8	uint8_T	
boolean	boolean_T	
int	int_T	
uint	uint_T	
char	char_T	

The table **Data type names** lists each Simulink built-in data type name along with its code generation data type name. Selectively fill in fields in the third column with your replacement data types. Each replacement data type should be the name of a `Simulink.AliasType` object that exists in the base workspace. Replacements may be specified or not for each individual built-in type. For each replacement data type entered, the `BaseType` property of the associated `Simulink.AliasType` object must be consistent with the built-in data type it replaces.

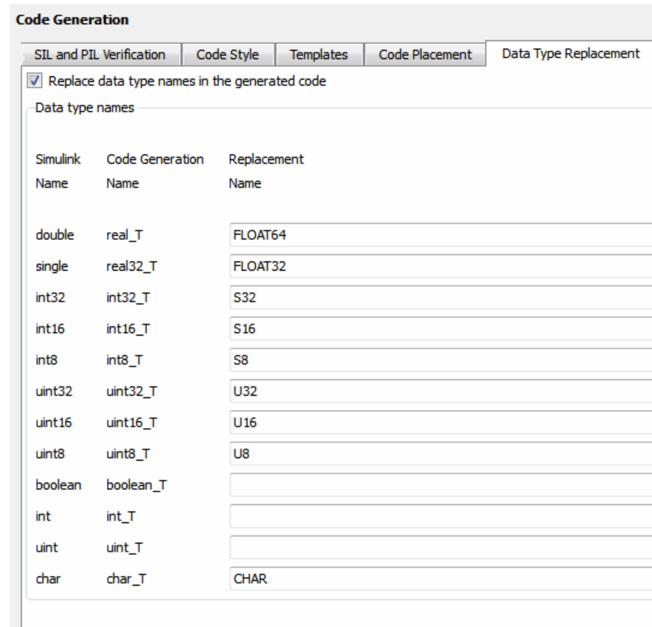
- For `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`, the replacement data type's `BaseType` must match the built-in data type.
- For `boolean`, the replacement data type's `BaseType` must be either an 8-bit integer or an integer of the size displayed for **Number of bits: int** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.
- For `int`, `uint`, and `char`, the replacement data type's size must match the size displayed for **Number of bits: int** or **Number of bits: char** on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

An error occurs if a replacement data type specification is inconsistent.

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

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

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



You can also apply a many-to-one data type replacement mapping. For example, in the following display:

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

Note Many-to-one data type replacement does not support the `char` (`char_T`) built-in data type. Use `char` only in one-to-one data type replacements.

Code Generation

SIL and PIL Verification | Code Style | Templates | Code Placement | Data Type Replacement

☑ Replace data type names in the generated code

Data type names

Simulink Name	Code Generation Name	Replacement Name
double	real_T	
single	real32_T	
int32	int32_T	S32
int16	int16_T	
int8	int8_T	
uint32	uint32_T	U32
uint16	uint16_T	
uint8	uint8_T	U8
boolean	boolean_T	U8
int	int_T	S32
uint	uint_T	U32
char	char_T	

The user-defined replacement types you specify will appear in your model's generated code in place of the corresponding built-in data types. For example, if you specify user-defined data type `FLOAT64` to replace built-in data type `real_T` (`double`), then the original generated code shown in Example 3: Generated Code with real_T Built-In Data Type on page 12-57 will become the modified generated code shown in Example 4: Generated Code with FLOAT64 Replacement Data Type on page 12-57.

Example 3: Generated Code with real_T Built-In Data Type

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

Example 4: Generated Code with FLOAT64 Replacement Data Type

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

Replacing `boolean` with an Integer Data Type

Using data type replacement, you can replace the `boolean` built-in data type in generated code with any integer type, among the following, that might improve the performance of generated code on your production hardware:

- `int8`
- `uint8`
- `int`*`n`*
- `uint`*`n`*

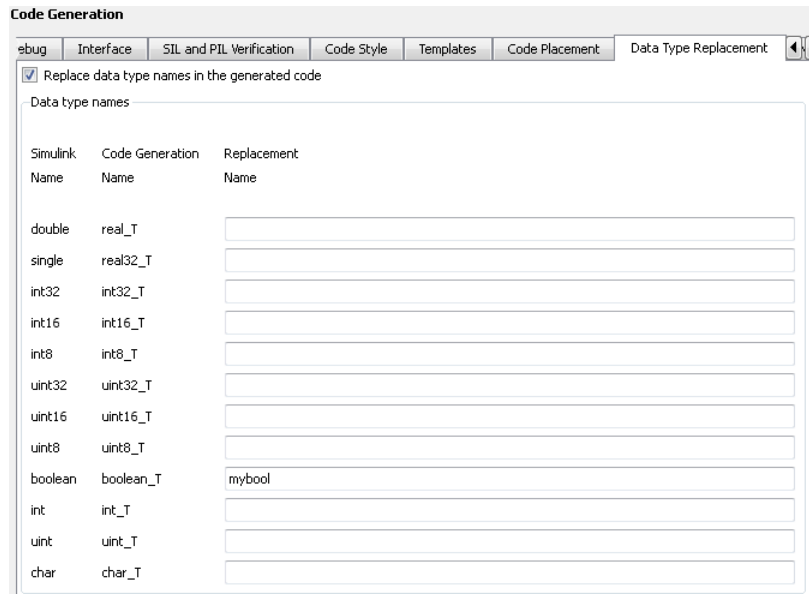
where *`n`* is 8, 16, or 32, matching the integer word size for the production hardware (for example, `int32` for 32-bit hardware).

For example, to map `boolean` to the `int32` data type, perform the following steps.

1 Define a `Simulink.AliasType` object with a base type of `int32`:

```
mybool = Simulink.AliasType;
mybool.BaseType = 'int32';
```

2 Open an ERT-based model. In the **Data Type Replacement** pane of the Configuration Parameters dialog box, map `boolean` (`boolean_T`) to the replacement data type `mybool`.



In the resulting generated code, `boolean` is replaced with `mybool`. For example, `rtwtypes.h` contains:

```
/* Generic type definitions ... */
...
typedef int boolean_T;
 ...
/* Define RTW replacement data types. */
typedef boolean_T mybool;    /* User defined replacement datatype for boolean_T */
```

Boolean data in the generated code is declared with `mybool`. For example, given a model with a Boolean output `Out1`, an `Out1` declaration such as

```
boolean_T Out1;                    /* '<Root>/Out1' */
```

instead is generated in *model*.h as

```
mybool Out1;                    /* '<Root>/Out1' */
```

Data Type Replacement Limitations

- Data type replacement does not support multiple levels of mapping. Each replacement data type name maps directly to one or more built-in data types.
- Data type replacement is not supported for simulation target code generation for referenced models.
- Data type replacement is not supported if the **GRT compatible call interface** option is selected for your model.
- Data type replacement occurs during code generation for all .c, .cpp, and .h files generated in build directories (including top and referenced model build directories) and in the _sharedutils directory. *Exceptions* are as follows:

  rtwtypes.h
  model_sf.c or .cpp (ERT S-function wrapper)
  model_dt.h (C header file supporting external mode)
  model_capi.c or .cpp
  model_capi.h

- Data type replacement is not supported for complex data types.
- Many-to-one data type replacement is not supported for the char built-in data type. Attempting to use char as part of a many-to-one mapping to a user-defined data type introduces a violation of the MISRA C® specification. Specifically, if char (char_T) and either int8 (int8_T) or uint8 (uint8_T) are mapped to the same user replacement type, the result is a MISRA C violation. Additionally, if you try to generate C++ code, invalid implicit type casts are made and compile-time errors may result. Use char only in one-to-one data type replacements.

Customizing Data Object Wizard User Packages

In this section...

"Introduction" on page 12-60

"Registering Data Object Wizard User Packages Using sl_customization.m" on page 12-61

"Example Data Object Wizard User Package Customization Using sl_customization.m" on page 12-62

Introduction

Data Object Wizard (DOW) can be run in connection with a Simulink model to quickly determine which model data are not associated with data objects and to create and associate data objects with the data. (For more information about Data Object Wizard, see "Data Object Wizard" in the Simulink documentation and "Creating Simulink Data Objects with Data Object Wizard" on page 12-5.) If you want the wizard to use data object classes from a package other than the standard Simulink class package to create the data objects, you select the package from the wizard's **Choose package for selected data objects** list. You can customize the package list by adding and removing packages and modifying the list order.

Note User-defined packages that you add to the list must contain a `Simulink.Signal` subclass named `Signal` and a `Simulink.Parameter` subclass named `Parameter`.

To register Data Object Wizard user package customizations, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

Registering Data Object Wizard User Packages Using sl_customization.m

To register Data Object Wizard user package customizations, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. For example,

```
function sl_customization(cm)
```

As a starting point for your customizations, the `sl_customization` function must first get the default (factory) customizations, using the following assignment statement:

```
hObj = cm.slDataObjectCustomizer;
```

You then invoke methods to register your customizations. The customization manager object includes the following methods for registering DOW user package customizations:

- `addUserPackage(hObj, packageName)`

  `addUserPackage(hObj, cellArrayOfStrings)`

  Adds the specified user package(s) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `moveUserPackageToTop(hObj, packageName)`

  Moves the specified user package to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `moveUserPackageToEnd(hObj, packageName)`

  Moves the specified user package to the end of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard.

- `removeUserPackage(hObj, packageName)`

  Removes the specified user package from the package list.

- `setUserPackages(hObj, cellArrayOfStrings)`

Replaces the entire package list with a specified list of user packages.

Your instance of the `sl_customization` function should use these methods to register DOW user package customizations for your Simulink installation.

The Simulink software reads the `sl_customization.m` file when it starts. If you subsequently change the file, you must restart your Simulink session or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Example Data Object Wizard User Package Customization Using sl_customization.m

The `sl_customization.m` file shown in Example 5: sl_customization.m for DOW User Package Customizations on page 12-62 uses the following methods:

- `addUserPackage` to add the user packages `ECoderDemos` and `SimulinkDemos` (present by default in the MATLAB path) to the top of the package list, as displayed in the **Choose package for selected data objects** pull-down list in Data Object Wizard

---

Note Packages`ECoderDemos` and `SimulinkDemos` must contain a `Simulink.Signal` subclass named `Signal` and a `Simulink.Parameter` subclass named `Parameter`.

- `moveUserPackageToEnd` to move `SimulinkDemos` to the end of the package list

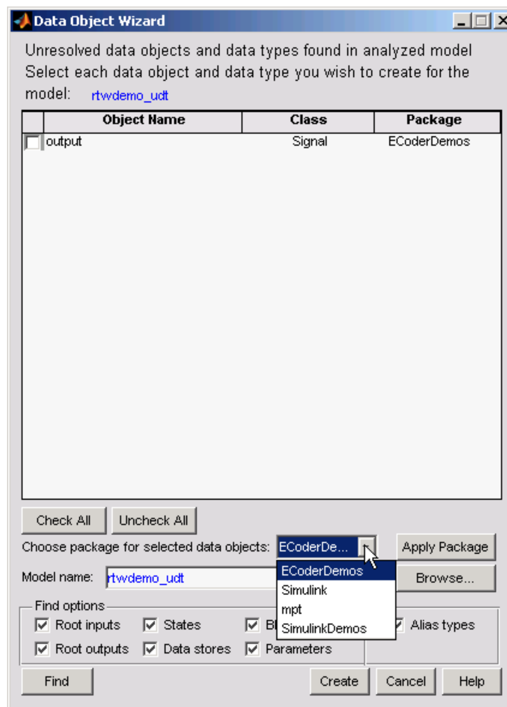
Example 5: sl_customization.m for DOW User Package Customizations

```
function sl_customization(cm)
% Register user customizations

% Get default (factory) customizations
hObj = cm.slDataObjectCustomizer;
```

```
% Add user packages
hObj.addUserPackage({'ECoderDemos', 'SimulinkDemos'});

% Move SimulinkDemos to end of list
hObj.moveUserPackageToEnd('SimulinkDemos');

end
```

If you include the above file on the MATLAB path of the Simulink installation that you want to customize, the specified customizations will appear in Data Object Wizard. For example, you could view the customizations as follows:

1. Start a MATLAB session.
2. Launch a model, such as `rtwdemo_udt`.
3. Open Data Object Wizard, for example, by selecting **Tools > Data Object Wizard** in the Simulink window.
4. In the Data Object Wizard dialog box, click the **Find** button to generate a list of one or more data objects.
5. Examine the **Choose package for selected data objects** drop-down list, noting the impact of the changes specified in Example 5: sl_customization.m for DOW User Package Customizations on page 12-62.



To replace the entire Data Object Wizard package list with a specified list of user packages, you can use a method invocation similar to the following:

```
hObj.setUserPackages({'myPackage1', 'ECoderDemos', 'mpt'});
```

13

Managing Placement of Data Definitions and Declarations

- "Overview of Data Placement" on page 13-2
- "Priority and Usage" on page 13-3
- "Ownership Settings" on page 13-10
- "Memory Section Settings" on page 13-11
- "Data Placement Rules" on page 13-12
- "Example Settings" on page 13-13
- "Data Placement Rules and Effects" on page 13-22

Overview of Data Placement

This chapter focuses on module packaging features (MPF) settings that are interdependent. Their combined values, along with Simulink partitioning, determine the file placement of data definitions and declarations, or *data placement*. This includes

- 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 settings are distributed among the main procedures and form an important interdependency:

- The **Data definition** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Data declaration** field on the **Code Placement** pane of the Configuration Parameters dialog box.
- The **Owner** field of the data object in the Model Explorer, and the **Module naming** and **Module name** fields on the **Code Placement** pane of the Configuration Parameters dialog box. The term "ownership settings" refers to **Owner**, **Module naming**, and **Module name** together.
- The **Definition file** field of the data object on the Model Explorer.
- The **Header file** field of the data object on the Model Explorer.
- The **Memory section** field of the data object on the Model Explorer.

Priority and Usage

In this section...

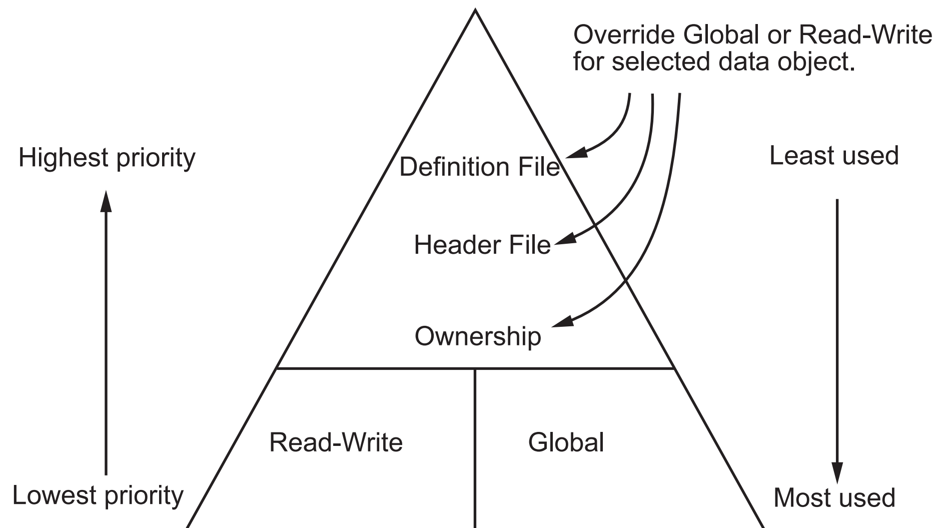
"Overview" on page 13-3
"Read-Write Priority" on page 13-4
"Global Priority" on page 13-7
"Definition File, Header File, and Ownership Priorities" on page 13-8

Overview

There is a priority order among interdependent MPF settings. From highest to lowest, the priorities are

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

Priority order varies inversely with frequency of use, as illustrated below. For example, Definition File is highest priority but least used.



MPF Settings Priority and Usage

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 Data Object Wizard. (See "Creating Simulink Data Objects with Data Object Wizard" on page 12-5 for details.) Before generating the files, you can use the higher priority Definition file, Header file, and Ownership, as desired, to override Read-Write or Global priorities for single data objects. Most users will employ Read-Write or Global, without an override. A few users, however, will want to do an override for certain data objects. We expect that those users whose applications include multiple modules will want to use the Ownership priority.

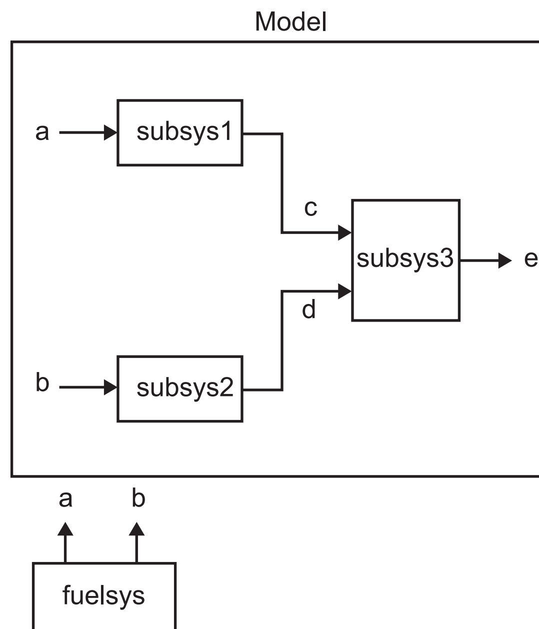
The priorities are in effect only for those data objects that are derived from `Simulink.Signal` and `Simulink.Parameter`, and whose custom storage classes are specified using the Custom Storage Class Designer. (For details, see "Designing Custom Storage Classes and Memory Sections" on page 8-12.) Otherwise, the build process determines the data placement.

Read-Write Priority

This is the lowest priority. Consider that a model consists of one or more Simulink blocks or Stateflow diagrams. There can be subsystems within

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

As explained in Chapter 12, "Managing Data Definitions and Declarations With the Data Dictionary", MPF provides you with the means of selecting a data object that you want defined as an identifier in the generated code. MPF also allows you to specify property values for each data object. For this illustration, we choose to include all of the data objects to be in the dictionary.



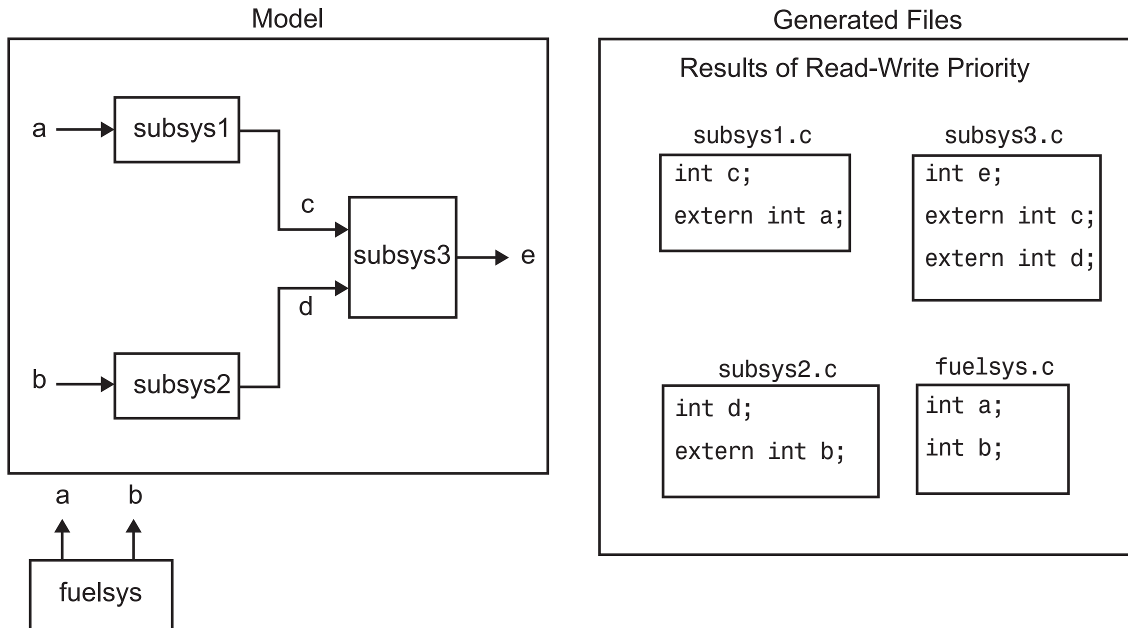
The Generated Files

We generate code for this model. As shown in the figure below, this results in a `.c` source file corresponding to each of the subsystems. (In actual applications, there could be more than one `.c` source file for a subsystem.

This is based on the file partitioning previously selected for the model. But for our illustration, we only need to show one for each subsystem.) Data objects `a` through `e` have corresponding identifiers in the generated files.

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

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



When the Read-Write priority is in effect, this source file contains the definitions for the subsystem's global data objects, if this is the file that first writes to the data object's address. Other files that read (use) that data object

only include a reference to it. This is why this priority is called Read-Write. Since a read and a write of a file are analogous to input and output of a model's block, respectively, there is another way of saying this. The definitions of a block's global data objects are located in the corresponding generated file, if that data object is an output from that block. The declarations (`extern`) of a block's global data objects are located in the corresponding generated file, if that data object is an input to that block.

Settings for Read-Write Priority

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

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

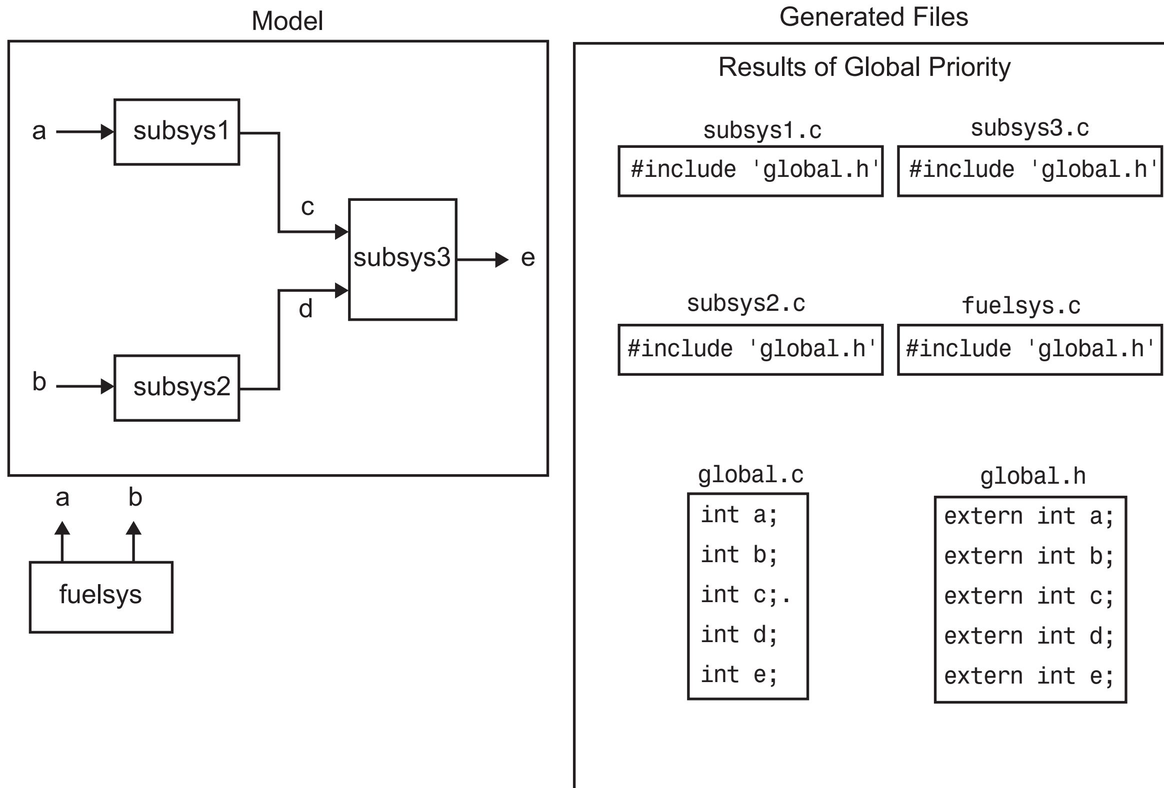
Global Priority

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

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



Definition File, Header File, and Ownership Priorities

While 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, as shown in MPF Settings Priority and Usage on page 13-4

Ownership Settings

Ownership settings refers to the values specified for the **Module naming** and **Module names** fields on the **Code Placement** pane of the Configuration Parameters dialog box, and the **Owner** field of a data object in the Model Explorer. These settings have no effect on what files are generated. Their effects only have to do with definitions and `extern` statements. There are five possible configurations, as shown in "Effects of Ownership Settings" on page 13-22.

Memory Section Settings

Memory sections allow you to specify storage directives for a data object. As shown in Parameter and Signal Property Values on page 7-2, the possible values for the **Memory section** property of a parameter or signal object are `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile`.

If you specify a filename for **Definition file**, and select `Default`, `MemConst`, `MemVolatile` or `MemConstVolatile` for the **Memory section** property, the code generation software generates a `.c` file and an `.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. The `.h` file can be included, using the preprocessor directive `#include`, in any file that needs to reference the data object.

You can add more memory sections. For more information, see "Designing Custom Storage Classes and Memory Sections" on page 8-12 and Chapter 9, "Memory Sections".

Data Placement Rules

For a complete set of data placement rules in convenient tabular form, based on the priorities discussed in this chapter, see "Data Placement Rules and Effects" on page 13-22.

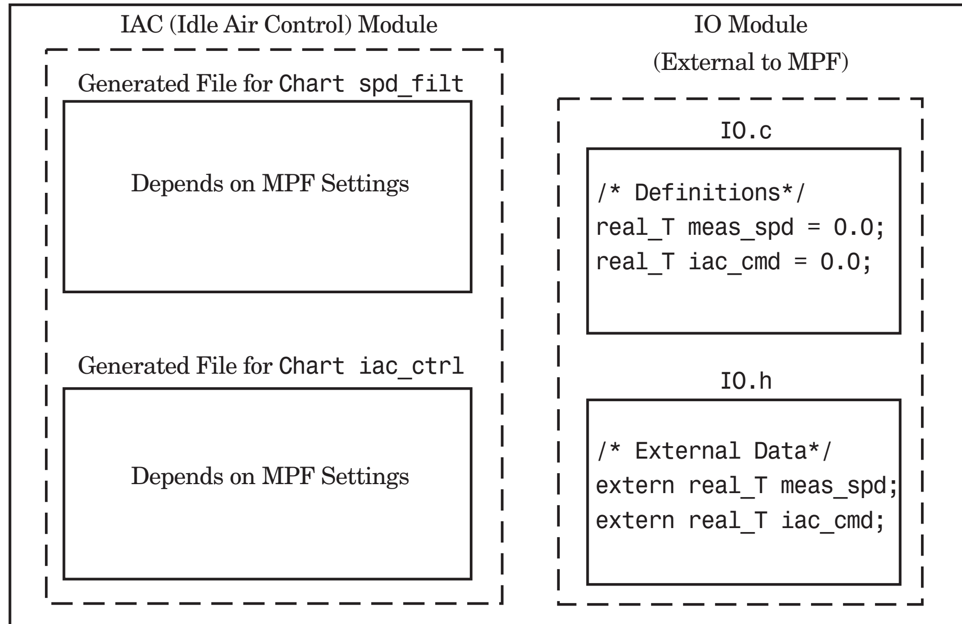
Example Settings

In this section...
"Introduction" on page 13-13
"Read-Write Example" on page 13-15
"Ownership Example" on page 13-17
"Header File Example" on page 13-18
"Definition File Example" on page 13-20

Introduction

"Example Settings and Resulting Generated Files" on page 13-23 provides example settings for one data object of a model. Eight examples are listed so that you can see the generated files that result from a wide variety of settings. Four examples from this table are discussed below in more detail. These discussions provide adequate information for understanding the effects of any settings you might choose. For illustration purposes, the four examples assume that we are dealing with an overall system that controls engine idle speed.

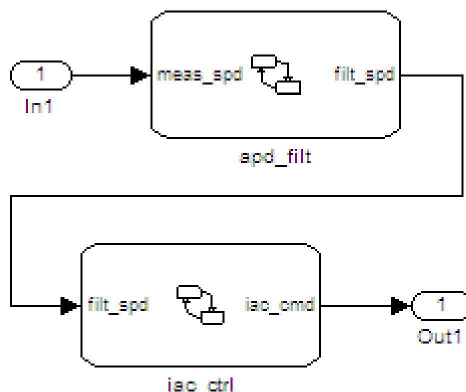
The next figure shows that the software component of this example system consists of two modules, IAC (Idle Air Control), and IO (Input-Output).



Engine Idle Speed Control System

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

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



Proceed to the discussion of the desired example settings:

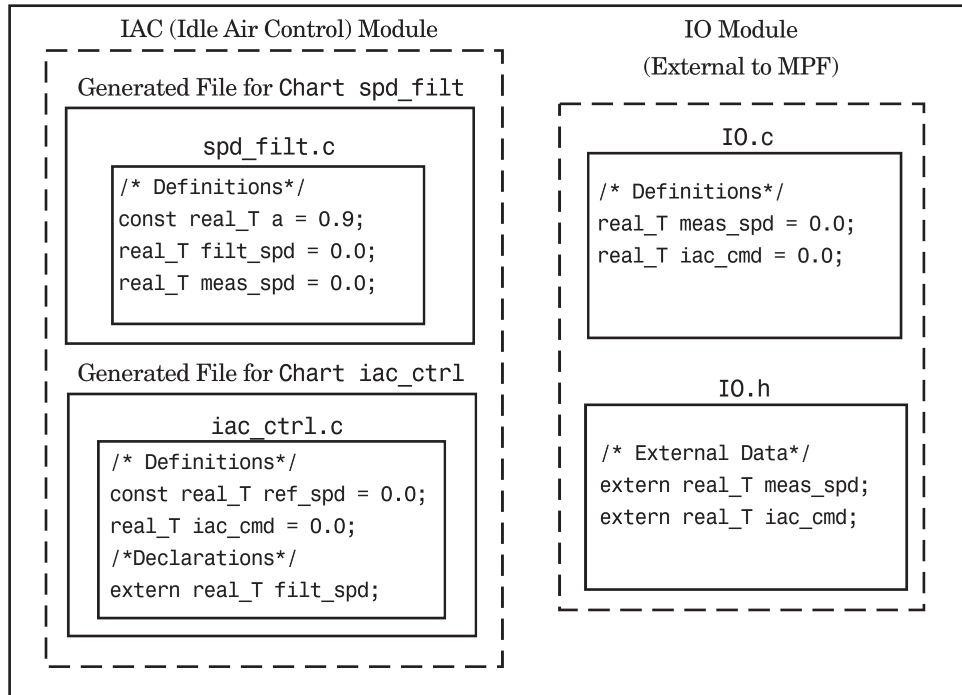
- "Read-Write Example" on page 13-15
- "Ownership Example" on page 13-17
- "Header File Example" on page 13-18
- "Definition File Example" on page 13-20

Read-Write Example

These settings and the generated files that result are shown as Example Settings 1 in "Example Settings and Resulting Generated Files" on page 13-23. 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 **Code Placement** pane of the Configuration Parameters dialog box. Accept the default `Not specified` selection in the **Module naming** field. Accept the default blank settings for the **Owner**, **Definition file** and **Header file** fields on the Model Explorer. For **Memory section**, accept `Default`. Now the Read-Write priority is in

effect. Generate code. The next figure shows the results in terms of definition and declaration statements.



Engine Idle Speed Control System (Read-Write Example)

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

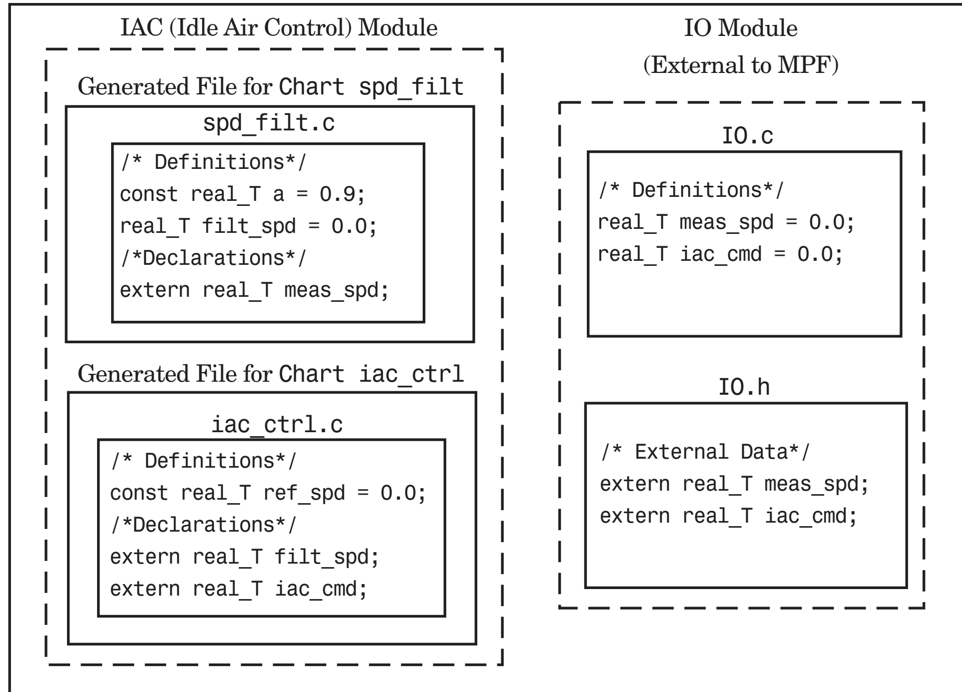
However, notice `real_T filt_spd`. This data object is defined in `spd_filt.c` and declared in `iac_ctrl.c`. That is, since the Read-Write priority is in effect, `filt_spd` is defined in the file that first writes to its address. And, it is declared in the file that reads (uses) it. Further, `real_T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

Ownership Example

See tables "Effects of Ownership Settings" on page 13-22 and "Example Settings and Resulting Generated Files" on page 13-23. In the "Read-Write Example" on page 13-15, there are several instances where the same data object is defined in more than one `.c` source file, and there is no declaration (`extern`) statement. This would result in compiler errors during link time. But in this example, we configure MPF Ownership rules so that adequate linking can take place. Notice the Example Settings 2 row in "Example Settings and Resulting Generated Files" on page 13-23. 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 **Code Placement** pane of the Configuration Parameters dialog box, select `User specified` in the **Module naming** field, and specify IAC in the **Module name** field (case sensitive). Open the Model Explorer (by issuing the MATLAB command `daexplr`) and, for all data objects except `meas_spd` and `iac_cmd`, type IAC in the **Owner** field (case sensitive). Then, only for the `meas_spd` and `iac_cmd` data objects, type IO as their **Owner** (case sensitive). Generate code.

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



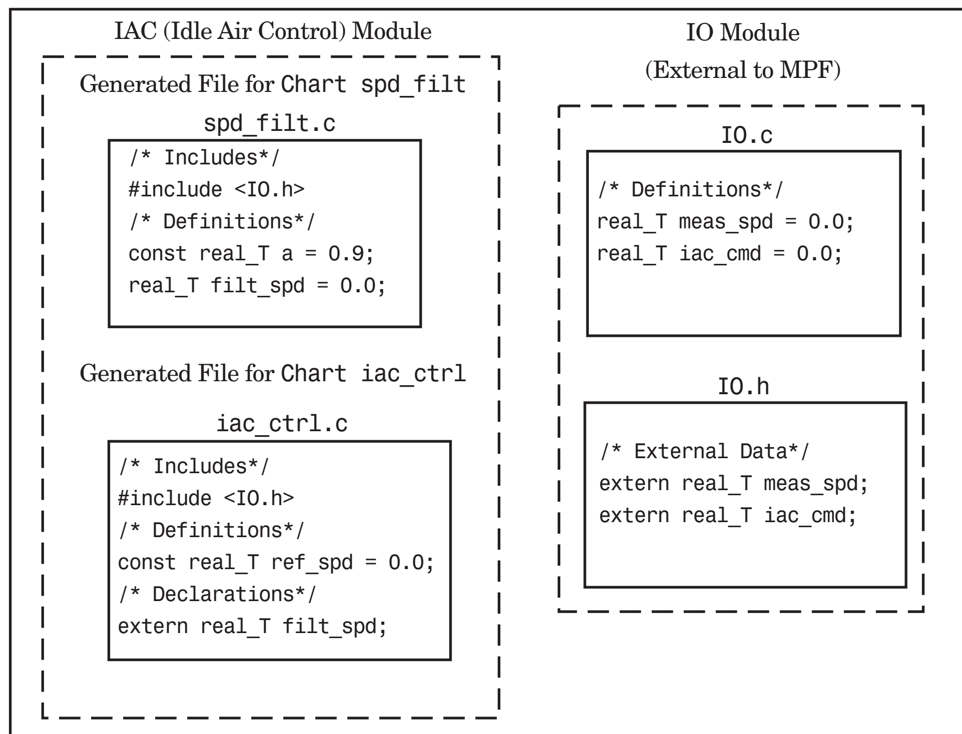
Engine Idle Speed Control System (Ownership Example)

Header File Example

These settings and the generated files that result are shown as Example Settings 3 in "Example Settings and Resulting Generated Files" on page 13-23. Since this example has no **Definition file** specified, it allows us to describe the effects of the **Header file** setting. (If there *were* a **Definition file**, MPF would ignore the **Header file** setting.) The focus of this example is to show how the **Header file** settings result in the linking of the two chart source files to the external IO files, shown in the next figure. (Also, ownership settings will be used to link the two chart files with each other.)

As you can see in the figure, the `meas_spd` and `iac_cmd` identifiers are defined in `IO.c` and declared in `IO.h`. Both of these identifiers are external to the generated `.c` files. You open the Model Explorer and select both the `meas_spd` and `iac_cmd` data objects. For each of these data objects, in the **Header file** field, specify `IO.h`, since this is where these two objects are declared. This setting ensures that the `spd_filt.c` source file will compile and link with the external `IO.c` file without errors.

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



Engine Idle Speed Control System (Header File Example)

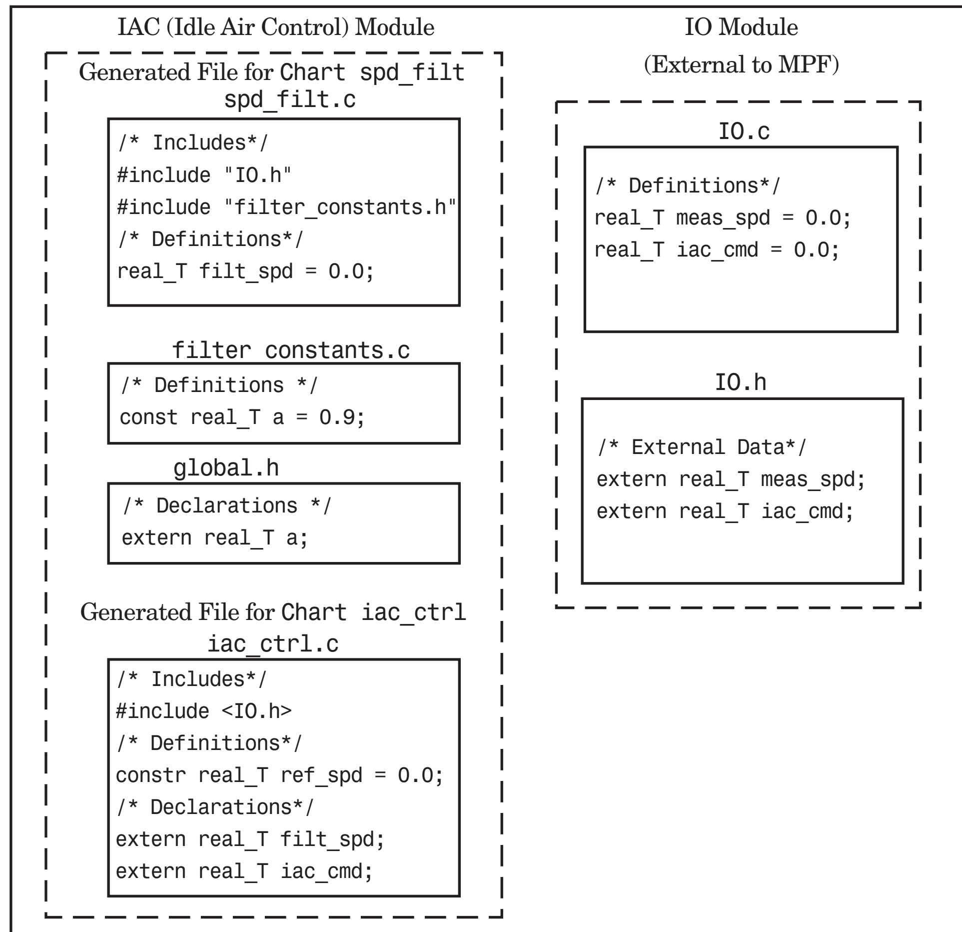
Since you specified the `IO.h` filename for the **Header file** field for the `meas_spd` and `iac_ctrl` objects, the code generator assumed correctly that their declarations are in `IO.h`. So the code generator placed `#include IO.h` in each source file: `spd_filt.c` and `iac_ctrl.c`. So these two files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

Definition File Example

These settings and the generated files that result are shown as Example Settings 4 in “Example Settings and Resulting Generated Files” on page 13-23. Notice that a definition filename is specified. The settings in the table only apply to the data object called `a`. You have decided that you do not want this object defined in `spd_filt.c`, the generated source file for the `spd_filt` chart. (There are many possible organizational reasons one might want an object declared in another file. It is not important for this example to specify the reason.)

For this example, assume the settings for all data objects are the same as those indicated in “Header File Example” on page 13-18, except for the data object `a`. The description below identifies only the differences that result from this.

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



Engine Idle Speed Control System (Definition File Example)

The code generator generates the same files as in the "Header File Example" on page 13-18, and adds a new file, `filter_constants.c`. Data object `a` now is defined in `filter_constants.c`, rather than in the source file `spd_filt.c`, as it is in the example. This data object is declared with an `extern` statement in `global.h`

Data Placement Rules and Effects

In this section...

"Effects of Ownership Settings" on page 13-22
"Example Settings and Resulting Generated Files" on page 13-23
"Data Placement Rules" on page 13-25

Effects of Ownership Settings

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

* See also “Ownership Settings” on page 13-10.
** Default.

Example Settings and Resulting Generated Files

	Data Defined In...	Data Declared In...	Owner- ship*	Defined File**	Header File	Generated Files
Example Settings 1 (Rd-Write Example)	Source file	Source file	Blank	Blank	Blank	`.c/.cpp` source file
Example Settings 2 (Owner- ship Example)	Source file	Source file	Name of module specified	Blank	Blank	`.c/.cpp` source file
Example Settings 3 (Header File Example)	Source file	Source file	Blank	Blank	Desired include filename specified.	`.c/.cpp` source file `.h` definition file
Example Settings 4 (Def. File Example)	Source file	Source file	Blank	Desired definition filename specified.	Desired include filename specified.	`.c/.cpp` source file `.c/.cpp` definition file* `.h` definition file*
Example Settings 5	Single separate source file	Source file	Blank	Blank	Blank	`.c/.cpp` source file global `.c/.cpp`

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

	Data Defined In...	Data Declared In...	Owner-ship*	Defined File**	Header File	Generated Files

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

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

* "Blank" in ownership setting means `Not specified` is selected in the **Module naming** field on the **Code Placement** pane, and the **Owner** field on the Model Explorer is blank. "Name of module specified" can be a variety of ownership settings as defined in "Effects of Ownership Settings" on page 13-22.

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

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

Data Placement Rules

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
mpt or Simulink Noncustom Storage Classes:								
`auto`	N/A	N/A	N/A	N/A	N/A	Note 12	`model.h`	Note 1
`Exported-Global`	N/A	N/A	N/A	N/A	N/A	`model.c`	`model.h`	Note 1
`Imported--Extern`, `Imported--Extern-Pointer`	N/A	N/A	N/A	N/A	N/A	None. External	`model_-private.h`	Note 2
`Simulink-Global`	N/A	N/A	N/A	N/A	N/A	Note 13	`model.h`	Note 1
mpt or Simulink Custom Storage Class: Imported Data:								
`Imported--FromFile`	D/C	D/C	D/C	N/A	null	None	`model_-private.h`	Note 3
`Imported--FromFile`	D/C	D/C	D/C	N/A	`hdr.h`	None	`model_-private.h`	Note 4
Simulink Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	N/A	N/A	`#define`, `model.h`	Note 5
mpt Custom Storage Class: #define Data:								
`Define`	D/C	D/C	N/A	N/A	null	N/A	`#define`, `model.h`	Note 5
`Define`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	`#define`, `model.h`	Note 6
mpt or Simulink Custom Storage Class: GetSet:								
`GetSet`	D/C	D/C	N/A	N/A	`hdr.h`	N/A	External `hdr.h`	Note 4

	Global Settings:		Override Settings for Specific Data Object:			Results in Generated Files:		
Storage Class Setting	**Data Def.**	**Data Dec.**	**Def. File**	**Owner**	**Header File**	**Where Data Def. Is**	**Where Data Dec. Is**	**Dec. Inclusion**
mpt or Simulink Custom Storage Class: Bitfield, Struct:								
`Bitfield`, `Struct`	D/C	D/C	N/A	N/A	N/A	`model.c`	`model.h`	Note 7
mpt Custom Storage Class: Global, Const, ConstVolatile, Volatile:								
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`auto`	null	null or locally owned	null	`model.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`auto`	null	null or locally owned	null	`src.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`auto`	null	null or locally owned	null	`gbl.c`	`model.h`	Note 1
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`src`	null	null or locally owned	null	`model.c`	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`src`	null	null or locally owned	null	`src.c`	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`src`	null	null or locally owned	null	`gbl.c`	`src.c`	Note 8
`Global`, `Const`, `Const-Volatile`, `Volatile`	`auto`	`sep`	null	null or locally owned	null	`model.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	`src`	`sep`	null	null or locally owned	null	`src.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	`sep`	`sep`	null	null or locally owned	null	`gbl.c`	`gbl.h`	Note 9
`Global`, `Const`, `Const-Volatile`, `Volatile`	D/C	D/C	data.c	D/C	null	`data.c`	See Note 10.	Note 10

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

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

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

Notes

In the previous table:

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

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

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

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

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

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

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

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

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

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

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

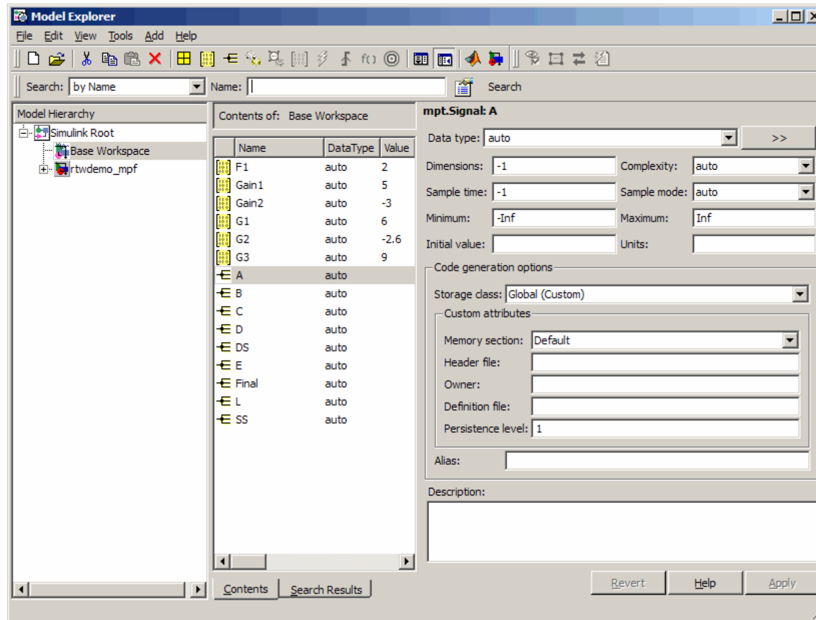
Note 11: `header.h` is included in `model.h`, which is in all source files.

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

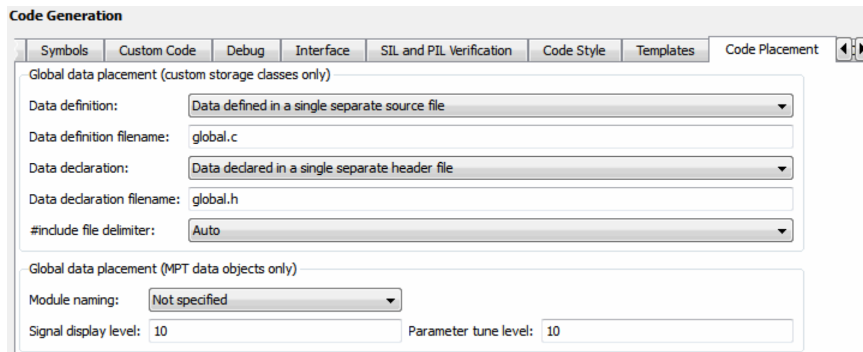
Note 13: Signal: In a structure that is defined in `model.c`. Parameter: In a structure that is defined in `model_data.c`.

Specifying the Persistence Level for Signals and Parameters

With this procedure, you can control the persistence level of signal and parameter objects associated with a model. Persistence level allows you to make intermediate variables or parameters global during initial development. At the later stages of development, you can use this procedure to remove these signals and parameters for efficiency. Notice the **Persistence Level** field on the Model Explorer, as illustrated in the figure below. For descriptions of the properties on the Model Explorer, see Parameter and Signal Property Values on page 7-2.



Notice also the **Signal display level** and **Parameter tune level** fields on the **Code Placement** pane of the Configuration Parameters dialog box, as illustrated in the next figure.



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

the **Persistence level** field. The **Signal display level** number is for all `mpt` (module packaging tool) signal data objects in the model. The **Persistence level** number is for a *particular* `mpt` signal data object. If the data object's **Persistence level** is equal to or less than the **Signal display level**, the signal appears in the generated code as global data with all of the properties (custom attributes) specified in "Creating mpt Data Objects with Data Object Wizard" on page 12-11. For example, this would occur if **Persistence level** is `2` and **Signal display level** is `5`.

Otherwise, the code generator automatically determines how the particular signal data object appears in the generated code. Depending on the settings on the **Optimization** pane of the Configuration Parameters dialog box, the signal data object could appear in the code as local data and thus have none of the custom attributes you specified for that data object. Or, based on expression folding, the code generator could remove the data object so that it does not appear in the code. (See "Tips for Optimizing the Generated Code" on page 21-19 and "Optimizing Generated Code" in the Simulink Coder documentation.)

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

The number you specify in this field is relative to the number you specify in the **Persistence level** field. The **Parameter tune level** number is for all `mpt` parameter data objects in the model. The **Persistence level** number is for a *particular* `mpt` parameter data object. If the data object's **Persistence level** is equal to or less than the **Parameter tune level**, the parameter appears in the generated code with all of the properties (custom attributes) specified in "Creating mpt Data Objects with Data Object Wizard" on page 12-11, and thus is tunable. For example, this would occur if **Persistence level** is `2` and **Parameter tune level** is `5`.

Otherwise, the parameter is inlined in the generated code, and the code generation settings determine its exact form.

Note that, in the initial stages of development, you may be more concerned about debugging than code size. Or, you may want to ensure that one or more particular data objects appear in the code so that you can analyze intermediate calculations of an equation. In this case, you may want to

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

1. With the model open, in the Configuration Parameters dialog box, click **Code Generation > Code Placement**.

2. Type the desired number in the **Signal display level** or **Parameter tune level** field, and click **Apply**.

3. In the Model Explorer, type the desired number in the **Persistence** field for the selected signal or parameter, and click **Apply**.

4. Save the model and generate code.

Preparing Models for Code Generation

- Chapter 15, “Mapping Application Objectives to Model Configuration Parameters”
- Chapter 16, “Selecting and Configuring an Embedded Real-Time Target”
- Chapter 17, “Specifying Code Appearance and Documentation”
- Chapter 18, “Defining Model Configuration Variations”

15

Mapping Application Objectives to Model Configuration Parameters

- "Considerations When Mapping Application Objectives" on page 15-2
- "Defining High-Level Code Generation Objectives" on page 15-3
- "Determining Whether the Model is Configured for Specified Objectives" on page 15-4
- "Creating Custom Objectives" on page 15-11

Considerations When Mapping Application Objectives

The first step in applying Embedded Coder configuration options to the application development process is to consider how your application objectives, particularly with respect to efficiency, traceability, and safety, map to code generation options in a model configuration set.

Parameters that you set in the **Solver**, **Data Import/Export**, **Diagnostics**, and **Code Generation** panes of the Configuration Parameters dialog box affect the behavior of a model in simulation and the code generated for the model.

Consider questions such as the following:

- What settings might help you debug your application?
- What is the highest objective for your application — efficiency, traceability, extra safety precaution, debugging, or some other criteria?
- What is the second highest objective?
- Can the objective at the start of the project differ from the objective required for the end result? What tradeoffs can you make?

After you answer these questions:

1. Define your objectives in the configuration set. For more information, see "Defining High-Level Code Generation Objectives" on page 15-3.

2. Use the Code Generation Advisor to identify parameter values that are not configured for the objectives that you selected. For more information, see "Determining Whether the Model is Configured for Specified Objectives" on page 15-4.

Defining High-Level Code Generation Objectives

When you are considering the objectives for your application, there are many different criteria. The code generation software identifies six high-level objectives that you might consider for your application:

- Execution efficiency — Configure code generation settings to achieve fast execution time.
- ROM efficiency — Configure code generation settings to reduce ROM usage.
- RAM efficiency — Configure code generation settings to reduce RAM usage.
- Traceability — Configure code generation settings to provide mapping between model elements and code.
- Safety precaution — Configure code generation settings to increase clarity, determinism, robustness, and verifiability of the code.
- Debugging — Configure code generation settings to debug the code generation build process.
- MISRA-C:2004 guidelines — Configure code generation settings to increase compliance with MISRA-C:2004 guidelines.

Once you have identified which of these four objectives are important for your application, you can use the Code Generation Advisor to identify the parameters that are not configured for the objectives that you selected. Review "Recommended Settings Summary" to see the settings the Code Generation Advisor recommends.

You can specify and prioritize any combination of the available objectives for the Code Generation Advisor to take into consideration. For more information, see "Determining Whether the Model is Configured for Specified Objectives" on page 15-4.

Determining Whether the Model is Configured for Specified Objectives

In this section...

"Specifying Code Generation Objectives Using the GUI" on page 15-4
"Specifying Code Generation Objectives at the Command Line" on page 15-6
"Reviewing Objectives in Referenced Models" on page 15-7
"Reviewing the Model Without Generating Code" on page 15-7
"Reviewing the Model During Code Generation" on page 15-9

You can use the Code Generation Advisor to review your model and identify the parameters that are not configured for your objective. The Code Generation Advisor reviews a subset of model configuration parameters and displays the results in the **Check model configuration settings against code generation objectives** check.

The Code Generation Advisor uses the information presented in "Mapping of Application Requirements to the Optimization Pane : General tab" to determine the recommended values. When there is a conflict due to multiple objectives, the higher-priority objective takes precedence.

Tip You can use the Code Generation Advisor to review a model before generating code, or as part of the code generation process. When you choose to review a model before generating code, you specify which model, subsystem, or referenced model the Code Generation Advisor reviews (see "Reviewing the Model Without Generating Code" on page 15-7). When you choose to review a model as part of the code generation process, the Code Generation Advisor reviews the entire system (see "Reviewing the Model During Code Generation" on page 15-9).

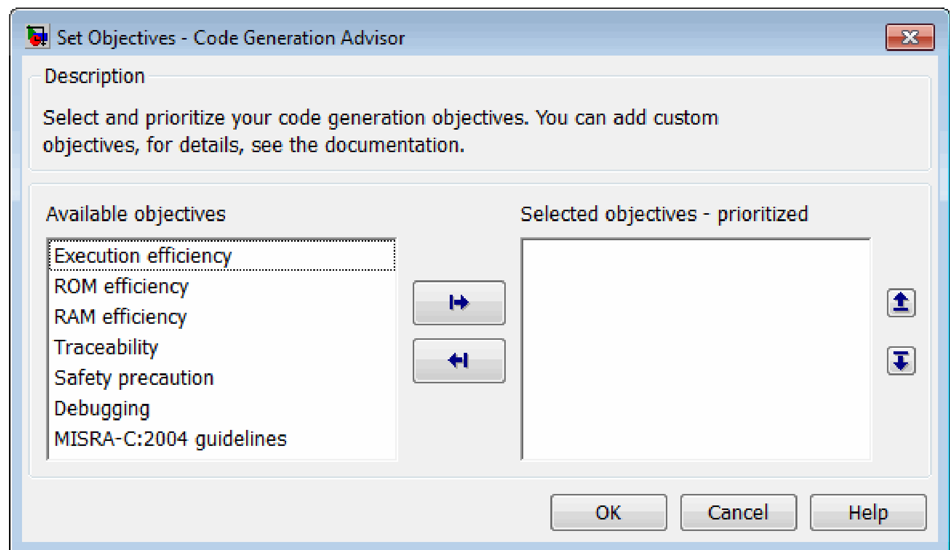
Specifying Code Generation Objectives Using the GUI

To specify code generation objectives in the Configuration Parameters dialog box:

1 Open the Configuration Parameters dialog box and select the **Code Generation > General** pane.

2 Specify a system target file. If you specify an ERT-based target, more objectives are available. For the purposes of this example, choose an ERT-based target such as `ert.tlc`.

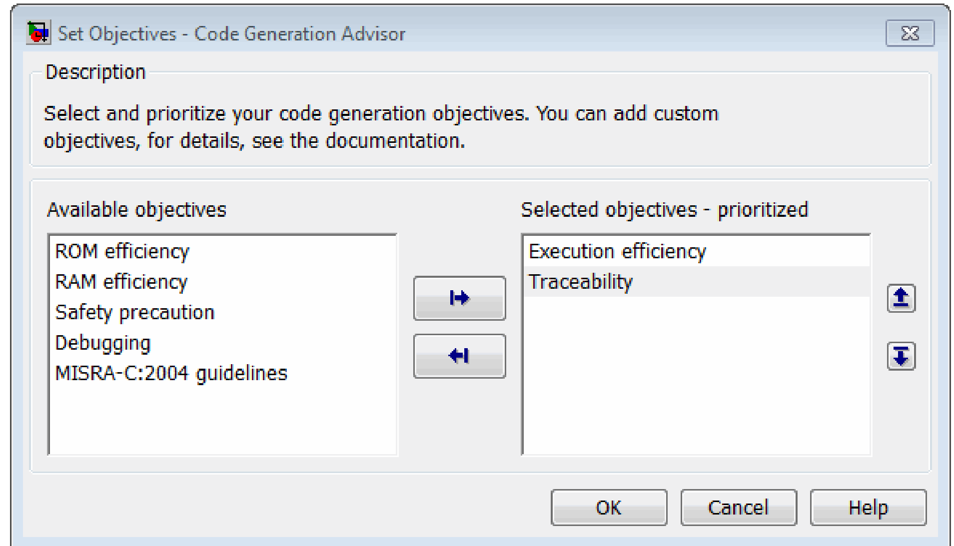
3 Click **Set objectives**. The Set Objectives - Code Generation Advisor dialog box opens.



4 In the Set Objectives - Code Generation Advisor dialog box, specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, do the following:

a In **Available objectives**, double-click `Execution efficiency`. `Execution efficiency` is added to **Selected objectives - prioritized**.

b In **Available objectives**, double-click `Traceability`. `Traceability` is added to **Selected objectives - prioritized** below `Execution efficiency`.



c Click **OK** to accept the objectives. In the Configuration Parameters dialog box, **Code Generation > General > Prioritized objectives** is updated.

Specifying Code Generation Objectives at the Command Line

To specify code generation objectives by writing a MATLAB script or entering commands at the command line:

1 Specify a system target file. If you specify an ERT-based target, more objectives are available. For the purposes of this example, specify `ert.tlc`, where *`model_name`* is the name or handle to the model.

```
set_param(model_name, 'SystemTargetFile', 'ert.tlc');
```

2 Specify your objectives. For example, if your objectives are execution efficiency and traceability, in that priority, enter:

```
set_param(model_name, 'ObjectivePriorities',...
{'Execution efficiency', 'Traceability'});
```

Caution When you specify a GRT-based system target file, you can specify any objective at the command line. If you specify `Execution efficiency`, `ROM efficiency`, `RAM efficiency`, `Traceability`, or `Safety precaution`, the build process changes the objective to `Unspecified` because you have specified a value that is invalid when using a GRT-based target.

Reviewing Objectives in Referenced Models

When you review a model during the code generation process, you must specify the same objectives in the top model and referenced models. If you specify different objectives for the top model and referenced model, the build process generates an error.

To specify different objectives for the top model and each referenced model, review the models separately without generating code.

Reviewing the Model Without Generating Code

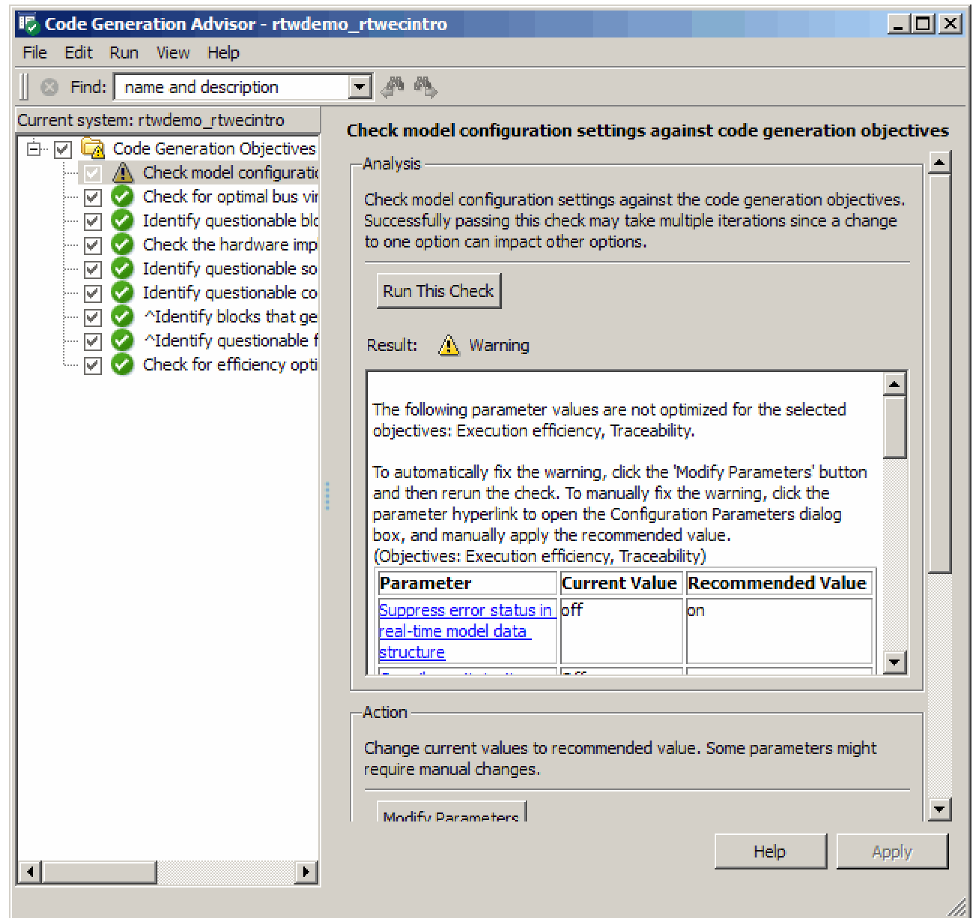
To review a model without generating code using the Code Generation Advisor:

1. Specify your code generation objectives.

2. In the **Configuration Parameters > Code Generation > General** pane, click **Check model**. The System Selector window opens.

3. Select the model or subsystem that you want to review and click **OK**. The Code Generation Advisor opens and reviews the model or subsystem that you specified.

4 In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.



- After reviewing the check results, you can choose to fix warnings and failures, as described in "Fixing a Warning or Failure" in the *Simulink User's Guide*.

Caution When you specify an efficiency or safety precaution objective, the Code Generation Advisor includes additional checks.

When you make changes to one check, the other check results are no longer valid and you must run the checks again for accurate results.

Reviewing the Model During Code Generation

To review a model as part of the code generation process using the Code Generation Advisor:

1. Specify your code generation objectives.

2. In the **Configuration Parameters > Code Generation > General** pane, select one of the following from **Check model before generating code**:

   - `On (proceed with warnings)`
   - `On (stop for warnings)`

3. Select **Generate code only** if you only want to generate code; otherwise clear the check box to build an executable.

4. Apply your changes and then click **Generate code/Build**. The Code Generation Advisor starts and reviews the top model and subsystems.

   If there are no failures or warnings in the Code Generation Advisor, the build process proceeds. If there are failures or warnings and you specified:

   - `On (proceed with warnings)` — The Code Generation Advisor window opens while the build process proceeds. You can review the results after the build process is complete.
   - `On (stop for warnings)` — The build process halts and displays the diagnostics viewer. To continue, you must review and resolve the Code Generation Advisor results or change the **Check model before generating code** selection.

5. In the Code Generation Advisor window, review the results by selecting a check from the left pane. The right pane populates the results for that check.

6. After reviewing the check results, you can choose to fix warnings and failures as described in “Fixing a Warning or Failure” in the *Simulink User’s Guide*.

Caution When you specify an efficiency or safety precaution objective, the Code Generation Advisor includes additional checks. When you make changes to one of these checks, the other check results are no longer valid and you must run the check again for accurate results.

Creating Custom Objectives

In this section...

"Specifying Parameters in Custom Objectives" on page 15-11

"Specifying Checks in Custom Objectives" on page 15-12

"Determining Checks and Parameters in Existing Objectives" on page 15-12

"How to Create Custom Objectives" on page 15-14

The Code Generation Advisor reviews your model based on objectives that you specify. If the predefined efficiency, traceability, safety precaution, and debugging objectives do not meet your requirements, you can create custom objectives.

You can create custom objectives by:

- Creating a new objective and adding parameters and checks to a new objective.
- Creating a new objective based on an existing objective, then adding, modifying and removing the parameters and checks within the new objective.

Specifying Parameters in Custom Objectives

When you create a custom objective, you specify the values of configuration parameters that the Code Generation Advisor reviews using the following methods:

- `addParam` — Add parameters and specify the values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**. When you add parameters that have dependencies, the software includes the dependencies in the list of parameter values that the Code Generation Advisor reviews.
- `modifyInheritedParam` — Modify inherited parameter values that the Code Generation Advisor reviews in **Check model configuration settings against code generation objectives**.

- `removeInheritedParam` — Remove inherited parameters from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this parameter, the Code Generation Advisor reviews the parameter value in **Check model configuration settings against code generation objectives**.

Specifying Checks in Custom Objectives

All objectives include the **Check model configuration settings against code generation objectives** check by default. When you create a custom objective, you specify the list of additional checks that are associated with the custom objective using the following methods:

- `addCheck` — Add checks to the Code Generation Advisor. When a user selects the custom objective, the Code Generation Advisor displays the check, unless the user specifies an additional objective with a higher priority that excludes the check.

  For example, you might add a check to the Code Generation Advisor to include a custom check in the automatic model checking process.

- `excludeCheck` — Exclude checks from the Code Generation Advisor. When a user selects multiple objectives, if the user specifies an additional objective that includes this check as a higher priority objective, the Code Generation Advisor displays this check.

  For example, you might exclude a check from the Code Generation Advisor when a check takes a long time to process.

- `removeInheritedCheck` — Remove inherited checks from a new objective that is based on an existing objective. When a user selects multiple objectives, if another selected objective includes this check, the Code Generation Advisor displays the check.

  For example, you might remove an inherited check, rather than exclude the check, when the check takes a long time to process, but the check is important for another objective.

Determining Checks and Parameters in Existing Objectives

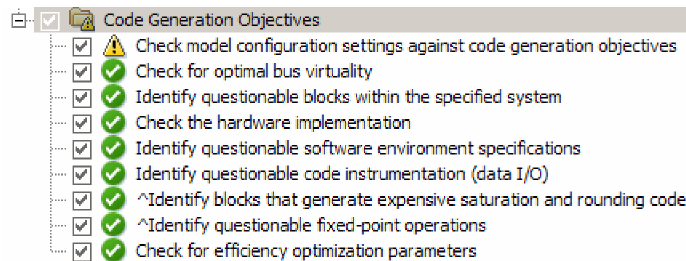
When you base a new objective on an existing objective, you can determine what checks and parameters the existing objective contains. Select the

existing objective and check the model. The Code Generation Advisor lists the checks and parameters associated with the existing objective.

The Code Generation Advisor contains the list of checks in each objective. For example, the `Efficiency` objective includes ten checks, which you can see in the Code Generation Advisor if you:

1. Open the `rtwdemo_rtwecintro` model.
2. Specify an ERT-based target.
3. Specify the `Execution efficiency` objective.
4. Check the model.

The Code Generation Advisor displays in the right pane the list of checks in the `Execution efficiency` objective.



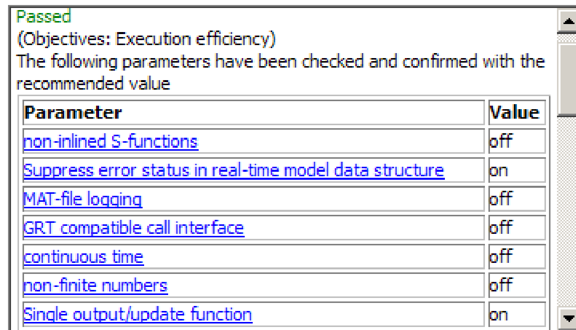
Caution The following objectives *exclude* the listed checks. For more information about excluding checks, see `excludeCheck`.

Objective	Excluded Checks
Traceability	• Identify questionable software environment specifications • Identify questionable code instrumentation (data I/O)
Debugging	Identify questionable code instrumentation (data I/O)

The first check, **Check model configuration settings against code generation objectives**, lists all parameters and values specified by the objective. For example, the Code Generation Advisor displays the list of parameters and the recommended values in the `Execution efficiency` objective, if you:

1. Run **Check model configuration settings against code generation objectives**.
2. Click **Modify Parameters**.
3. Rerun the check.

The Code Generation Advisor displays in the check results the list of parameters and recommended values in the `Execution efficiency` objective.



How to Create Custom Objectives

To create a custom objective:

1. Create an `sl_customization.m` file.

Note

- Specify all custom objectives in a single `sl_customization.m` file only, or the software generates an error. This holds true even if you have more than one `sl_customization.m` file on your MATLAB path.
- Except for the *`matlabroot`*`/work` folder, do not place an `sl_customization.m` file in your root MATLAB folder, or any of its subfolders. Otherwise, the software ignores the customizations that the file specifies.

---

2 Create an `sl_customization` function that takes a single argument. When the software invokes the function, the value of this argument is the Simulink customization manager. In the function:

   a Create a handle to the code generation objective, using the `ObjectiveCustomizer` constructor.

   b Register a callback function for the custom objectives, using the `ObjectiveCustomizer.addCallbackObjFcn` method.

   c Add a call to execute the callback function, using the `ObjectiveCustomizer.callbackFcn` method.

   For example

```
function sl_customization(cm)
%SL_CUSTOMIZATION objective customization callback

objCustomizer = cm.ObjectiveCustomizer;
index = objCustomizer.addCallbackObjFcn(@addObjectives);
objCustomizer.callbackFcn{index}();

end
```

3 Create a MATLAB callback function that:

- Creates code generation objective objects using the `rtw.codegenObjectives.Objective` constructor.

- Adds, modifies, and removes configuration parameters for each objective using the `addParam`, `modifyInheritedParam`, and `removeInheritedParam` methods.
- Includes and excludes checks for each objective using the `addCheck`, `excludeCheck`, and `removeInheritedCheck` methods.
- Registers objectives using the `register` method.

The following example shows you how to create an objective, `Reduce RAM Example`. `Reduce RAM Example` includes five parameters and three checks that the Code Generation Advisor reviews:

```
function addObjectives

% Create the custom objective
obj = rtw.codegenObjectives.Objective('ex_ram_1');
setObjectiveName(obj, 'Reduce RAM Example');

% Add parameters to the objective
addParam(obj, 'InlineParams', 'on');
addParam(obj, 'BooleanDataType', 'on');
addParam(obj, 'OptimizeBlockIOStorage', 'on');
addParam(obj, 'EnhancedBackFolding', 'on');
addParam(obj, 'BooleansAsBitfields', 'on');

% Add additional checks to the objective
% The Code Generation Advisor automatically includes 'Check model
% configuration settings against code generation objectives' in every
% objective.
addCheck(obj, 'Identify unconnected lines, input ports, and output ports');
addCheck(obj, 'Check model and local libraries for updates');

%Register the objective
register(obj);

end
```

The following example shows you how to create an objective, `My Traceability Example`, based on the existing Traceability objective. The custom objective modifies, removes, and adds parameters that the Code

Generation Advisor reviews. It also adds and removes checks from the Code Generation Advisor:

```
function addObjectives

% Create the custom objective from an existing objective
obj = rtw.codegenObjectives.Objective('ex_my_trace_1', 'Traceability');
setObjectiveName(obj, 'My Traceability Example');

% Modify parameters in the objective
modifyInheritedParam(obj, 'GenerateTraceReportSf', 'Off');
removeInheritedParam(obj, 'ConditionallyExecuteInputs');
addParam(obj, 'MatFileLogging', 'On');

% Modify checks in the objective
addCheck(obj, 'Identify questionable software environment specifications');
removeInheritedCheck(obj, 'Identify questionable code instrumentation (data I/O)');

%Register the objective
register(obj);

end
```

4. If you previously opened the Code Generation Advisor, close the model from which you opened the Code Generation Advisor.

5. Refresh the customization manager. At the MATLAB command line, enter the `sl_refresh_customizations` command.

6. Open your model and review the new objectives.

Selecting and Configuring an Embedded Real-Time Target

- "Introduction" on page 16-2
- "Selecting an ERT Target" on page 16-4
- "Customizing an ERT Target" on page 16-6

Introduction

The first step to configuring a model for code generation is to choose and configure a code generation target. When you select a target, other model configuration parameters change automatically to best serve requirements of the target. For example:

- Code interface parameters
- Build process parameters, such as the template make file
- Target hardware parameters, such as word size and byte ordering

Use the **Browse** button on the **Code Generation** pane to open the System Target File Browser (see "Selecting a Target" in the Simulink Coder documentation). The browser lets you select a preset target configuration consisting of a system target file, template makefile, and `make` command.

If you select a target configuration by using the System Target File Browser, your selection appears in the **System target file** field (*`target`*`.tlc`).

If you are using a target configuration that does not appear in the System Target File Browser, enter the name of your system target file in the **System target file** field. Click **Apply** or **OK** to configure for that target.

"Selecting and Configuring a Target" in the Simulink Coder documentation describes the use of the browser and includes a complete list of available target configurations.

You also can select a system target file programmatically from MATLAB code, as described in "Selecting a System Target File Programmatically" in the Simulink Coder documentation.

After selecting a system target, you can modify model configuration parameter settings, if necessary.

If you want to switch between different targets in a single workflow for different code generation purposes (for example, rapid prototyping versus product code deployment), set up different configuration sets for the same model and switch the active configuration set for the current operation. For more information on how to set up configuration sets and change the

active configuration set, see “Setting Up Configuration Sets” in the Simulink documentation.

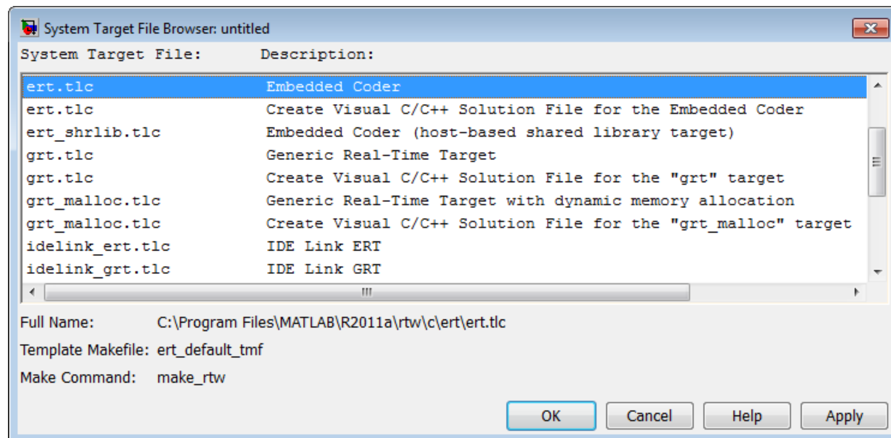
Selecting an ERT Target

The **Browse** button in the **Target Selection** subpane of the **Code Generation > General** pane lets you select an ERT target with the System Target File Browser. See "Selecting and Configuring a Target" in the Simulink Coder documentation for a general discussion of target selection.

The code generator provides variants of the ERT target including the following:

- Default ERT target
- ERT target for generating and building a Visual C++ Solution (`.sln`) file for the Visual C++ IDE
- ERT target for generating a Windows or UNIX host-based shared library

These targets are based on a common system target file, `ert.tlc`. They are displayed in the System Target File Browser as shown in the figure below.



You can use the `ert_shrlib.tlc` target to generate a host-based shared library from your Simulink model. Selecting this target allows you to generate a shared library version of your model code that is appropriate for your host platform, either a Windows dynamic link library (`.dll`) file or a UNIX shared object (`.so`) file. This feature can be used to package your source code securely

for easy distribution and shared use. For more information, see "Creating and Using Host-Based Shared Libraries" on page 4-17.

Customizing an ERT Target

For information on customizing ERT targets, see "Customizing Targets" in the Simulink Coder documentation.

17

Specifying Code Appearance and Documentation

- “Customizing Comments in Generated Code” on page 17-2
- “Configuring the Appearance of Generated Identifiers” on page 17-12
- “Controlling Code Style” on page 17-22
- “Configuring Templates for Customizing Code Organization and Format” on page 17-23
- “Configuring the Placement of Data in Generated Code” on page 17-68
- “Ensuring Delimiter Is Specified for All #Includes” on page 17-69

Customizing Comments in Generated Code

In this section...

"Adding Custom Comments to Generated Code" on page 17-2

"Adding Global Comments" on page 17-5

Adding Custom Comments to Generated Code

You can customize the comments in the generated code for ERT targets by setting or clearing several parameters on the **Code Generation > Comments** pane. These options let you enable or suppress generation of descriptive information in comments for blocks and other objects in the model.

To...	Select...
Include the text specified in the **Description** field of a block's Block Properties dialog box as comments in the code generated for each block	**Simulink block descriptions**.
Add a comment that includes the block name at the start of the code for each block	**Simulink block descriptions**
Include the text specified in the **Description** field of a Simulink data object (such as a signal, parameter, data type, or bus) in the Simulink Model Explorer as comments in the code generated for each object	**Simulink data object descriptions**.
Include comments just above signals and parameter identifiers in the generated code as specified in the MATLAB or TLC function.	**Custom comments (MPT objects only)**.

To...	Select...
Include the text specified in the **Description** field of the Properties dialog box for a Stateflow object as comments just above the code generated for each object	**Stateflow object descriptions** .
Include requirements assigned to Simulink blocks in the generated code comments (for more information, see "Including Requirements Information with Generated Code" in the Simulink Verification and Validation documentation)	**Requirements in block comments**.

When you select **Simulink block descriptions**,

- The description text for blocks and Stateflow objects and block names generated as comments can include international (non-US-ASCII) characters. (For details on international character support, see "Support for International (Non-US-ASCII) Characters" in the Simulink Coder documentation.)
- The code generation software automatically inserts comments into the generated code for custom blocks. Therefore, it is not necessary to include block comments in the associated TLC file for a custom block.

Note If you have existing TLC files with manually inserted comments for block descriptions, the code generation process emits these comments instead of the automatically generated comments. Consider removing existing block comments from your TLC files. Manually inserted comments might be poorly formatted in the generated code and code-to-model traceability might not work.

- For virtual blocks or blocks that have been removed due to block reduction, no comments are generated.

For more information, see "Code Generation Pane: Comments" in the Simulink Coder documentation.

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

You may include at least some or all of the property values for the data object. Each Simulink data object (signal or parameter) has properties, as described in Parameter and Signal Property Values on page 7-2. This example comment contains some of the property values for the data object `MAP` as specified on the Model Explorer:

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

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

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

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

   The MATLAB function must have three arguments that correspond to `objectName`, `modelName`, and `request`, respectively. The TLC function

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

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

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

4. Click **Comments** under **Code Generation** 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

- "Introduction" on page 17-5
- "Using a Simulink DocBlock to Add a Comment" on page 17-6
- "Using a Simulink Annotation to Add a Comment" on page 17-9
- "Using a Stateflow Note to Add a Comment" on page 17-9
- "Using Sorted Notes to Add Comments" on page 17-10

Introduction

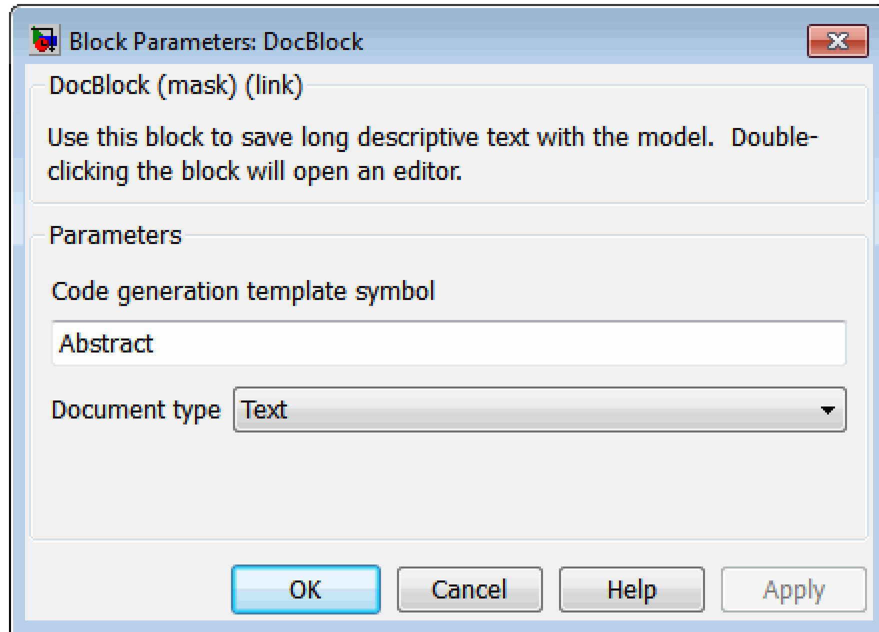
The procedures in this section explain how to add a global comment to a Simulink model so that the comment text appears in the generated file or files where desired. This is accomplished by specifying a template symbol name

with a Simulink DocBlock, a Simulink annotation, or a Stateflow note, or by using a sorted-notes capability that works with Simulink annotations or Stateflow notes (but not DocBlocks). For more information about template symbols, see "Template Symbols and Rules" on page 17-59.

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

Using a Simulink DocBlock to Add a Comment

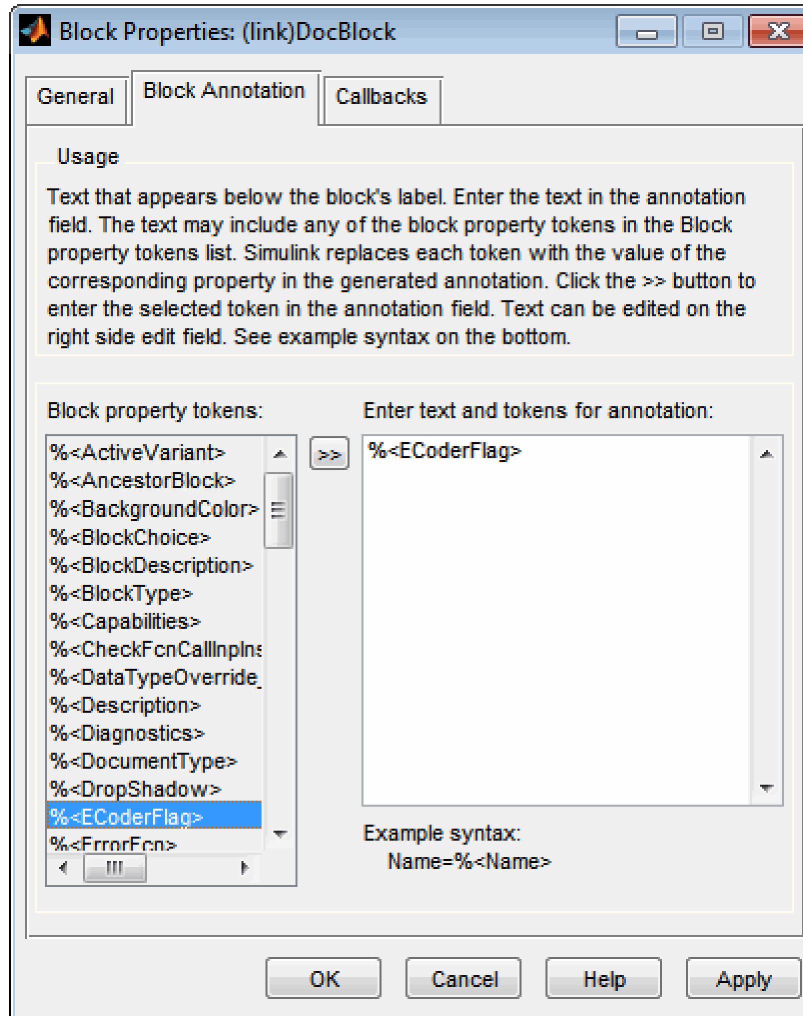
1. With the model open, select **Library Browser** from the **View** menu.
2. Drag the DocBlock from **Model-Wide Utilities** in the Simulink library onto the model.
3. After double-clicking the DocBlock and typing the desired comment in the editor, save and close the editor. See DocBlock in the Simulink documentation for details.
4. Right-click the DocBlock and select **Mask Parameters**. The Block Parameters dialog box appears.
5. Type one of the following into the **Code generation template symbol** field, illustrated below, and then click **OK**: `Abstract`, `Description`, `History`, `ModifiedHistory`, or `Notes`. Template symbol names are case sensitive.



Note If you are using a DocBlock to add comments to your code during code generation, ensure that you set the **Document Type** as `Text`. If you set the **Document Type** as `RTF` or `HTML`, your comments will not appear in the code.

---

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



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

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

Using a Simulink Annotation to Add a Comment

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

Note If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with “Using Sorted Notes to Add Comments” on page 17-10.

---

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

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

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

Using a Stateflow Note to Add a Comment

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

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

Note If you want the code generator to sort multiple comments for the `Notes` symbol name, replace the next step with “Using Sorted Notes to Add Comments” on page 17-10.

---

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

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

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

Using Sorted Notes to Add Comments

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

The sorting order the code generator uses is

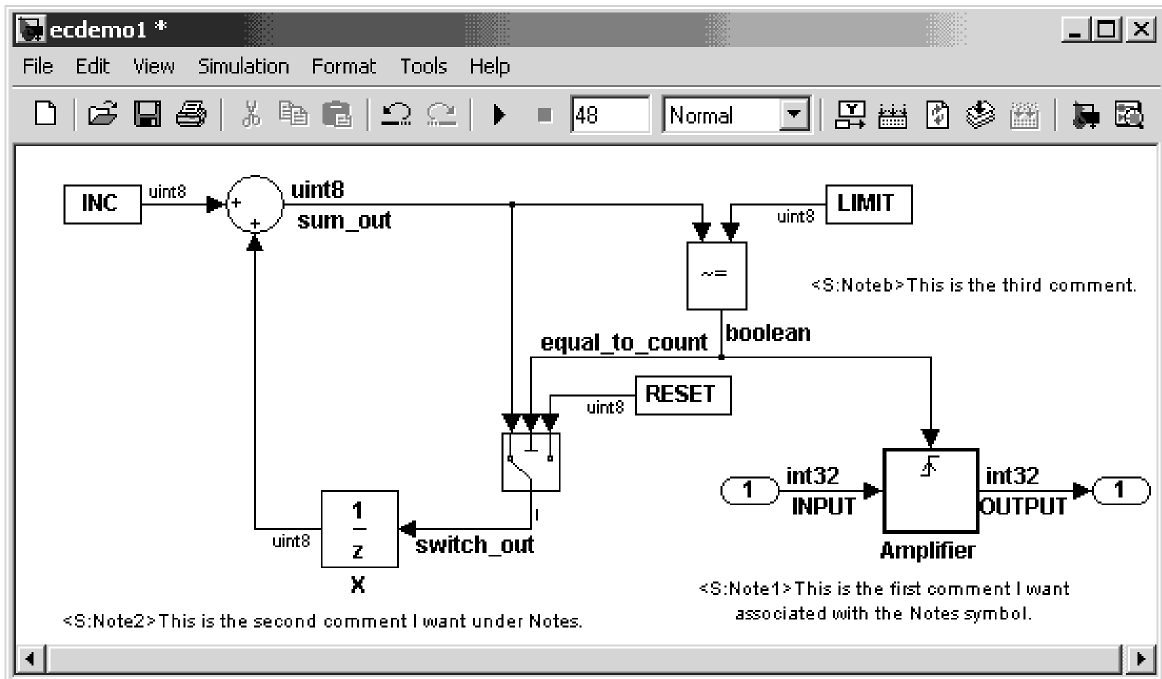
- Numbers before letters
- Among numbers, 0 is first
- Among letters, uppercase are before lowercase.

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

- In the Simulink annotation or the Stateflow note, type `<S:NoteY>` followed by the first comment, where `Y` is a number or letter.

- Repeat for as many additional comments you want, except replace Y with a subsequent number or letter.

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



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

```
** NOTES

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

**
```

Configuring the Appearance of Generated Identifiers

In this section...
"Customizing Generated Identifiers" on page 17-12
"Configuring Symbols" on page 17-13

Customizing Generated Identifiers

Several parameters are available for customizing generated symbols.

To...	Specify...
Define a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers for global variables, global types, field names of global types, subsystem methods, subsystem method arguments, local temporary variables, local block output variables, and constant macros	The macro string with the **Identifier format control** parameter (for details on how to specify formats, see "Specifying Identifier Formats" on page 17-13 and for limitations, see "Identifier Format Control Parameters Limitations" on page 17-19).
Specify the minimum number of characters the code generator uses for mangled symbols	Specify an integer value for the **Minimum mangle length** (for details, see "Name Mangling" on page 17-16).
Specify the maximum number of characters the code generator can use for function, `typedef`, and variable names (default 31)	Specify an integer value for the **Maximum identifier length**. If you expect your model to generate lengthy identifiers (due to use of long signal or parameter names, for example), or you find that identifiers are being mangled more than expected, you should increase the value of this parameter.
Control whether scalar inlined parameter values are expressed in generated code as literal values or macros	The value `Literals` or `Macros` for the **Generate scalar inlined parameters as** parameter . • `Literals`: Parameters are expressed as numeric constants and takes effect if **Inline parameters** is selected.

To...	Specify...
	• `Macros`: Parameters are expressed as variables (with `#define` macros). This setting makes code more readable.

For more information, see "Code Generation Pane: Symbols" in the Simulink Coder documentation.

Configuring Symbols

Specifying Simulink Data Object Naming Rules

To Define Rules that Change the Names of a Model's...	Specify a Naming Rule with the ...
Signals	**Signal naming** parameter
Parameters	**Parameter naming** parameter
Parameters that have a storage class of `Define`	**#define naming** parameter

For more information on these parameters, see "Specifying Simulink Data Object Naming Rules" on page 12-34.

Specifying Identifier Formats

The **Identifier format control** parameters let you customize generated identifiers by entering a macro string that specifies whether, and in what order, certain substrings are included within generated identifiers. For example, you can specify that the root model name be inserted into each identifier.

The macro string can include

- Tokens of the form `$X`, where `X` is a single character. Valid tokens are listed in Identifier Format Tokens on page 17-14. You can use or omit tokens as you want, with the exception of the `$M` token, which is required (see "Name

Mangling” on page 17-16) and subject to the use and ordering restrictions noted in Identifier Format Control Parameter Values on page 17-15.

- Any valid C or C++ language identifier characters (`a-z, A-Z, _ , 0-9`).

The build process generates each identifier by expanding tokens (in the order listed in Identifier Format Tokens on page 17-14) and inserting the resultant strings into the identifier. Character strings between tokens are simply inserted directly into the identifier. Contiguous token expansions are separated by the underscore (_) character.

Identifier Format Tokens

Token	Description
`$M`	Insert name mangling string if required to avoid naming collisions (see “Name Mangling” on page 17-16). **Note:** This token is required.
`$F`	Insert method name (for example, `_Update` for update method). This token is available only for subsystem methods.
`$N`	Insert name of object (block, signal or signal object, state, parameter or parameter object) for which identifier is being generated.
`$R`	Insert root model name into identifier, replacing any unsupported characters with the underscore (_) character. Note that when using model referencing, this token is required in addition to `$M` (see “Model Referencing Considerations” on page 17-18). **Note:** This token replaces the **Prefix model name to global identifiers** option used in previous releases.
`$H`	Insert tag indicating system hierarchy level. For root-level blocks, the tag is the string `root_`. For blocks at the subsystem level, the tag is of the form `sN_`, where `N` is a unique system number assigned by the Simulink software. This token is available only for subsystem methods and field names of global types. **Note:** This token replaces the **Include System Hierarchy Number in Identifiers** option used in previous releases.

Identifier Format Tokens (Continued)

Token	Description
$A	Insert data type acronym (for example, i32 for long integers) to signal and work vector identifiers. This token is available only for local block output variables and field names of global types. **Note:** This token replaces the **Include data type acronym in identifier** option used in previous releases.
$I	Insert u if the argument is an input or y if the argument is an output, (for example, rtu_ for an input argument and rty_ for an output argument). This token is available only for subsystem method arguments.

Identifier Format Control Parameter Values on page 17-15 lists the default macro string, the supported tokens, and the applicable restrictions for each **Identifier format control** parameter.

Identifier Format Control Parameter Values

Parameter	Default Value	Supported Tokens	Restrictions
Global variables	RN$M	$R, $N, $M	$F, $H, $A, and $I are disallowed.
Global types	NR$M	$N, $R, $M	$F, $H, $A, and $I are disallowed.
Field name of global types	NM	$N, $M, $H, $A	$R, $F, and $I are disallowed.
Subsystem methods	RNMF	$R, $N, $M, $F, $H	$F and $H are empty for Stateflow functions; $A and $I are disallowed.
Subsystem method arguments	rtu_NM or rty_NM	$N, $M, $I	$R, $F, $H, and $A are disallowed.

Identifier Format Control Parameter Values (Continued)

Parameter	Default Value	Supported Tokens	Restrictions
Local temporary variables	`$N$M`	`$N`, `$M`, `$R`	`$F`, `$H`, `$A`, and `$I` are disallowed.
Local block output variables	`rtb_$N$M`	`$N`, `$M`, `$A`	`$R`, `$F`, `$H`, and `$I` are disallowed.
Constant macros	`$R$N$M`	`$R`, `$N`, `$M`	`$F`, `$H`, `$A`, and `$I` are disallowed.

Non-ERT based targets (such as the GRT target) implicitly use a default `$R$N$M` specification. This specifies identifiers consisting of the root model name, followed by the name of the generating object (signal, parameter, state, and so on), followed by a name mangling string (see "Name Mangling" on page 17-16).

For limitations that apply to **Identifier format control** parameters, see "Identifier Format Control Parameters Limitations" on page 17-19.

Name Mangling

In identifier generation, a circumstance that would cause generation of two or more identical identifiers is called a *name collision*. Name collisions are never permissible. When a potential name collision exists, unique *name mangling* strings are generated and inserted into each of the potentially conflicting identifiers. Each name mangling string is guaranteed to be unique for each generated identifier.

The position of the `$M` token in the **Identifier format control** parameter specification determines the position of the name mangling string in the generated identifiers. For example, if the specification `$R$N$M` is used, the name mangling string is appended (if required) to the end of the identifier.

The **Minimum mangle length** parameter specifies the minimum number of characters used when a name mangling string is generated. The default

is 1 character. As described below, the actual length of the generated string may be longer than this minimum.

Traceability

An important aspect of model based design is the ability to generate identifiers that can easily be traced back to the corresponding entities within the model. To ensure traceability, it is important to make sure that incremental revisions to a model have minimal impact on the identifier names that appear in generated code. There are two ways of achieving this:

1. Choose unique names for Simulink objects (blocks, signals, states, and so on) as much as possible.

2. Make use of name mangling when conflicts cannot be avoided.

When conflicts cannot be avoided (as may be the case in models that use libraries or model reference), name mangling ensures traceability. The position of the name mangling string is specified by the placement of the `$M` token in the **Identifier format control** parameter specification. Mangle characters consist of lower case characters (`a-z`) and numerics (`0-9`), which are chosen with a checksum that is unique to each object. How Name Mangling Strings Are Computed on page 17-17 describes how this checksum is computed for different types of objects.

How Name Mangling Strings Are Computed

Object Type	Source of Mangling String
Block diagram	Name of block diagram
Simulink block	Full path name of block
Simulink parameter	Full name of parameter owner (that is, model or block) and parameter name
Simulink signal	Signal name, full name of source block, and port number
Stateflow objects	Complete path to Stateflow block and Stateflow computed name (unique within chart)

The length of the name mangling string is specified by the **Minimum mangle length** parameter. The default value is `1`, but this automatically increases during code generation as a function of the number of collisions.

To minimize disturbance to the generated code during development, specify a larger **Minimum mangle length**. A **Minimum mangle length** of `4` is a conservative and safe value. A value of `4` allows for over 1.5 million collisions for a particular identifier before the mangle length is increased.

Minimizing Name Mangling

Note that the length of generated identifiers is limited by the **Maximum identifier length** parameter. When a name collision exists, the `$M` token is always expanded to the minimum number of characters required to avoid the collision. Other tokens and character strings are expanded in the order listed in Identifier Format Tokens on page 17-14. If the **Maximum identifier length** is not large enough to accommodate full expansions of the other tokens, partial expansions are used. To avoid this outcome, it is good practice to

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

  Set the **Minimum mangle length** parameter to reserve at least three characters for the name mangling string. The length of the name mangling string increases as the number of name collisions increases.

  Note that an existing name mangling string increases (or decreases) in length if changes to model create more (or fewer) collisions. If the length of the name mangling string increases, additional characters are appended to the existing string. For example, `'xyz'` might change to `'xyzQ'`. In the inverse case (fewer collisions) `'xyz'` would change to `'xy'`.

Model Referencing Considerations

Within a model that uses model referencing, there can be no collisions between the names of the constituent models. When generating code from a model that uses model referencing:

- The `$R` token must be included in the **Identifier format control** parameter specifications (in addition to the `$M` token).
- The **Maximum identifier length** must be large enough to accommodate full expansions of the `$R` and `$M` tokens. A code generation error occurs if **Maximum identifier length** is not large enough.

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

Exceptions to Identifier Formatting Conventions

There are some exceptions to the identifier formatting conventions described above:

- Type name generation: The above name mangling conventions do not apply to type names (that is, `typedef` statements) generated for global data types. If the `$R` token is included in the **Identifier format control** parameter specification, the model name is included in the `typedef`. The **Maximum identifier length** parameter is not respected when generating type definitions.
- Non-`Auto` storage classes: The **Identifier format control** parameter specification does not affect objects (such as signals and parameters) that have a storage class other than `Auto` (such as `ImportedExtern` or `ExportedGlobal`).

Identifier Format Control Parameters Limitations

The following limitations apply to the **Identifier format control** parameters:

- The following autogenerated identifiers currently do not fully comply with the setting of the **Maximum identifier length** parameter on the **Code Generation > Symbols** pane of the Configuration Parameters dialog box.
  - Model methods
    - The applicable format string is `$R$F`, and the longest `$F` is `_derivatives`, which is 12 characters long. The model name can

be up to 19 characters without exceeding the default **Maximum identifier length** of 31.

- Local functions generated by S-functions or by add-on products such as DSP System Toolbox™ that rely on S-functions
- Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
- `DWork` identifiers generated by S-functions in referenced models
- Fixed-point shared utility macros or shared utility functions
- Simulink `rtm` macros
  - Most are within the default **Maximum identifier length** of 31, but some exceed the limit. Examples are `RTMSpecAccsGetStopRequestedValStoredAsPtr`, `RTMSpecAccsGetErrorStatusPointer`, and `RTMSpecAccsGetErrorStatusPointerPointer`.
- Define protection guard macros
  - Header file guards, such as `_RTW_HEADER_$(filename)_h_`, which can exceed the default **Maximum identifier length** of 31 given a filename such as `$R_private.h`.
  - Include file guards, such as `_$R_COMMON_INCLUDES_`.
  - `typedef` guards, such as `_CSCI_$R_CHARTSTRUCT_`.

- In some situations, the following identifiers potentially can conflict with others.
  - Model methods
  - Reentrant model function arguments
  - Local functions generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Local variables generated by S-functions or by add-on products such as DSP System Toolbox that rely on S-functions
  - Fixed-point shared utility macros or shared utility functions
  - Include header guard macros

- The following external identifiers that are unknown to the Simulink software might conflict with autogenerated identifiers.
  - Identifiers defined in custom code
  - Identifiers defined in custom header files
  - Identifiers introduced through a non-ANSI C standard library
  - Identifiers defined by custom TLC code
- Identifiers generated for simulation targets may exceed the **Maximum identifier length**. Simulation targets include the model reference simulation target, the accelerated simulation target, the RSim target, and the S-function target.

Controlling Code Style

You can control the following style aspects in generated code:

- Level of parenthesization
- Whether to preserve order of operands in expressions
- Whether to preserve empty primary condition expressions in `if` statements
- Whether to generate code for `if-elseif-else` decision logic as `switch-case` statements
- Whether to include the `extern` keyword in function declarations
- Whether to always generate default cases for switch-case statements in the code for Stateflow charts

For example, C code contains some syntactically required parentheses, and can contain additional parentheses that change semantics by overriding default operator precedence. C code can also contain optional parentheses that have no functional significance, but serve only to increase the readability of the code. Optional C parentheses vary between two stylistic extremes:

- Include the minimum parentheses required by C syntax and any precedence overrides, so that C precedence rules specify all semantics unless overridden by parentheses.
- Include the maximum parentheses that can exist without duplication, so that C precedence rules become irrelevant: parentheses alone completely specify all semantics.

Understanding code with minimum parentheses can require correctly applying nonobvious precedence rules, but maximum parentheses can hinder code reading by belaboring obvious precedence rules. Various parenthesization standards exist that specify one or the other extreme, or define an intermediate style that can be useful to human code readers.

You control the code style options by setting parameters on the **Code Generation > Code Style** pane. For details on the parameters, see "Code Generation Pane: Code Style".

Configuring Templates for Customizing Code Organization and Format

In this section...

"Overview" on page 17-24

"Custom File Processing Components" on page 17-25

"Custom File Processing User Interface Options" on page 17-25

"Code Generation Template (CGT) Files" on page 17-27

"Using Custom File Processing (CFP) Templates" on page 17-31

"Custom File Processing (CFP) Template Structure" on page 17-31

"Changing the Organization of a Generated File" on page 17-33

"Generating Source and Header Files with a Custom File Processing (CFP) Template" on page 17-35

"Comparison of a Template and Its Generated File" on page 17-44

"Code Template API Summary" on page 17-47

"Generating Custom File and Function Banners" on page 17-50

"Template Symbols and Rules" on page 17-59

Customize generated code using code and data templates

To...	Enter or Select...
Specify a template that defines the top-level organization and formatting of generated source code (`.c` or `.cpp`) files	Enter a code generation template (CGT) file for the **Source file (*.c) template** parameter.
Specify a template that defines the top-level organization and formatting of generated header (`.h`) files	Enter a CGT file for the **Header file (*.h) template** parameter. This template file can be the same template file that you specify for **Source file (.c) template**. If you use the same template file, source and header files contain identical banners. The default template is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.

To...	Enter or Select...
Specify a template that organizes generated code into sections (such as includes, `typedefs`, functions, and more)	Enter a custom file processing (CFP) template file for the "File customization template" parameter. A CFP template can emit code, directives, or comments into each section. For more information, see "Using Custom File Processing (CFP) Templates" on page 17-31.
Generate a model-specific example main program module	Select **Generate an example main program**. For more information, see "Generating a Standalone Program" on page 34-3.

Note Place the template files that you specify on the MATLAB path.

Overview

This section describes Embedded Coder *custom file processing* (CFP) features. Custom file processing simplifies generation of custom source code. You can:

- Generate any type of source (`.c` or `.cpp`) or header (`.h`) file. Using a *custom file processing template* (CFP template), you can control how code emits to the standard generated model files (for example, `model.c` or `.cpp`, `model.h`) or generate files that are independent of model code.
- Organize generated code into sections (such as includes, `typedefs`, functions, and more). Your CFP template can emit code (for example, functions), directives (such as `#define` or `#include` statements), or comments into each section.
- Generate custom *file banners* (comment sections) at the start and end of generated code files and custom *function banners* that precede functions in the generated code.
- Generate code to call model functions, such as `model_initialize`, `model_step`, and so on.
- Generate code to read and write model inputs and outputs.
- Generate a main program module.
- Obtain information about the model and the generated files from the model.

Custom File Processing Components

The custom file processing features are based on the following interrelated components:

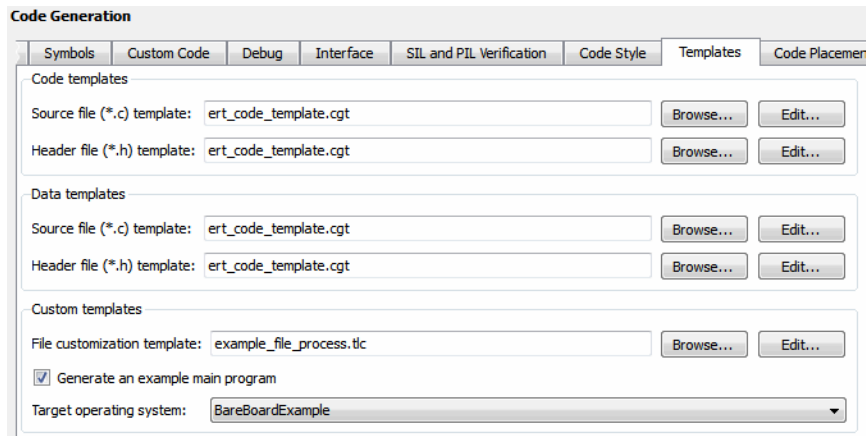
- *Code generation template* (CGT) files: a CGT file defines the top-level organization and formatting of generated code. See "Code Generation Template (CGT) Files" on page 17-27.
- The *code template API*: a high-level Target Language Compiler (TLC) API that provides functions with which you can organize code into named sections and subsections of generated source and header files. The code template API also provides utilities that return information about generated files, generate standard model calls, and perform other functions. See "Code Template API Summary" on page 17-47.
- *Custom file processing (CFP) templates*: a CFP template is a TLC file that manages the process of custom code generation. A CFP template assembles code to be generated into buffers. A CFP template also calls the code template API to emit the buffered code into specified sections of generated source and header files. A CFP template interacts with a CGT file, which defines the ordering of major sections of the generated code. See "Using Custom File Processing (CFP) Templates" on page 17-31.

To use CFP templates, you must understand TLC programming. See the Target Language Compiler document.

Custom File Processing User Interface Options

To use custom file processing features, create CGT files and CFP templates. These files are based on default templates provided by the code generation software. Once you have created your templates, you must integrate them into the code generation process.

Select and edit CGT files and CFP templates, and specify their use in the code generation process in the **Code Generation > Templates** pane of a model configuration set. The following figure shows all options configured for their defaults.



Code Generation: Templates Pane

The options related to custom file processing are:

- The **Source file (.c) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating source (`.c` or `.cpp`) files. You must place this file on the MATLAB path.
- The **Header file (.h) template** field in the **Code templates** and **Data templates** sections. This field specifies the name of a CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path.

  By default, the template for both source and header files is `*matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- The **File customization template** edit field in the **Custom templates** section. This field specifies the name of a CFP template file to use when generating code files. You must place this file on the MATLAB path. The default CFP template is `*matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc`.

In each of these fields, click **Browse** to navigate to and select an existing CFP template or CGT file. Click **Edit** to open the specified file into the MATLAB editor where you can customize it. You must place this file on the

Code Generation Template (CGT) Files

CGT files have the following applications:

- Generation of custom banners (comments sections) in code files. See "Generating Custom File and Function Banners" on page 17-50.
- Advanced features, as described in Defining Data Representation and Storage for Code Generation on page 1 that use CGT files.
- Generation of custom code using a CFP template requires a CGT file. To correctly use CFP templates, you must understand the CGT file structure. In many cases, however, you can use the default CGT file without modifying it.

Default CGT file

The code generation software provides a default CGT file, `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`. Base your custom CGT files on the default file.

CGT File Structure

A CGT file consists of one required section and four optional sections:

Code Insertion Section. (Required) This section contains tokens that define an ordered partitioning of the generated code into a number of sections (such as `Includes` and `Defines` sections). Tokens have the form of:

```
%<SectionName>
```

For example,

```
%<Includes>
```

The code generation software defines a minimal set of required tokens. These tokens generate C or C++ source or header code. They are *built-in* tokens (see "Built-In Tokens and Sections" on page 17-28). You can also define *custom* tokens and add them to the code insertion section (see "Generating a Custom Section" on page 17-42).

Each token functions as a placeholder for a corresponding section of generated code. The ordering of the tokens defines the order in which the corresponding

sections appear in the generated code. A token in the CGT file does not guarantee that the corresponding section is generated. To generate code into a given section, explicitly call the code template API from a CFP template, as described in "Using Custom File Processing (CFP) Templates" on page 17-31.

The CGT tokens define the high-level organization of generated code. Using the code template API, you can partition each code section into named subsections, as described in "Subsections" on page 17-30.

In the code insertion section, you can also insert C or C++ comments between tokens. Such comments emit directly into the generated code.

File Banner Section. (Optional) This section contains comments and tokens you use in generating a custom file banner. See "Generating Custom File and Function Banners" on page 17-50.

Function Banner Section. (Optional) This section contains comments and tokens for use in generating a custom function banner. See"Generating Custom File and Function Banners" on page 17-50.

Shared Utility Function Banner Section. (Optional) This section contains comments and tokens for use in generating a custom shared utility function banner. See "Generating Custom File and Function Banners" on page 17-50.

File Trailer Section. (Optional) This section contains comments for use in generating a custom trailer banner. See "Generating Custom File and Function Banners" on page 17-50.

Built-In Tokens and Sections

The following code extract shows the required code insertion section of the default CGT file with the required built-in tokens.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Code insertion section (required)
%%   These are required tokens. You can insert comments and other tokens in
%% between them, but do not change their order or remove them.
%%
%<Includes>
%<Defines>
```

```
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Note the following requirements for customizing a CGT file:

- Do not remove required built-in tokens.
- Built-in tokens must appear in the order shown because each successive section has dependencies on previous sections.
- Only one token per line.
- Do not repeat tokens.
- You can add custom tokens and comments to the code insertion section as long as you do not violate the previous requirements.

The following table summarizes the built-in tokens and corresponding section names, and describes the code sections.

Built-In CGT Tokens and Corresponding Code Sections

Token and Section Name	Description
`Includes`	`#include` directives section
`Defines`	`#define` directives section
`Types`	`typedef` section. `Typedefs` can depend on any previously defined type
`Enums`	Enumerated types section
`Definitions`	Data definitions (for example, `double x = 3.0`;)
`Declarations`	Data declarations (for example, `extern double x`;)
`Functions`	C or C++ functions

Subsections

You can define one or more named subsections for any section. Some of the built-in sections have predefined subsections summarized in Subsections Defined for Built-In Sections on page 17-30.

Note Sections and subsections emit to the source or header file in the order listed in the CGT file.

Using the custom section feature, you can define additional sections. See "Generating a Custom Section" on page 17-42.

Subsections Defined for Built-In Sections

Section	Subsections	Subsection Description
`Includes`	N/A	
`Defines`	N/A	
`Types`	`IntrinsicTypes`	Intrinsic `typedef` section. Intrinsic types depend only on intrinsic C or C++ types.
`Types`	`PrimitiveTypedefs`	Primitive `typedef` section. Primitive `typedefs` depend only on intrinsic C or C++ types and on any `typedefs` previously defined in the `IntrinsicTypes` section.
`Types`	`UserTop`	You can place any type of code in this section, including code that has dependencies on the previous sections.
`Types`	`Typedefs`	`typedef` section. `Typedefs` can depend on any previously defined type
`Enums`	N/A	
`Definitions`	N/A	
`Declarations`	N/A	
`Functions`		C or C++ functions
`Functions`	`CompilerErrors`	`#warning` directives

Subsections Defined for Built-In Sections (Continued)

Section	Subsections	Subsection Description
`Functions`	`CompilerWarnings`	`#error` directives
`Functions`	`Documentation`	Documentation (comment) section
`Functions`	`UserBottom`	You can place any code in this section.

Using Custom File Processing (CFP) Templates

The files provided to support custom file processing are

- `*matlabroot*/rtw/c/tlc/mw/codetemplatelib.tlc`: A TLC function library that implements the code template API. `codetemplatelib.tlc` also provides the comprehensive documentation of the API in the comments headers preceding each function.
- `*matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc`: An example custom file processing (CFP) template, which you should use as the starting point for creating your own CFP templates. Guidelines and examples for creating a CFP template are provided in "Generating Source and Header Files with a Custom File Processing (CFP) Template" on page 17-35.
- TLC files supporting generation of single-rate and multirate main program modules (see "Customizing Main Program Module Generation" on page 17-40).

Once you have created a CFP template, you must integrate it into the code generation process, using the **File customization template** edit field (see "Custom File Processing User Interface Options" on page 17-25).

Custom File Processing (CFP) Template Structure

A custom file processing (CFP) template imposes a simple structure on the code generation process. The template, a code generation template (CGT) file, partitions the code generated for each file into a number of sections. These sections are summarized in Built-In CGT Tokens and Corresponding Code Sections on page 17-29 and Subsections Defined for Built-In Sections on page 17-30.

Code for each section is assembled in buffers and then emitted, in the order listed, to the file being generated.

To generate a file section, your CFP template must first assemble the code to be generated into a buffer. Then, to emit the section, your template calls the TLC function

```
LibSetSourceFileSection(fileH, section, tmpBuf)
```

where

- `fileH` is a file reference to a file being generated.
- `section` is the code section or subsection to which code is to be emitted. section must be one of the section or subsection names listed in Subsections Defined for Built-In Sections on page 17-30.

  Determine the `section` argument as follows:

  - If Subsections Defined for Built-In Sections on page 17-30 defines no subsections for a given section, use the section name as the `section` argument.
  - If Subsections Defined for Built-In Sections on page 17-30 defines one or more subsections for a given section, you can use either the section name or a subsection name as the `section` argument.
  - If you have defined a custom token denoting a custom section, do not call `LibSetSourceFileSection`. Special API calls are provided for custom sections (see "Generating a Custom Section" on page 17-42).
- `tmpBuf` is the buffer containing the code to be emitted.

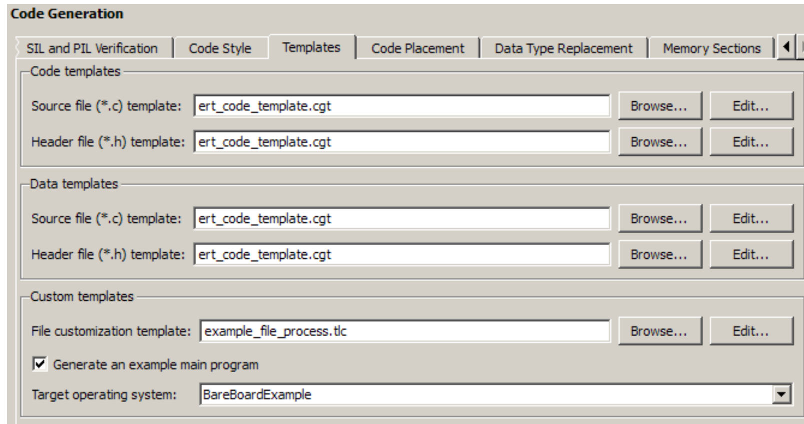
There is no requirement to generate all of the available sections. Your template need only generate the sections you require in a particular file.

Note that no legality or syntax checking is performed on the custom code within each section.

The next section, "Generating Source and Header Files with a Custom File Processing (CFP) Template" on page 17-35, provides typical usage examples.

Changing the Organization of a Generated File

The files you generated in the previous procedures are organized according to the general code generation template. This template has the filename `ert_code_template.cgt`, and is specified by default in **Templates** pane of the Configuration Parameters dialog box.



The following fragment shows the `rtwdemo_mpf.c` file header that is generated using this default template:

```
/*
 * File: rtwdemo_mpf.c
 *
 * Code generated for Simulink model 'rtwdemo_mpf'.
 *
 * Model version                        : 1.88
 * Simulink Coder version               : 8.0  (R2011a)  26-Aug-2010
 * TLC version                          : 7.6 (Sep  3 2010)
 * C/C++ source code generated on       : Thu Sep 09 10:10:14 2010
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Generic->32-bit Embedded Processor
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */
```

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

1. Display the active **Templates** configuration parameters.

2. In the **Code templates** section of the **Templates** pane, type `code_c_template.cgt` into the **Source file (*.c) templates** text box.

3. Type `code_h_template.cgt` into the **Header file (*.h) templates** text box.

4. In the **Data templates** section, type `data_c_template.cgt` into the **Source file (*.c) templates** text box.

5. Type `data_h_template.cgt` into the **Header file (*.h) templates** text box, and click **Apply**.

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

```
/**
 ******************************************************************************
 **  FILE INFORMATION:
 **  Filename:           rtwdemo_mpf.c
 **  File Creation Date: 09-Sep-2010
 **
 **  ABSTRACT:
 **
 **
 **  NOTES:
 **
 **
 **  MODEL INFORMATION:
 **  Model Name:         rtwdemo_mpf
 **  Model Description:  Data packaging examples
 **  Model Version:      1.89
 **  Model Author:       The MathWorks Inc. - Mon Mar 01 11:23:00 2004
 **
 **  MODIFICATION HISTORY:
 **  Model at Code Generation: ssulliva - Thu Sep 09 10:19:35 2010
 **
 **  Last Saved Modification:  ssulliva - Thu Sep 09 10:19:13 2010
 **
 **
 ******************************************************************************
 **/
```

Generating Source and Header Files with a Custom File Processing (CFP) Template

This section walks you through the process of generating a simple source (`.c` or `.cpp`) and header (`.h`) file using the example CFP template. Then, it examines the template and the code generated by the template.

The example CFP template, *matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc, demonstrates some of the capabilities of the code template API, including

- Generation of simple source (.c or .cpp) and header (.h) files
- Use of buffers to generate file sections for includes, functions, and so on
- Generation of includes, defines, into the standard generated files (for example, *model*.h)
- Generation of a main program module

Generating Code with a CFP Template

This section sets up a CFP template and configures a model to use the template in code generation. The template generates (in addition to the standard model files) a source file (timestwo.c or .cpp) and a header file (timestwo.h).

Follow the steps below to become acquainted with the use of CFP templates:

1. Copy the example CFP template, *matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc, to a directory outside the MATLAB directory structure (that is, not under *matlabroot*). If the directory is not on the MATLAB path or the TLC path, then add it to the MATLAB path. It is good practice to locate the CFP template in the same directory as your system target file, which is guaranteed to be on the TLC path.

2. Rename the copied example_file_process.tlc to test_example_file_process.tlc.

3. Open test_example_file_process.tlc into the MATLAB editor.

4. Uncomment the following line:

   ```
   %%  %assign ERTCustomFileTest = TLC_TRUE
   ```

   It now reads:

   ```
     %assign ERTCustomFileTest = TLC_TRUE
   ```

If `ERTCustomFileTest` is not assigned as shown, the CFP template is ignored in code generation.

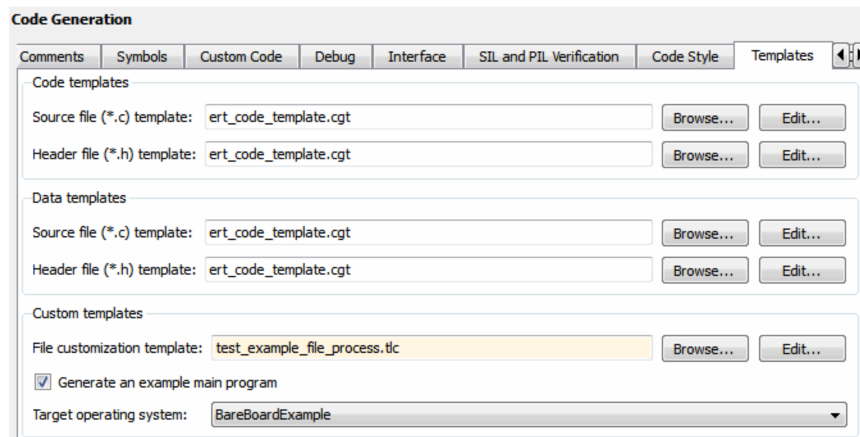
5. Save your changes to the file. Keep `test_example_file_process.tlc` open, so you can refer to it later.

6. Open the `rtwdemo_udt` model.

7. Open the Simulink Model Explorer. Select the active configuration set of the model, and open the **Code Generation** pane of the active configuration set.

8. Click the **Templates** tab.

9. Configure the **File customization template** field as shown below. The `test_example_file_process.tlc` file, which you previously edited, is now the specified CFP template for your model.



10. Select the **Generate code only** option.

11. Click **Apply**.

12. Click **Generate code**. During code generation, notice the following message on the MATLAB command window:

```
Warning:  Overriding example ert_main.c!
```

This message is displayed because `test_example_file_process.tlc` generates the main program module, overriding the default action of the ERT target. This is explained in greater detail below.

13 The `rtwdemo_udt` model is configured to generate an HTML code generation report. After code generation completes, view the report. Notice that the **Generated Files** list contains the files `timestwo.c`, `timestwo.h`, and `ert_main.c`. These files were generated by the CFP template. The next section examines the template to learn how this was done.

14 Keep the model, the code generation report, and the `test_example_file_process.tlc` file open so you can refer to them in the next section.

Analysis of the Example CFP Template and Generated Code

This section examines excerpts from `test_example_file_process.tlc` and some of the code it generates. Refer to the comments in *`matlabroot`*`/rtw/c/tlc/mw/codetemplatelib.tlc` while reading the following discussion.

Generating Code Files. Source (`.c` or `.cpp`) and header (`.h`) files are created by calling `LibCreateSourceFile`, as in the following excerpts:

```
%assign cFile = LibCreateSourceFile("Source", "Custom", "timestwo")
...
%assign hFile = LibCreateSourceFile("Header", "Custom", "timestwo")
```

Subsequent code refers to the files by the file reference returned from `LibCreateSourceFile`.

File Sections and Buffers. The code template API lets you partition the code generated to each file into sections, tagged as `Definitions`, `Includes`, `Functions`, `Banner`, and so on. You can append code to each section as many times as required. This technique gives you a great deal of flexibility in the formatting of your custom code files.

Subsections Defined for Built-In Sections on page 17-30 describes the available file sections and their order in the generated file.

For each section of a generated file, use `%openfile` and `%closefile` to store the text for that section in temporary buffers. Then, to write (append) the buffer contents to a file section, call `LibSetSourceFileSection`, passing in the desired section tag and file reference. For example, the following code uses two buffers (`tmwtypesBuf` and `tmpBuf`) to generate two sections (tagged `"Includes"` and `"Functions"`) of the source file `timestwo.c` or `.cpp` (referenced as `cFile`):

```
 %openfile tmwtypesBuf

 #include "tmwtypes.h"

%closefile tmwtypesBuf

%<LibSetSourceFileSection(cFile,"Includes",tmwtypesBuf)>

 %openfile tmpBuf

 /* Times two function */
 real_T timestwofcn(real_T input) {
   return (input * 2.0);
}

%closefile tmpBuf

%<LibSetSourceFileSection(cFile,"Functions",tmpBuf)>
```

These two sections generate the entire `timestwo.c` or `.cpp` file:

```
#include "tmwtypes.h"

/* Times two function */
FLOAT64 timestwofcn(FLOAT64 input)
{
  return (input * 2.0);
}
```

Adding Code to Standard Generated Files. The timestwo.c or .cpp file generated in the previous example was independent of the standard code files generated from a model (for example, *model*.c or .cpp, *model*.h, and so on). You can use similar techniques to generate custom code within the model files. The code template API includes functions to obtain the names of the standard models files and other model-related information. The following excerpt calls LibGetMdlPubHdrBaseName to obtain the correct name for the *model*.h file. It then obtains a file reference and generates a definition in the Defines section of *model*.h:

```
%% Add a #define to the model's public header file model.h

%assign pubName = LibGetMdlPubHdrBaseName()
%assign modelH  = LibCreateSourceFile("Header", "Simulink", pubName)

%openfile tmpBuf

 #define ACCELERATION 9.81

 %closefile tmpBuf

%<LibSetSourceFileSection(modelH,"Defines",tmpBuf)>
```

Examine the generated rtwdemo_udt.h file to see the generated #define directive.

Customizing Main Program Module Generation. Normally, the ERT target determines whether and how to generate an ert_main.c or .cpp module based on the settings of the **Generate an example main program** and **Target operating system** options on the **Templates** pane of the Configuration Parameters dialog box. You can use a CFP template to override the normal behavior and generate a main program module customized for your target environment.

To support generation of main program modules, two TLC files are provided:

- bareboard_srmain.tlc: TLC code to generate an example single-rate main program module for a bareboard target environment. Code is generated by a single TLC function, FcnSingleTaskingMain.

- bareboard_mrmain.tlc: TLC code to generate a multirate main program module for a bareboard target environment. Code is generated by a single TLC function, FcnMultiTaskingMain.

In the example CFP template file *matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc, the following code generates either a single- or multitasking ert_main.c or .cpp module. The logic depends on information obtained from the code template API calls LibIsSingleRateModel and LibIsSingleTasking:

```
%% Create a simple main.  Files are located in MATLAB/rtw/c/tlc/mw.

 %if LibIsSingleRateModel() || LibIsSingleTasking()
   %include "bareboard_srmain.tlc"
   %<FcnSingleTaskingMain()>
 %else
   %include "bareboard_mrmain.tlc"
   %<FcnMultiTaskingMain()>
 %endif
```

Note that bareboard_srmain.tlc and bareboard_mrmain.tlc use the code template API to generate ert_main.c or .cpp.

When generating your own main program module, you disable the default generation of ert_main.c or .cpp. The TLC variable GenerateSampleERTMain controls generation of ert_main.c or .cpp. You can directly force this variable to TLC_FALSE. The examples bareboard_mrmain.tlc and bareboard_srmain.tlc use this technique, as shown in the following excerpt from bareboard_srmain.tlc.

```
%if GenerateSampleERTMain
    %assign CompiledModel.GenerateSampleERTMain = TLC_FALSE
    %warning Overriding example ert_main.c!
%endif
```

Alternatively, you can implement a SelectCallback function for your target. A SelectCallback function is a MATLAB function that is triggered during model loading, and also when the user selects a target with the System Target File browser. Your SelectCallback function should deselect and disable the **Generate an example main program** option. This prevents the TLC variable GenerateSampleERTMain from being set to TLC_TRUE.

See the “rtwgensettings Structure” section in the Simulink Coder documentation for information on creating a `SelectCallback` function.

The following code illustrates how to deselect and disable the **Generate an example main program** option in the context of a `SelectCallback` function.

```
slConfigUISetVal(hDlg, hSrc, 'GenerateSampleERTMain', 'off');
slConfigUISetEnabled(hDlg, hSrc, 'GenerateSampleERTMain',0);
```

Note Creation of a main program for your target environment requires some customization; for example, in a bareboard environment you need to attach `rt_OneStep` to a timer interrupt. It is expected that you will customize either the generated code, the generating TLC code, or both. See “Guidelines for Modifying the Main Program” on page 34-5 and “Guidelines for Modifying rt_OneStep” on page 34-12 for further information.

Generating a Custom Section

You can define custom tokens in a CGT file and direct generated code into an associated built-in section. This feature gives you additional control over the formatting of code within each built-in section. For example, you could add subsections to built-in sections that do not already define any subsections. All custom sections must be associated with one of the built-in sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`. To create custom sections, you must

- Add a custom token to the code insertion section of your CGT file.
- In your CFP file:
  - Assemble code to be generated to the custom section into a buffer.
  - Declare an association between the custom section and a built-in section, with the code template API function `LibAddSourceFileCustomSection`.
  - Emit code to the custom section with the code template API function `LibSetSourceFileCustomSection`.

The following code examples illustrate the addition of a custom token, `Myincludes`, to a CGT file, and the subsequent association of the custom section `Myincludes` with the built-in section `Includes` in a CFP file.

Note If you have not already created custom CGT and CFP files for your model, copy the default template files *matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt and *matlabroot*/toolbox/rtw/targets/ecoder/example_file_process.tlc to a work directory that is outside the MATLAB directory structure but on the MATLAB or TLC path, rename them (for example, add the prefix test_ to each file), and update the **Templates** pane of the Configuration Parameters dialog box to correctly reference them.

First, add the token Myincludes to the code insertion section of your CGT file. For example:

```
%<Includes>
%<Myincludes>
%<Defines>
%<Types>
%<Enums>
%<Definitions>
%<Declarations>
%<Functions>
```

Next, in the CFP file, add code to generate include directives into a buffer. For example, in your copy of the example CFP file, you could insert the following section between the Includes section and the Create a simple main section:

```
%% Add a custom section to the model's C file model.c

%openfile tmpBuf
#include "moretables1.h"
#include "moretables2.h"
%closefile tmpBuf

%<LibAddSourceFileCustomSection(modelC,"Includes","Myincludes")>
%<LibSetSourceFileCustomSection(modelC,"Myincludes",tmpBuf)>
```

The LibAddSourceFileCustomSection function call declares an association between the built-in section Includes and the custom section Myincludes. In effect, Myincludes is a subsection of Includes.

The LibSetSourceFileCustomSection function call directs the code in the tmpBuf buffer to the Myincludes section of the generated file. LibSetSourceFileCustomSection is syntactically identical to LibSetSourceFileSection.

In the generated code, the include directives generated to the custom section appear after other code directed to Includes.

```
#include "rtwdemo_udt.h"
#include "rtwdemo_udt_private.h"

/* #include "mytables.h" */
#include "moretables1.h"
#include "moretables2.h"
```

Note The placement of the custom token in this example CGT file is arbitrary. By locating %<Myincludes> after %<Includes>, the CGT file ensures only that the Myincludes code appears after Includes code.

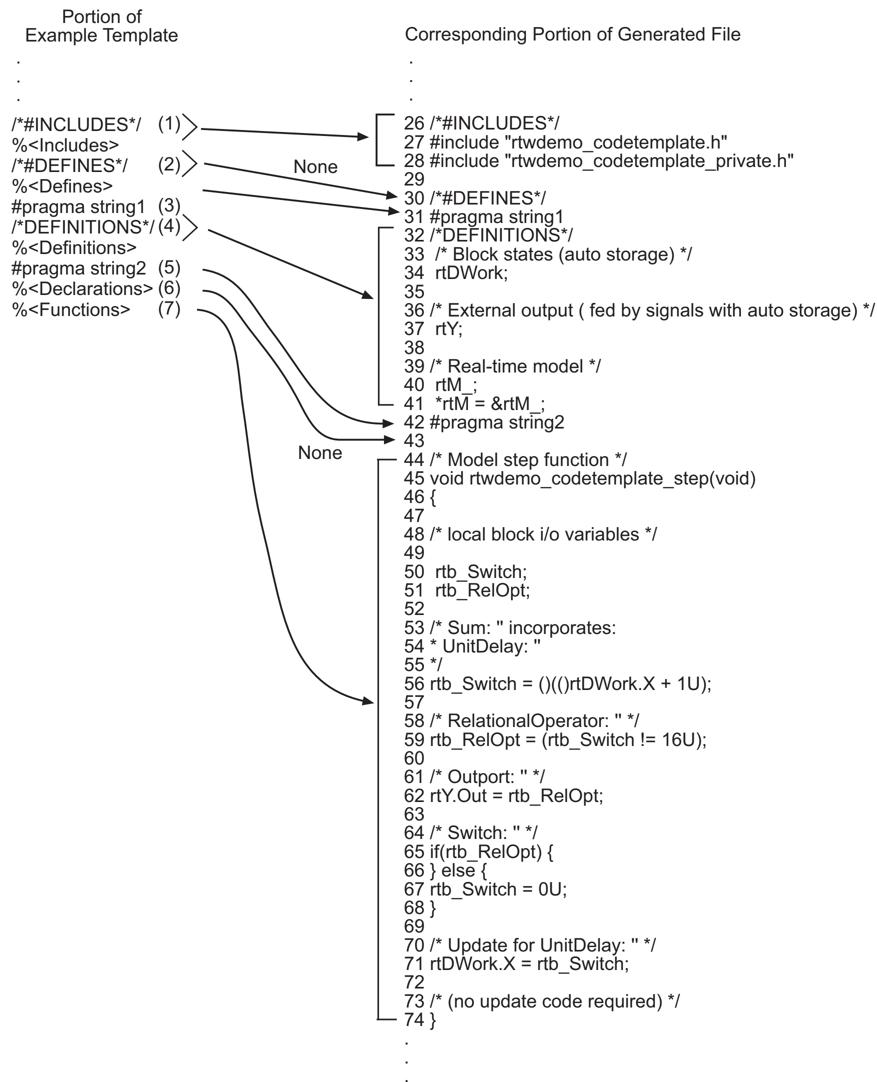
Comparison of a Template and Its Generated File

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

- Define what code the code generation software should add to the generated file
- Control the location of code in the file
- Optionally insert comments in the generated file

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

Template and Generated File



How the Template Affects Code Generation

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

How the Template Affects Code Generation (Continued)

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

For a list of template symbols and the rules for using them, see "Template Symbol Groups" on page 17-60, "Template Symbols" on page 17-62, and "Rules for Modifying or Creating a Template" on page 17-66. To set comment options, from the **Simulation** menu, select **Configuration Parameters**. On the Configuration Parameters dialog box, select the **Code Generation > Comments** pane. For details, see "Configuring a Model for Code Generation" in the Simulink Coder documentation.

Code Template API Summary

Code Template API Functions on page 17-48 summarizes the code template API. See the source code in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc` for detailed information on the arguments, return values, and operation of these calls.

Code Template API Functions

Function	Description
LibGetNumSourceFiles	Returns the number of created source files (.c or .cpp and .h).
LibGetSourceFileTag	Returns <filename>_h and <filename>_c for header and source files, respectively, where filename is the name of the model file.
LibCreateSourceFile	Creates a new C or C++ file and returns its reference. If the file already exists, simply returns its reference.
LibGetSourceFileFromIdx	Returns a model file reference based on its index. This is useful for a common operation on all files, such as to set the leading file banner of all files.
LibSetSourceFileSection	Adds to the contents of a specified section within a specified file (see also "Custom File Processing (CFP) Template Structure" on page 17-31).
LibIndentSourceFile	Indents a file with the c_indent utility (from within the TLC environment).
LibCallModelInitialize	Returns code for calling the model's *model*_initialize function (valid for ERT only).
LibCallModelStep	Returns code for calling the model's *model*_step function (valid for ERT only).
LibCallModelTerminate	Returns code for calling the model's *model*_terminate function (valid for ERT only).
LibCallSetEventForThisBaseStep	Returns code for calling the model's set events function (valid for ERT only).
LibWriteModelData	Returns data for the model (valid for ERT only).

Code Template API Functions (Continued)

Function	Description
`LibSetRTModelErrorStatus`	Returns the code to set the model error status.
`LibGetRTModelErrorStatus`	Returns the code to get the model error status.
`LibIsSingleRateModel`	Returns true if model is single rate and false otherwise.
`LibGetModelName`	Returns name of the model (no extension).
`LibGetMdlSrcBaseName`	Returns the name of model's main source file (for example, *`model`*`.c` or `.cpp`).
`LibGetMdlPubHdrBaseName`	Returns the name of model's public header file (for example, *`model`*`.h`).
`LibGetMdlPrvHdrBaseName`	Returns the name of the model's private header file (for example, *`model`*`_private.h`).
`LibIsSingleTasking`	Returns `true` if the model is configured for single-tasking execution.
`LibWriteModelInput`	Returns the code to write to a particular root input (that is, a model inport block). (valid for ERT only).
`LibWriteModelOutput`	Returns the code to write to a particular root output (that is, a model outport block). (valid for ERT only).
`LibWriteModelInputs`	Returns the code to write to root inputs (that is, all model inport blocks). (valid for ERT only)
`LibWriteModelOutputs`	Returns the code to write to root outputs (that is, all model outport blocks). (valid for ERT only).
`LibNumDiscreteSampleTimes`	Returns the number of discrete sample times in the model.

Code Template API Functions (Continued)

Function	Description
`LibSetSourceFileCodeTemplate`	Set the code template to be used for generating a specified source file.
`LibSetSourceFileOutputDirectory`	Set the directory into which a specified source file is to be generated.
`LibAddSourceFileCustomSection`	Add a custom section to a source file. The custom section must be associated with one of the built-in (required) sections: `Includes`, `Defines`, `Types`, `Enums`, `Definitions`, `Declarations`, or `Functions`.
`LibSetSourceFileCustomSection`	Adds to the contents of a specified custom section within a specified file. The custom section must have been previously created with `LibAddSourceFileCustomSection`.
`LibGetSourceFileCustomSection`	Returns the contents of a specified custom section within a specified file.
`LibSetCodeTemplateComplianceLevel`	This function must be called from your CFP template before any other code template API functions are called. Pass in `2` as the `level` argument. **Note** Some MathWorks TLC files pass in 1 as the `level` argument. Currently, there is no difference in handling of level 1 versus level 2 by MathWorks software.

Generating Custom File and Function Banners

Using code generation template (CGT) files, you can specify custom file banners and function banners for the generated code files. File banners are comment sections in the header and trailer sections of a generated file.

Function banners are comment sections for each function in the generated code. Use these banners to add a company copyright statement, specify a special version symbol for your configuration management system, remove time stamps, and for many other purposes. These banners can contain characters, which propagate to the generated code.

To specify banners, create a custom CGT file with customized banner sections. The build process creates an executable TLC file from the CGT file. The code generation process then invokes the TLC file.

You do not need to be familiar with TLC programming to generate custom banners. You can modify example files that are supplied with the ERT target.

Note Prior releases supported direct use of customized TLC files as banner templates. You specified these with the **Source file (.c) banner template** and **Header file (.h) banner template** options of the ERT target. You can still use a custom TLC file banner templates, however, you can now use CGT files instead.

ERT template options on the **Code Generation > Templates** pane of a configuration set, in the **Code templates** section, support banner generation.

Code Generation: Templates Pane

The options for function and file banner generation are:

- "Code templates: Source file (*.c) template": CGT file to use when generating source (`.c` or `.cpp`) files. Place this file on the MATLAB path.
- "Code templates: Header file (*.h) template": CGT file to use when generating header (`.h`) files. You must place this file on the MATLAB path. This file can be the same template specified in the **Code templates: Source file (*.c) template** field, in which case identical banners are generated in source and header files.

  By default, the template for both source and header files is *`matlabroot`*`/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
- In each of these fields, click **Browse** to navigate to and select an existing CGT file for use as a template. Click **Edit** to open the specified file into the MATLAB editor, where you can customize it.

Creating a Custom File and Function Banner Template

To customize a CGT file for custom banner generation, make a local copy of the default code template and edit it, as follows:

1. Activate the configuration set you that want to work with.
2. Open the **Code Generation** pane of the active configuration set.
3. Click the **Templates** tab.
4. By default, the code template specified in the **Code templates: Source file (*.c) template** and **Code templates: Header file (*.h) template** fields is `matlabroot/toolbox/rtw/targets/ecoder/ert_code_template.cgt`.
5. If you want to use a different template as your starting point, click **Browse** to locate and select a CGT file.
6. Click **Edit** button to open the CGT file into the MATLAB editor.
7. Save a local copy of the CGT file. Store the copy in a directory that is outside of the MATLAB directory structure, but on the MATLAB path. If necessary, add the directory to the MATLAB path.
8. If you intend to use the CGT file with a custom target, locate the CGT file in a folder under your target root directory.
9. Rename your local copy of the CGT file. When you rename the CGT file, update the associated **Code templates: Source file (*.c) template** or **Code templates: Header file (*.h) template** field to match the new file name.
10. Edit and customize the local copy of the CGT file for banner generation, using the information provided in "Customizing a Code Generation Template (CGT) File for File and Function Banner Generation" on page 17-54.
11. Save your changes to the CGT file.
12. Click **Apply** to update the configuration set.
13. Save your model.

14 Generate code. Examine the generated source and header files to confirm that they contain the banners specified by the template or templates.

Customizing a Code Generation Template (CGT) File for File and Function Banner Generation

This section describes how to edit a CGT file for custom file and function banner generation. For a description of CGT files, see "Code Generation Template (CGT) Files" on page 17-27.

Components of the File and Function Banner Sections in the CGT file.

In a CGT file, you can modify the following sections: file banner, function banner, shared utility function banner, and file trailer. Each section is defined by open and close tags. The tags specific to each section are shown in the following table.

CGT File Section	Open Tag	Close Tag
"File Banner" on page 17-56	`<FileBanner>`	`</FileBanner>`
"Function Banner" on page 17-57	`<FunctionBanner>`	`</FunctionBanner>`
"Shared Utility Function Banner" on page 17-58	`<SharedUtilityBanner>`	`</SharedUtilityBanner>`
"File Trailer" on page 17-59	`<FileTrailer>`	`</FileTrailer>`

You can customize your banners by including tokens and comments between the open and close tag for each section. Tokens are typically TLC variables, for example `<ModelVersion>`, which are replaced with values in the generated code.

Note Including C comment indicators, '/*' or a '*/', in the contents of your banner might introduce an error in the generated code.

An open tag includes tag attributes. Enclose the value of the attribute in double quotes. The attributes available for an open tag are:

- `width`: specifies the width of the file or function banner comments in the generated code. The default value is 80.
- `style`: specifies the boundary for the file or function banner comments in the generated code. See Built-in Styles on page 17-55 for style options.

The open tag syntax is as follows:

```
<OpenTag style = “style_value” width = “num_width”>
```

The following table includes the built-in style options for the `style` attribute.

Built-in Styles

Style Value	Example

`classic`	`/* single line comments */`
	`/*`
	` * multiple line comments`
	` * second line`
	` */`

`classic_cpp`	`// single line comments`
	`//`
	`// multiple line comments`
	`// second line`
	`//`

`box`	`/************************************************************`
`box_cpp`	`///////////////////////////////////////////////////**********`
`open_box`	`/*************************************************************`
`open_box_cpp`	`/////////////////////////////////////////////////////////////`

```
// banner contents *******************************************
/////////////////////////////////////////////////////////////
```

File Banner. This section contains comments and tokens for use in generating a custom file banner. The file banner precedes any C or C++ code generated by the model. If you omit the file banner section from the CGT file, then no file banner emits to the generated code. The following section is the file banner section provided with the default CGT file, *matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file banner section (optional)
%%
<FileBanner style="classic">
File: %<FileName>

Code generated for Simulink model %<ModelName>.

Model version                     : %<ModelVersion>
Simulink Coder version            : %<RTWFileVersion>
TLC version                       : %<TLCVersion>
C/C++ source code generated on    : %<SourceGeneratedOn>
%<CodeGenSettings>
</FileBanner>
```

Summary of Tokens for File Banner Generation

FileName	Name of the generated file (for example, "rtwdemo_udt.c").
FileType	Either "source" or "header". Designates whether generated file is a .c or .cpp file or an .h file.
FileTag	Given file names file.c or .cpp and file.h; the file tags are "file_c" and "file_h", respectively.
ModelName	Name of generating model.
ModelVersion	Version number of model.
RTWFileVersion	Version number of *model*.rtw file.
RTWFileGeneratedOn	Timestamp of *model*.rtw file.

Summary of Tokens for File Banner Generation (Continued)

TLCVersion	Version of Target Language Compiler.
SourceGeneratedOn	Timestamp of generated file.
CodeGenSettings	Code generation settings for model: target language, target selection, embedded hardware selection, emulation hardware selection, code generation objectives (in priority order), and Code Generation Advisor validation result.

Function Banner. This section contains comments and tokens for use in generating a custom function banner. The function banner precedes any C or C++ function generated during the build process. If you omit the function banner section from the CGT file, the default function banner emits to the generated code. The following section is the default function banner section provided with the default CGT file, *matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom function banner section (optional)
%%   Customize function banners by using the following predefined tokens:
%% %<ModelName>, %<FunctionName>, %<FunctionDescription>, %<Arguments>,
%% %<ReturnType>, %<GeneratedFor>, %<BlockDescription>.
%%
<FunctionBanner style="classic">
%<FunctionDescription>
%<BlockDescription>
</FunctionBanner>
```

Summary of Tokens for Function Banner Generation

FunctionName	Name of function
Arguments	List of function arguments
ReturnType	Return type of function
ModelName	Name of generating model
FunctionDescription	Short abstract about the function

Summary of Tokens for Function Banner Generation (Continued)

GeneratedFor	Full block path for the generated function
BlockDescription	User input from the **Block Description** parameter of the block properties dialog box. `BlockDescription` contains an optional token attribute, `style`. The only valid value for`style` is `content_only`, which is case-sensitive and enclosed in double quotes. Use the `content_only` style when you want to include only the block description content that you entered in the block parameter dialog. The syntax for the token attribute `style` is: %<BlockDescription style = content_only >

Shared Utility Function Banner. The shared utility function banner section contains comments and tokens for use in generating a custom shared utility function banner. The shared utility function banner precedes any C or C++ shared utility function generated during the build process. If you omit the shared utility function banner section from the CGT file, the default shared utility function banner emits to the generated code. The following section is the default shared utility function banner section provided with the default CGT file, *matlabroot*/toolbox/rtw/targets/ecoder/ert_code_template.cgt.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom shared utility function banner section (optional)
%%   Customize banners for functions generated in shared location by using the
%% following predefined tokens: %<FunctionName>, %<FunctionDescription>,
%% %<Arguments>, %<ReturnType>.
%%
<SharedUtilityBanner style="classic">
%<FunctionDescription>
</SharedUtilityBanner>
```

Summary of Tokens for Shared Utility Function Banner Generation

`FunctionName`	Name of function
`Arguments`	List of function arguments
`ReturnType`	Return type of function
`FunctionDescription`	Short abstract about function

File Trailer. The file trailer section contains comments for generating a custom file trailer. The file trailer follows any C or C++ code generated from the model. If you omit the file trailer section from the CGT file, no file trailer emits to the generated code. The following section is the default file trailer provided in the default CGT file.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Custom file trailer section (optional)
%%
<FileTrailer style="classic">
File trailer for generated code.

[EOF]
</FileTrailer>
```

All of the tokens available for the file banner are available for the file trailer. See Summary of Tokens for File Banner Generation on page 17-56.

Template Symbols and Rules

Introduction

"Template Symbol Groups" on page 17-60 and "Template Symbols" on page 17-62 describe MPF template symbols and rules for using them. The location of a symbol in one of the supplied 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 that symbol are located in the corresponding generated file. "Template Symbol Groups" on page 17-60 identifies the symbol groups, starting with the parent ("Base") group, followed by the children of each parent. "Template Symbols" on page 17-62 lists the symbols alphabetically.

Template Symbol Groups

Symbol Group	Symbol Names in This Group
Base (Parents)	Declarations Defines Definitions Documentation Enums Functions Includes Types
Declarations	ExternalCalibrationLookup1D ExternalCalibrationLookup2D ExternalCalibrationScalar ExternalVariableScalar
Defines	LocalDefines LocalMacros
Definitions	FilescopeCalibrationLookup1D FilescopeCalibrationLookup2D FilescopeCalibrationScalar FilescopeVariableScalar GlobalCalibrationLookup1D GlobalCalibrationLookup2D GlobalCalibrationScalar GlobalVariableScalar

Symbol Group	Symbol Names in This Group

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

	`Notes` `ToolVersion`
Functions	`CFunctionCode`
Types	This parent has no children.

Template Symbols

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

Symbol Name*	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.
`Enums`	Base	File	Enumerated data type definitions.
`ExternalCalibrationLookup1D`	Declarations	External	***
`ExternalCalibrationLookup2D`	Declarations	External	***
`ExternalCalibrationScalar`	Declarations	External	***
`ExternalVariableScalar`	Declarations	External	***
`FileName`	Documentation	N/A	Name of the generated file.
`FilescopeCalibrationLookup1D`	Definitions	File	***
`FilescopeCalibrationLookup2D`	Definitions	File	***
`FilescopeCalibrationScalar`	Definitions	File	***
`FilescopeVariableScalar`	Definitions	File	***
`Functions`	Base	File	Generated function code.
`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.**

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`Includes`	Base	File	`#include` preprocessor directives.
`LastModifiedDate`	Documentation	N/A	Date when model was last saved. From **Last saved on** field on Model Properties dialog box.
`LastModifiedBy`	Documentation	N/A	User who last saved model. From **Last saved by** field on Model Properties dialog box.
`LocalDefines`	Defines	File	`#define` preprocessor directives from code-generation data dictionary.
`LocalMacros`	Defines	File	C/C++ macros local to the file.
`ModelName`	Documentation	N/A	Name of the model.
`ModelVersion`	Documentation	N/A	Version number of the Simulink model.
`ModifiedBy`	Documentation	N/A	Name of user who last modified the model. From **Model version** field on Model Properties dialog box.
`ModifiedComment`	Documentation	N/A	Comment user enters in the **Modified Comment** field on the Log Change dialog box. See "Creating a Model Change History" in the Simulink documentation.
`ModifiedDate`	Documentation	N/A	Date model was last modified before code was generated.

Symbol Name*	Symbol Group	Symbol Scope	Symbol Description (What the symbol puts in the generated file)
`ModifiedHistory`	Documentation	N/A	Text from **Modified history** field on Model Properties dialog box.**
`Notes`	Documentation	N/A	User-supplied miscellaneous notes about the model or generated files. Placed in the generated file based on the Stateflow note, Simulink annotation, or DocBlock on the model.**
`ToolVersion`	Documentation	N/A	A list of the versions of the toolboxes used in generating the code.
`Types`	Base		Data types of generated code.

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

** This symbol can be used to add a comment to the generated files. See “Adding Global Comments” on page 17-5. The code generator places the comment in each generated file whose template has this symbol name. The code generator places the comment at the location that corresponds to where the symbol name is located in the template file.

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

Rules for Modifying or Creating a Template

The following are the rules for creating any MPF template. “Comparison of a Template and Its Generated File” on page 17-44 illustrates several of these rules.

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

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

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

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

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

6. Place a comment on the template between /* */ as in standard ANSI C[4]. This results in /*comment*/ on the generated file.

7. Each MPF template must have all of the Base group symbols, in predefined order. They are listed in “Template Symbol Groups” on page 17-60. 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.

---

4. ANSI® is a registered trademark of the American National Standards Institute, Inc.

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

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

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

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

Configuring the Placement of Data in Generated Code

To...	Select or Enter...
Specify whether data is to be defined in the generated source file or in a single separate header file	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **Data definition** parameter.
Specify whether data is to be declared in the generated source file or in a single separate header file	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **Data declaration** parameter.
Specify the `#include` file delimiter to be used in generated files that contain the `#include` preprocessor directive for `mpt` data objects	Select **Auto**, **Data defined in source file**, or **Data defined in single separate source file** for the **#include file delimiter** parameter.
Name the generated module using the same name as the model or a user-specified name	Select **Not specified**, **Same as model**, or **User specified** for the **Module naming** parameter.
Control whether signal data objects are to be declared as global data in the generated code	Enter an integer value for the **Signal display level** parameter.
Declare a parameter data object as tunable global data in the generated code	Enter an integer value for the **Parameter tune level** parameter.

For details about data placement, see Chapter 13, “Managing Placement of Data Definitions and Declarations”.

Ensuring Delimiter Is Specified for All #Includes

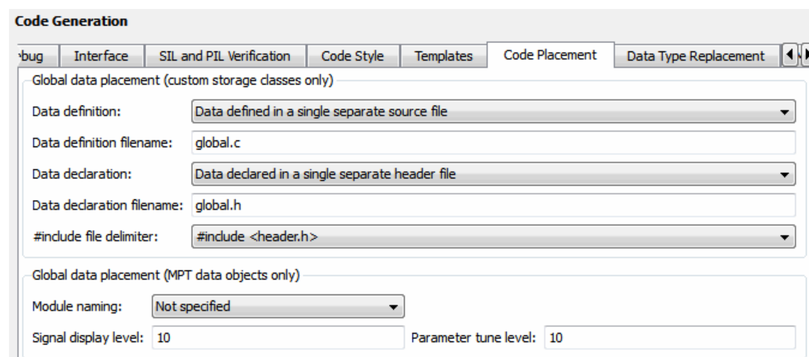
Understanding the purpose of this procedure requires understanding the Header file property of a data object, described in Parameter and Signal Property Values on page 7-2, and applied in “Creating mpt Data Objects with Data Object Wizard” on page 12-11. For a particular data object, you can specify as the Header file property value a .h filename where that data object will be declared. Then, in the IncludeFile section of the generated file, this .h file is indicated in a #include preprocessor directive.

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

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

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

2 Click **Apply**.



18

Defining Model Configuration Variations

- "Introduction" on page 18-2
- "Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer" on page 18-3

Introduction

This chapter explains how to use the Embedded Real-Time (ERT) target code generation options to configure models for production code generation. The discussion also includes other options that are not specific to the ERT target, but which affect ERT code generation.

Every model contains one or more named configuration sets that specify model parameters such as solver options, code generation options, and other choices. A model can contain multiple configuration sets, but only one configuration set is active at any time. A configuration set includes options that affect code generation in general, and options that are specific to a given target, such as the ERT target.

Configuration sets can be particularly useful in embedded systems development. By defining multiple configuration sets in a model, you can easily retarget code generation from that model. For example, one configuration set might specify the default ERT target with external mode support enabled for rapid prototyping, while another configuration set might specify the ERT-based target for Visual C++ to generate production code for deployment of the application. Activation of either configuration set fully reconfigures the model for the appropriate type of code generation.

Before you work with the ERT target options, you should become familiar with

- Configuration sets and how to view and edit them in the Configuration Parameters dialog box. The *Simulink User's Guide* document contains detailed information on these topics.
- Code generation options and the use of the System Target File Browser. The "Configuration Parameters for Simulink Models" and "Preparing Models for Code Generation" contains detailed information on these topics in the Simulink Coder documentation.

For descriptions of the Embedded Real-Time (ERT) target code generation options, see "Configuration Parameters".

Viewing ERT Target Options in the Configuration Parameters Dialog Box or Model Explorer

The Configuration Parameters dialog box and Model Explorer provide the quickest routes to a model's active configuration set. Illustrations throughout this chapter and the "Configuration Parameters" reference in the show the Configuration Parameters dialog box view of model parameters (unless otherwise noted).

Generating Code and Building Executables

- Chapter 19, "Generating Code Modules"
- Chapter 20, "Generating Reports for Code Reviews and Traceability Analysis"
- Chapter 21, "Optimizing Generated Code"
- Chapter 22, "Developing Models and Code That Comply with Industry Standards and Guidelines"
- Chapter 23, "Generating Reentrant Code from MATLAB Code"

19

Generating Code Modules

Code Modules

In this section...

"Introduction" on page 19-2
"Generated Code Modules" on page 19-2
"User-Written Code Modules" on page 19-5
"Customizing Generated Code Modules" on page 19-5

Introduction

This section summarizes the code modules and header files that make up a Embedded Coder program and describes where to find the code modules and header files.

The easiest way to locate and examine the generated code files is to use the HTML code generation report. The code generation report provides a table of hyperlinks that you click to view the generated code in the MATLAB Help browser. For more information, see "Generating an HTML Code Generation Report" on page 20-4.

Generated Code Modules

The Embedded Coder software creates a build folder in your working folder to store generated source code. The build folder also contains object files, a makefile, and other files created during the code generation process. The default name of the build folder is `model_ert_rtw`.

The Embedded Coder™ File Packaging on page 19-3 section summarizes the structure of source code generated by the Embedded Coder software.

Note The Embedded Coder file packaging differs slightly (but significantly) from the file packaging employed by the GRT, GRT malloc, and other nonembedded targets. For more information, see the Simulink Coder documentation.

Embedded Coder File Packaging

File	Description
model.c or .cpp	Contains entry points for code implementing the model algorithm (for example, *model*_step, *model*_initialize, and *model*_terminate).
*model*_private.h	Contains local macros and local data that are required by the model and subsystems. This file is included in the model.c file as a #include statement. You do not need to include *model*_private.h when interfacing handwritten code to the generated code of a model.
model.h	Declares model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (*model*_M) with accessor macros. *model*.h is included in the subsystem .c or .cpp files of the model. If you are interfacing your handwritten code to generated code for one or more models, include *model*.h for each of those models.
*model*_data.c or .cpp (conditional)	*model*_data.c or .cpp is conditionally generated. It contains the declarations for the parameters data structure, the constant block I/O data structure, and any zero representations for the model structure data types. If these data structures and zero representations are not used in the model, *model*_data.c or .cpp is not generated. These structures and zero representations are declared extern in *model*.h.
*model*_types.h	Provides forward declarations for the real-time model data structure and the parameters data structure. Function declarations of reusable functions might need these declarations. Also provides type definitions for user-defined types used by the model.
rtwtypes.h	Defines data types, structures, and macros required by Embedded Coder generated code. Most other generated code modules also require these definitions.
ert_main.c or .cpp (optional)	If the **Generate an example main program** option is on, this file is generated. (This option is on by default.) See “Generate an example main program”.

Embedded Coder File Packaging (Continued)

File	Description
autobuild.h (optional)	If the **Generate an example main program** option is off, this file is generated. (See "Generate an example main program".) autobuild.h contains #include directives required by the static version of the ert_main.c main program module. Because the static ert_main.c is not created at code generation time, ert_main.c includes autobuild.h to access model-specific data structures and entry points. For more information, see "Static Main Program Module" on page 34-14.
*model*_capi.c or .cpp *model*_capi.h (optional)	Provides data structures that enable a running program to access model signals, states, and parameters without external mode. To learn how to generate and use the *model*_capi.c or .cpp and .h files, see "Interacting with Target Application Data Using the C API" in the Simulink Coder documentation.

You can customize the generated set of files in several ways:

- File packaging formats: Specify the number of source files generated for your model. In the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, specify the **File packaging format** parameter. For more information, see "Customizing Generated Code Modules" on page 19-5.

- Nonvirtual subsystem code generation: Instruct the code generation software to generate separate functions, within separate code files, for any nonvirtual subsystems. You can control the names of the functions and of the code files. For further information, see "Creating Subsystems" in the Simulink Coder documentation.
- Custom storage classes: Use custom storage classes to partition generated data structures into different files based on file names that you specify. For further information, see Chapter 8, "Creating and Using Custom Storage Classes".

- Module Packaging Features (MPF): Direct the generated code into a required set of `.c` or `.cpp` and `.h` files, and control the internal organization of the generated files. For details, see Defining Data Representation and Storage for Code Generation on page 1.

User-Written Code Modules

Code that you write to interface with generated model code usually includes a customized main module (based on a main program provided by the code generation software), and may also include interrupt handlers, device driver blocks and other S-functions, and other supervisory or supporting code.

Establish a working folder for your own code modules. Put your working folder on the MATLAB path. Minimally, you must also modify the ERT template makefile and system target file so that the build process can find your source and object files. If you want to generate code for a particular microprocessor or development board and deploy the code on target hardware with a cross-development system, make more extensive modifications to the ERT target files.

For information on how to customize the ERT target for your production requirements, see “Customizing Targets” in the Simulink Coder documentation.

Customizing Generated Code Modules

Embedded Coder software provides a configuration parameter to specify how the generated source code is packaged into files. The configuration parameter “File packaging format” drop-down list options are located in the Configuration Parameter dialog box, on the **Code Generation > Code Placement** pane, in the Code Packaging section. The options are: `Modular`, `Compact (with separate data file)`, and `Compact`. Generated Files According to File Packaging Format on page 19-6 shows the files generated for each file packaging format and the files that have been removed.

Generated Files According to File Packaging Format

File Packaging Format	Generated Files	Removed Files
Modular (default)	*model*.c subsystem files (optional) *model*.h *model*_types.h *model*_private.h *model*_data.c (conditional)	None
Compact (with separate data file)	*model*.c *model*.h *model*_data.c (conditional)	*model*_private.h *model*_types.h
Compact	*model*.c *model*.h	*model*_data.c *model*_private.h *model*_types.h

The code generation process places the content of the removed files as follows:

Removed File	Generated Content In File
*model*_private.h	*model*.c and *model*.h
*model*_types.h	*model*.h
*model*_data.c	*model*.c

You can specify a different file packaging format for each referenced model.

If you specify **Utility code generation** as Shared location on the **Code Generation > Interface** pane of the Configuration Parameter dialog box, the code generation process generates separate files for utility code in a shared

location, regardless of the file packaging format. If you specify the **Utility code generation** as `Auto`, the generated code for utilities is dependent on the file packaging format as follows:

- `Modular`: Some shared utility files are in the build directory
- `Compact (with separate data file)`: Utility code is generated in *`model`*`.c`
- `Compact`: Utility code is generated in *`model`*`.c`

File packaging formats, `Compact` or `Compact (with separate data file)` generate *`model`*`_types.h` for models containing:

- A Model Variants block or a Variant Subsystem block. The *`model`*`_types.h` file includes preprocessor directives defining the variant objects associated with a variant block.
- Custom storage classes specifying a separate header file. The *`model`*`_types.h` file includes the `#include` call to the external header file.

File packaging formats, `Compact` or `Compact (with separate data file)` are not compatible with the following:

- A model containing a subsystem, which is configured to generate separate source files
- A model containing a noninlined S-function

20

Generating Reports for Code Reviews and Traceability Analysis

- "About HTML Code Generation Report Extensions" on page 20-2
- "Generating an HTML Code Generation Report" on page 20-4
- "Using the Code Interface Report to Analyze the Generated Code Interface" on page 20-6

About HTML Code Generation Report Extensions

The Embedded Coder code generation report is an enhanced version of the HTML code generation report normally generated by the Simulink Coder build process. In the report:

- The **Summary** section lists version, date, and code generation objectives information. The **Configuration settings at the time of code generation** link opens a noneditable view of the Configuration Parameters dialog box that shows the Simulink model settings, including TLC options, at the time of code generation.
- The **Subsystem Report** section contains information on nonvirtual subsystems in the model.
- The **Code Interface Report** section provides information about the generated code interface, including model entry point functions and input/output data. For more information, see “Using the Code Interface Report to Analyze the Generated Code Interface” on page 20-6.
- The **Traceability Report** section allows you to account for **Eliminated / Virtual Blocks** that are untraceable, versus the listed **Traceable Simulink Blocks / Stateflow Objects / MATLAB Scripts**, providing a complete mapping between model elements and code. For more information, see “Customizing Traceability Reports” on page 37-8.

In the **Generated Files** section of the **Contents** pane, you can click the names of source code files generated from your model to view their contents in a MATLAB Web browser window. In the displayed source code:

- The summary information is included as the code header.
- Global variable instances are hyperlinked to their definitions.
- If you selected the traceability option **Code-to-model**, hyperlinks within the displayed source code let you view the blocks or subsystems from which the code was generated. Click the hyperlinks to view the relevant blocks or subsystems in the Simulink model window. For more information about tracing code to blocks, see “Tracing Code to Model Objects Using Hyperlinks” on page 37-2.
- If you selected the traceability option **Model-to-code**, you can view the generated code for any block in the model. To highlight the generated code

for a block in the HTML report, right-click the block and select **Code Generation > Navigate to Code**. For more information about tracking blocks to generated code, see "Tracing Model Objects to Generated Code" on page 37-4.

- If you set the **Code coverage tool** parameter on the **Code Generation > SIL and PIL Verification** pane, you can view the code coverage data and annotations in the generated code in the HTML Code Generation Report. For more information about the code coverage tool for SIL and PIL verification, see "Using a Code Coverage Tool in a SIL Simulation" on page 39-25.

For a complete discussion on using traceability to verify generated code, see "Traceability for Production Code Generation" on page 37-2.

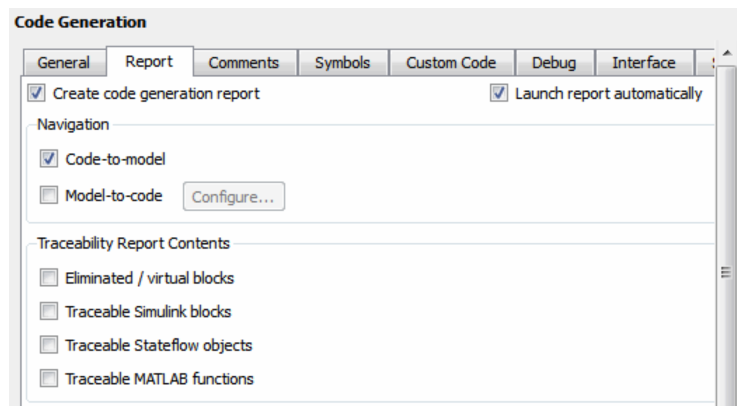
Generating an HTML Code Generation Report

To generate an HTML code generation report,

1. With your ERT-based model open, open the Configuration Parameters dialog box or Model Explorer and navigate to the **Code Generation > Report** pane.

2. Select **Create code generation report** if it is not already selected. By default, **Launch report automatically** and **Code-to-model** also are selected, and **Model-to-code** is cleared, as shown in the figure below.

   You can select or clear any of these options.



3. Generate code from your model or subsystem (for example, for a model, by clicking **Build** on the **Code Generation** pane of the Configuration Parameters dialog box).

4. The build process writes the code generation report files in the `html` subdirectory of the build directory. The top-level HTML report file is named *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`.

5. If you selected **Launch report automatically**, the build process automatically opens a MATLAB Web browser window and displays the code generation report.

If you did not select **Launch report automatically**, you can open the code generation report (`*model*_codegen_rpt.html` or `*subsystem*_codegen_rpt.html`) manually into a MATLAB Web browser window, or into another Web browser.

6. If you selected **Code-to-model**, hyperlinks to blocks in the generating model are created in the report files. When you view the report files in a MATLAB Web browser, clicking on these hyperlinks displays and highlights the referenced blocks in the model. For more information, see "Tracing Code to Model Objects Using Hyperlinks" on page 37-2.

7. If you selected **Model-to-code**, model-to-code highlighting support is included in the generated HTML report. To highlight the generated code for a block in your Simulink model, right-click the block and select **Code Generation > Navigate to Code**. This selection highlights the generated code for the block in the HTML code generation report. For more information, see "Tracing Model Objects to Generated Code" on page 37-4 and "Customizing Traceability Reports" on page 37-8.

Notes

- For large models (containing over 1000 blocks), you may find that HTML report generation takes longer than you want. In this case, consider clearing the **Code-to-model** and **Model-to-code** check boxes. The report will be generated faster.
- You can also view the HTML report files, as well as the generated code files, in the Simulink Model Explorer. See "Viewing Generated Code in the Model Explorer Code Viewer" in the Simulink Coder documentation for details.

Using the Code Interface Report to Analyze the Generated Code Interface

In this section...

"Code Interface Report Overview" on page 20-6

"Generating a Code Interface Report" on page 20-7

"Navigating Code Interface Report Subsections" on page 20-9

"Interpreting the Entry Point Functions Subsection" on page 20-10

"Interpreting the Inports and Outports Subsections" on page 20-13

"Interpreting the Interface Parameters Subsection" on page 20-14

"Interpreting the Data Stores Subsection" on page 20-16

"Code Interface Report Limitations" on page 20-17

Code Interface Report Overview

When you select the **Create code generation report** option for an ERT-based model, a **Code Interface Report** section is automatically included in the generated HTML report. The **Code Interface Report** section provides documentation of the generated code interface, including model entry point functions and interface data, for consumers of the generated code. The information in the report can help facilitate code review and code integration.

The code interface report includes the following subsections

- **Entry Point Functions** — interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`
- **Inports** and **Outports** — interface information about each model inport and outport
- **Interface Parameters** — interface information about tunable parameters that are associated with the model
- **Data Stores** — interface information about global data stores and data stores with non-`auto` storage that are associated with the model

For limitations that apply to code interface reports, see "Code Interface Report Limitations" on page 20-17

Note This section uses the following demo models for illustration purposes:

- `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window) for examples of report subsections
- `rtwdemo_mrmtbb` for examples of timing information
- `rtwdemo_fcnprotoctrl` for examples of function argument and return value information

---

Generating a Code Interface Report

To generate a code interface report for your model, perform the following steps.

1. Open your model, go to the **Code Generation** pane of the Configuration Parameters dialog box, and select `ert.tlc` or an ERT-based **System target file**, if one is not already selected.

2. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**, if it is not already selected. The `rtwdemo_basicsc`, `rtwdemo_mrmtbb`, and `rtwdemo_fcnprotoctrl` demo models used in this section select every **Report** pane option by default, but selecting **Create code generation report** alone is sufficient to generate a **Code Interface Report** section in the HTML report.

   Alternatively, you can programmatically select the option by issuing the following MATLAB command:

   ```
   set_param(bdroot, 'GenerateReport', 'on')
   ```

   If the **Report** pane option **Code-to-model** is selected, the generated report will contain hyperlinks to the model. You should leave this value selected unless you plan to use the report outside the MATLAB environment.

3 Build the model. If you selected the **Report** pane option **Launch report automatically**, the code generation report opens automatically after the build process completes. (Otherwise, you can launch it manually from within the model build directory.)

4 To display the code interface report for your model, go to the **Contents** pane of the HTML report and click the **Code Interface Report** link. For example, here is the generated code interface report for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Entry Point Functions

Function: rtwdemo_basicsc_initialize

Prototype	**void rtwdemo_basicsc_initialize(void)**
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: rtwdemo_basicsc_step

Prototype	**void rtwdemo_basicsc_step(void)**
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interface Parameters

[+] ...

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

For help navigating the content of the code interface report subsections, see "Navigating Code Interface Report Subsections" on page 20-9. For help interpreting the content of the code interface report subsections, see the sections beginning with "Interpreting the Entry Point Functions Subsection" on page 20-10.

Navigating Code Interface Report Subsections

To help you navigate code interface descriptions, the code interface report provides collapse/expand tokens and hyperlinks, as follows:

- For any lengthy subsection, the report provides `[-]` and `[+]` symbols that allow you to collapse or expand that section. In the example in the previous section, the symbols are provided for the **Inports** and **Interface Parameters** sections.
- Several forms of hyperlink navigation are provided in the code interface report. For example,
  - The **Table of Contents** located at the top of the code interface report provides links to each subsection.
  - You can click on each function name to go to its declaration in *`model`*`.c`.
  - You can click on each function's header file name to go to the header file source listing.
  - If you selected the **Report** pane option **Code-to-model** for your model, you can click hyperlinks for any of the following to go to the corresponding location in the model display:
    - Function argument
    - Function return value
    - Inport
    - Outport
    - Interface parameter (if the parameter source is a block)
    - Data store (if the data store source is a Data Store Memory block)

For general backward and forward navigation within the HTML code generation report, use the **Back** and **Forward** buttons above the **Contents** section in the upper left corner of the report.

Interpreting the Entry Point Functions Subsection

The **Entry Point Functions** subsection of the code interface report provides the following interface information about each model entry point function, including `model_initialize`, `model_step`, and (if applicable) `model_terminate`.

Field	Description
Function:	Lists the function name. You can click on the function name to go to its declaration in *`model`*`.c`.
Prototype	Displays the function prototype, including the function return value, name, and arguments.
Description	Provides a text description of the function's purpose in the application.
Timing	Describes the timing characteristics of the function, such as how many times the function is called, or if it is called periodically, at what time interval. For a multirate timing example, see the `rtwdemo_mrmtbb` report excerpt below.
Arguments	If the function has arguments, displays the number, name, data type, and Simulink description for each argument. If you selected the **Report** pane option **Code-to-model** for your model, you can click the hyperlink in the description to go to the block corresponding to the argument in the model display. For argument examples, see the `rtwdemo_fcnprotoctrl` report excerpt below.

Field	Description
Return value	If the function has a return value, displays the return value data type and Simulink description. If you selected the **Report** pane option **Code-to-model** for your model, you can click the hyperlink in the description to go to the block corresponding to the return value in the model display. For a return value example, see the `rtwdemo_fcnprotoctrl` report excerpt below.
Header file	Lists the name of the header file for the function. You can click on the header file name to go to the header file source listing.

For example, here is the **Entry Point Functions** subsection for the demo model `rtwdemo_basicsc`.

Entry Point Functions

Function: rtwdemo_basicsc_initialize

Prototype	**void rtwdemo_basicsc_initialize(void)**
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

Function: rtwdemo_basicsc_step

Prototype	**void rtwdemo_basicsc_step(void)**
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_basicsc.h

To illustrate how timing information might be listed for a multirate model, here are the **Entry Point Functions** and **Inports** subsections for the demo model `rtwdemo_mrmtbb`. This multirate, discrete-time, multitasking model contains Inport blocks 1 and 2, which specify 1-second and 2-second sample times, respectively. The sample times are constrained to the specified times

by the **Periodic sample time constraint** option on the **Solver** pane of the Configuration Parameters dialog box.

Entry Point Functions

Function: rtwdemo_mrmtbb_initialize

Prototype	**void rtwdemo_mrmtbb_initialize(void)**
Description	Initialization entry point of generated code
Timing	Called once
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step0

Prototype	**void rtwdemo_mrmtbb_step0(void)**
Description	Output entry point of generated code
Timing	Called periodically, every 1 second
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

Function: rtwdemo_mrmtbb_step1

Prototype	**void rtwdemo_mrmtbb_step1(void)**
Description	Output entry point of generated code
Timing	Called periodically, every 2 seconds
Arguments	None
Return value	None
Header file	rtwdemo_mrmtbb.h

To illustrate how function arguments and return values are displayed in the report, here is the **Entry Point Functions** description of the model step function for the demo model `rtwdemo_fcnprotoctrl`.

Function: rtwdemo_fcnprotoctrl_step_custom

Prototype	**boolean_T rtwdemo_fcnprotoctrl_step_custom (const real_T argIn1, const BusObject *const argIn2, BusObject *argOut2, const BusObject *const argIn3, uint8_T *argIn4)**
Description	Output entry point of generated code
Timing	Called periodically, every 0.2 seconds
Arguments	[-]
Return value	
Header file	rtwdemo_fcnprotoctrl.h

Arguments:

#	Name	Data Type	Description
1	argIn1	const real_T	<Root>/In1
2	argIn2	const BusObject *const	<Root>/In2
3	argOut2	BusObject *	<Root>/Out2
4	argIn3	const BusObject *const	<Root>/In3
5	argIn4	uint8_T *	<Root>/In4

Return value:

Data Type	Description
boolean_T	<Root>/Out1

Interpreting the Inports and Outports Subsections

The **Inports** and **Outports** subsections of the code interface report provide the following interface information about each inport and outport in the model.

Field	Description
Block Name	Displays the Simulink block name of the inport or outport. If you selected the **Report** pane option **Code-to-model** for your model, you can click on each inport or outport **Block Name** value to go to its location in the model display.
Code Identifier	Lists the identifier associated with the inport or outport data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier string. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label '`Imported data:`'.

Field	Description
	• If the data is neither defined nor declared in the generated code — for example, if the option **Generate reusable code** is selected for the model — the field displays the string '`Defined externally`'.
Data Type	Lists the data type of the inport or outport.
Dimension	Lists the dimensions of the inport or outport (for example, `1` or `[4, 5]`).

For example, here are the **Inports** and **Outports** subsections for the demo model `rtwdemo_basicsc`.

Inports

[-]

Block Name	Code Identifier	Data Type	Dimension
<Root>/In1	input1	real32_T	1
<Root>/In2	input2	real32_T	1
<Root>/In3	input3	real32_T	1
<Root>/In4	input4	real32_T	1

Outports

Block Name	Code Identifier	Data Type	Dimension
<Root>/Out1	output	real32_T	1

Interpreting the Interface Parameters Subsection

The **Interface Parameters** subsection of the code interface report provides the following interface information about tunable parameters that are associated with the model.

Field	Description
Parameter Source	Lists the source of the parameter value, as follows: • If the source of the parameter value is a block, the field displays the block name, such as `<Root>/Gain2` or `<S1>/Lookup1`. If you selected the **Report** pane option **Code-to-model** for your model, you can click on the **Parameter Source** value to go to the parameter's location in the model display. • If the source of the parameter value is a workspace variable, the field displays the name of the workspace variable prefixed with the label '`Workspace variable:`'; for example, `Workspace variable: K2`.
Code Identifier	Lists the identifier associated with the tunable parameter data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier string. • If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if the option **Generate reusable code** is selected for the model — the field displays the string '`Defined externally`'.
Data Type	Lists the data type of the tunable parameter.
Dimension	Lists the dimensions of the tunable parameter (for example, `1` or `[4, 5, 6]`).

For example, here is the **Interface Parameters** subsection for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Interface Parameters

[-]

Parameter Source	Code Identifier	Data Type	Dimension
Workspace variable: K2	K2	real_T	1
Workspace variable: LOWER	LOWER	real32_T	1
Workspace variable: T1Break	T1Break	real32_T	[1 11]
Workspace variable: T1Data	T1Data	real32_T	[1 11]
Workspace variable: T2Break	T2Break	real32_T	[1 3]
Workspace variable: T2Data	T2Data	real32_T	[3 3]
Workspace variable: UPPER	UPPER	real32_T	1
Workspace variable: K1	K1	int8_T	1

Interpreting the Data Stores Subsection

The **Data Stores** subsection of the code interface report provides the following interface information about global data stores and data stores with non-`auto` storage that are associated with the model.

Field	Description
Data Store Source	Lists the source of the data store memory, as follows: • If the data store is defined using a Data Store Memory block, the field displays the block name, such as `<Root>/DS1`. If you selected the **Report** pane option **Code-to-model** for your model, you can click on the **Data Store Source** value to go to the data store's location in the model display. • If the data store is defined using a `Simulink.Signal` object, the field displays the name of the `Simulink.Signal` object prefixed with the label '`Global:`'.
Code Identifier	Lists the identifier associated with the data store data in the generated code, as follows: • If the data is defined in the generated code, the field displays the identifier string.

Field	Description
	• If the data is declared but not defined in the generated code — for example, if the data is resolved with an imported storage class — the field displays the identifier string prefixed with the label '`Imported data:`'. • If the data is neither defined nor declared in the generated code — for example, if the option **Generate reusable code** is selected for the model — the field displays the string '`Defined externally`'.
Data Type	Lists the data type of the data store.
Dimension	Lists the dimensions of the data store (for example, `1` or `[1, 2]`).

For example, here is the **Data Stores** subsection for the demo model `rtwdemo_basicsc` (with the **ExportedGlobal Storage Class** button selected in the demo model window).

Data Stores

Data Store Source	Code Identifier	Data Type	Dimension
<Root>/Data Store Memory	mode	boolean_T	1

Code Interface Report Limitations

The following limitations apply to code interface section of the HTML code generation reports.

- The code interface report does not support the GRT interface with an ERT target or the **C++ (Encapsulated)** language option. For these configurations, the code interface report will not be generated and will not appear in the HTML code generation report **Contents** pane.
- The code interface report supports data resolved with most custom storage classes (CSCs), except when the CSC properties are set in any of the following ways:

  - The CSC property **Type** is set to `FlatStructure`. For example, the `BitField` and `Struct` CSCs in the Simulink package have **Type** set to `FlatStructure` and are not supported by the code interface report.
  - The CSC property **Type** is set to `Other`. For example, the `GetSet` CSC in the Simulink package has **Type** set to `Other` and is not supported by the code interface report.
  - The CSC property **Data access** is set to `Pointer`, indicating that imported symbols are declared as pointer variables rather than simple variables. This property is accessible only when the CSC property **Data scope** is set to `Imported` or `Instance-specific`.

  In these cases, the report displays empty **Data Type** and **Dimension** fields.

- For outports, the code interface report cannot describe the associated memory (data type and dimensions) if the memory is optimized. In these cases, the report displays empty **Data Type** and **Dimension** fields.
- The code interface report does not support data type replacement using the **Code Generation > Data Type Replacement** pane of the Configuration Parameters dialog box. The data types listed in the report will link to built-in data types rather than their specified replacement data types.

21

Optimizing Generated Code

- "Configuring Production Code Optimizations" on page 21-2
- "Optimization Dependencies" on page 21-5
- "Optimizing Your Model with Configuration Wizard Blocks and Scripts" on page 21-7
- "Tips for Optimizing the Generated Code" on page 21-19

Configuring Production Code Optimizations

Several parameters available on the **Optimization** panes configure your model to optimize product code generation. The following table includes optimization parameters on the **Optimization > General** pane:

To...	Select or Specify...
Generate initialization code for root-level inports and outports with a value of zero	Select **Remove root level I/O zero initialization**.
Generate additional code to set float and double storage explicitly to value `0.0`	Select **Use memset to initialize floats and doubles to 0.0** When you set this parameter, the `memset` function clears internal storage (regardless of type) to the integer bit pattern `0` (that is, all bits are off). The additional code generated when the option is off, is slightly less efficient.If the representation of floating-point zero used by your compiler and target CPU is identical to the integer bit pattern `0`, you can gain efficiency by setting this parameter.
Suppress the generation of code that initializes internal work structures (for example, block states and block outputs) to zero	Select **Remove internal state zero initialization**.
Generate run-time initialization code for a block that has states only if the block is in a system that can reset its states, such as an enabled subsystem	Select **Optimize initialization code for model reference** This results in more efficient code. The following restrictions apply to using the **Optimize initialization code for model reference** parameter. However, these restrictions do not apply to a Model block that references a function-call model. • In a subsystem that resets states, do not include a Model block that references a model that has this parameter set to `on`. For example, in an enabled subsystem with the **States when enabling** block parameter set to `reset`, do not include a Model block that references a model that has

To...	Select or Specify...
	the **Optimize initialization code for model reference** parameter set to `on`. • If you set the **Optimize initialization code for model reference** parameter to `off` in a model that includes a Model block that directly references a submodel, do not set the **Optimize initialization code for model reference** parameter for the submodel to `on`.
Remove code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur	Select **Remove code from floating-point to integer conversions that wraps out-of-range values**. This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.
Suppress generation of code that guards against fixed-point division by zero	Select **Remove code that protects against division arithmetic exceptions**. When you select this parameter, simulation results and results from generated code may no longer be in bit-for-bit agreement.
To minimize the amount of memory allocated for absolute and elapsed time counters	Specify an integer value for **Application lifespan (days)** For more information on the allocation and operation of absolute and elapsed timers, see “Using Timers”, “Using Timers in Asynchronous Tasks”, and “Controlling Memory Allocation for Time Counters” in the Simulink Coder documentation.

The following table includes optimization parameters on the **Optimization > Signals and Parameters** pane:

To...	Select or Specify...
Control whether parameter data for reusable subsystems is generated in a separate header file for each subsystem or in a single parameter data structure	Select `Hierarchical` or `NonHierarchical` for **Parameter structure**.
Replace multiply operations in array indices when accessing arrays in a loop	Select **Simplify array indexing**.
Store Boolean signals as one-bit bitfields instead of as a Boolean data type	Select **Pack Boolean data into bitfields**. Selecting this parameter enables the **Bitfield declarator type specifier**. To optimize your code further, select `uchar_T`, however this optimization benefit is dependent on your choice of target.
Pass each reusable subsystem output argument as an address of a local to reduce global memory usage and eliminate copying local variables back to global block I/O structures	Select `Individual arguments` for **Pass reusable subsystem outputs as**.

Optimization Dependencies

Several parameters available on the **Optimization** panes have dependencies on settings of other options. The following table summarizes the dependencies on the **Optimization > General** pane.

Option	Dependencies?	Dependency Details
Block reduction	No	
Conditional input branch execution	No	
Implement logic signals as Boolean data (versus double)	Yes	Disable for models created with a Simulink version that supports only signals of type `double`
Application lifespan (days)	No	
Remove root level I/O zero initialization (ERT targets only)	No	
Use memset to initialize floats and doubles to 0.0	No	
Remove internal data zero initialization (ERT targets only)	No	
Optimize initialization code for model reference (ERT targets only)	Yes	Disable if model includes an enabled subsystem *and* the model is referred to from another model with a Model block
Remove code from floating-point to integer conversions that wrap out-of-range values	No	
Remove code from floating-point to integer conversions with saturation that maps NaN to zero	Yes (ERT targets) No (GRT targets)	For ERT targets, enabled by **Support floating-point numbers** and **Support non-finite numbers** in the **Code Generation > Interface** pane
Remove code that protects against division arithmetic exceptions (ERT targets only)	No	

The following table summarizes the dependencies on the **Optimization > Signals and Parameters** pane.

Option	Dependencies?	Dependency Details
Signal storage reuse	No	
Inline parameters	Yes	Disable for referenced models in a model reference hierarchy
Parameter structure (ERT targets only)	Yes	Enabled by **Inline parameters**
Enable local block outputs	Yes	Enabled by **Signal storage reuse**
Reuse block outputs	Yes	Enabled by **Signal storage reuse**
Inline invariant signals	Yes	Enabled by **Inline parameters**
Eliminate superfluous local variables (Expression folding)	Yes	Enabled by **Signal storage reuse**
Pack Boolean data into bitfields (ERT targets only)	No	
Bitfield declarator type specifier (ERT targets only)	Yes	Enabled by **Pack Boolean data into bitfields**
Minimize data copies between local and global variables	Yes	Enabled by **Signal storage reuse**
Simplify array indexing (ERT targets only)	No	
Loop unrolling threshold	No	
Maximum stack size (bytes)	No	
Use memcpy for vector assignment	No	
Memcpy threshold (bytes)	Yes	Enabled by **Use memcpy for vector assignment**
Pass reusable subsystem output as (ERT targets only)	No	

Optimizing Your Model with Configuration Wizard Blocks and Scripts

In this section...

"Overview" on page 21-7
"Adding a Configuration Wizard Block to Your Model" on page 21-9
"Using Configuration Wizard Blocks" on page 21-11
"Creating a Custom Configuration Wizard Block" on page 21-11

Overview

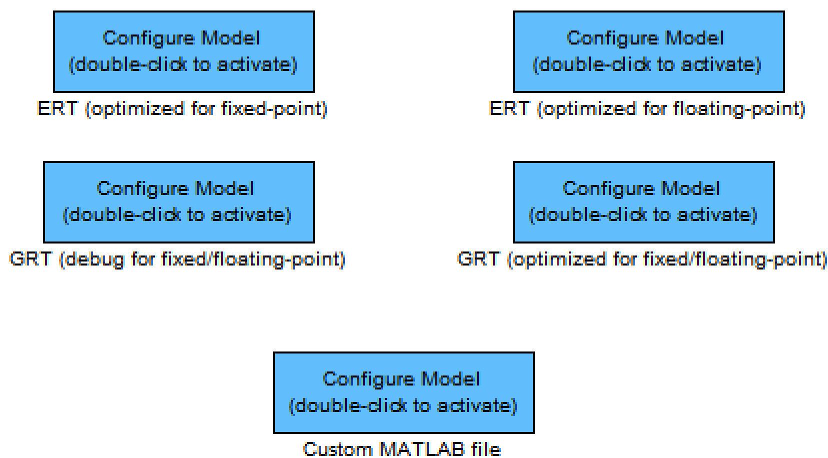
The Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

The library provides a Configuration Wizard block you can customize, and four preset Configuration Wizard blocks.

Block	Description
Custom MATLAB file	Automatically update active configuration parameters of parent model using a custom file
ERT (optimized for fixed-point)	Automatically update active configuration parameters of parent model for ERT fixed-point code generation
ERT (optimized for floating-point)	Automatically update active configuration parameters of parent model for ERT floating-point code generation

Block	Description
GRT (debug for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation with debugging enabled
GRT (optimized for fixed/floating-point)	Automatically update active configuration parameters of parent model for GRT fixed- or floating-point code generation

These are shown in the figure below.



When you add one of the preset Configuration Wizard blocks to your model and double-click it, a predefined MATLAB file script executes and configures all parameters of the model's active configuration set without manual intervention. The preset blocks configure the options optimally for one of the following cases:

- Fixed-point code generation with the ERT target
- Floating-point code generation with the ERT target

- Fixed/floating-point code generation with TLC debugging options enabled, with the GRT target.
- Fixed/floating-point code generation with the GRT target

The Custom block is associated with an example MATLAB file script that you can adapt to your requirements.

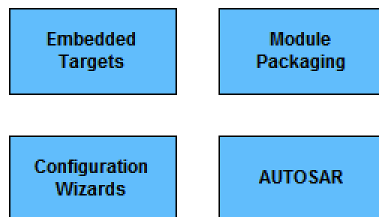
You can also set up the Configuration Wizard blocks to invoke the build process after configuring the model.

Adding a Configuration Wizard Block to Your Model

This section describes how to add one of the preset Configuration Wizard blocks to a model.

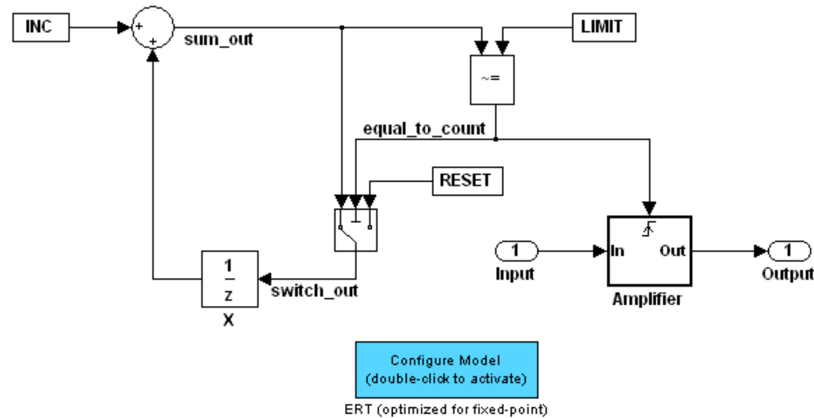
The Configuration Wizard blocks are available in the Embedded Coder block library. To use a Configuration Wizard block:

1. Open the model that you want to configure.
2. Open the Embedded Coder library by typing the command `rtweclib`.
3. The top level of the library is shown below.



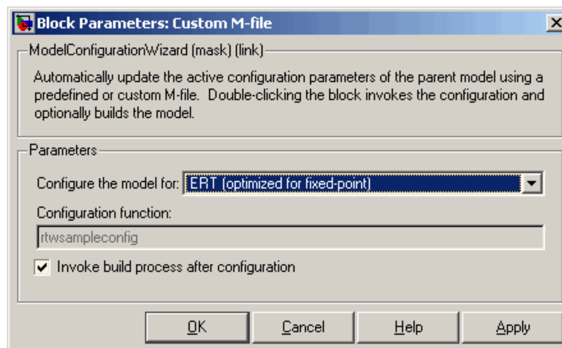
4. Double-click the Configuration Wizards icon. The Configuration Wizards sublibrary opens.

5 Select the Configuration Wizard block you want to use and drag and drop it into your model. In the figure below, the `ERT (optimized for fixed-point)` Configuration Wizard block has been added to the model.



6 You can set up the Configuration Wizard block to invoke the build process after executing its configuration script. If you do not want to use this feature, skip to the next step.

If you want the Configuration Wizard block to invoke the build process, right-click on the Configuration Wizard block in your model, and select **Mask Parameters...** from the context menu. Then, select the **Invoke build process after configuration** option, as shown below.



7 Click **Apply**, and close the Mask Parameters dialog box.

Note You should not change the **Configure the model for** option, unless you want to create a custom block and script. In that case, see "Creating a Custom Configuration Wizard Block" on page 21-11.

---

8 Save the model.

9 You can now use the Configuration Wizard block to configure the model, as described in the next section.

Using Configuration Wizard Blocks

Once you have added a Configuration Wizard block to your model, just double-click the block. The script associated with the block automatically sets all parameters of the active configuration set that are relevant to code generation (including selection of the appropriate target). You can verify that the options have changed by opening the Configuration Parameters dialog box and examining the settings.

If the **Invoke build process after configuration** option for the block was selected, the script also initiates the code generation and build process.

Note You can add more than one Configuration Wizard block to your model. This provides a quick way to switch between configurations.

Creating a Custom Configuration Wizard Block

The Custom Configuration Wizard block is shipped with an associated MATLAB file script, `rtwsampleconfig.m`. The script is located in the directory *`matlabroot`*`/toolbox/rtw/rtw`.

Both the block and the script are intended to provide a starting point for customization. This section describes:

- How to create a custom Configuration Wizard block linked to a custom script.

- Operation of the example script, and programming conventions and requirements for a customized script.
- How to run a configuration script from the MATLAB command line (without a block).

Setting Up a Configuration Wizard Block

This section describes how to set up a custom Configuration Wizard block and link it to a script. If you want to use the block in more than one mode, it is advisable to create a Simulink library to contain the block.

To begin, make a copy of the example script for later customization:

1. Create a directory to store your custom script. This directory should not be anywhere inside the MATLAB directory structure (that is, it should not be under `matlabroot`).

   The discussion below refers to this directory as `/my_wizards`.

2. Add the directory to the MATLAB path. Save the path for future sessions.

3. Copy the example script (`matlabroot/toolbox/rtw/rtw/rtwsampleconfig.m`) to the `/my_wizards` directory you created in the previous steps. Then, rename the script as desired. The discussion below uses the name `my_configscript.m`.

4. Open the example script into the MATLAB editor. Scroll to the end of the file and enter the following line of code:

   ```
   disp('Custom Configuration Wizard Script completed.');
   ```

   This statement is used later as a test to verify that your custom block has executed the script.

5. Save your script and close the MATLAB editor.

The next step is to create a Simulink library and add a custom block to it. Do this as follows:

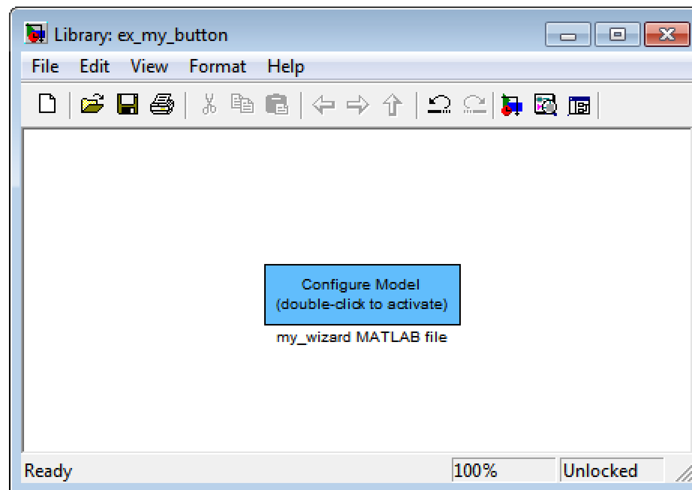
1 Open the Embedded Coder library and the Configuration Wizards sublibrary, as described in "Adding a Configuration Wizard Block to Your Model" on page 21-9.

2 Select **New > Library** from the **File** menu of the Configuration Wizards sublibrary window. An empty library window opens.

3 Select the Custom MATLAB file block from the Configuration Wizards sublibrary and drag and drop it into the empty library window.

4 To distinguish your custom block from the original, edit the Custom MATLAB file label under the block as desired.

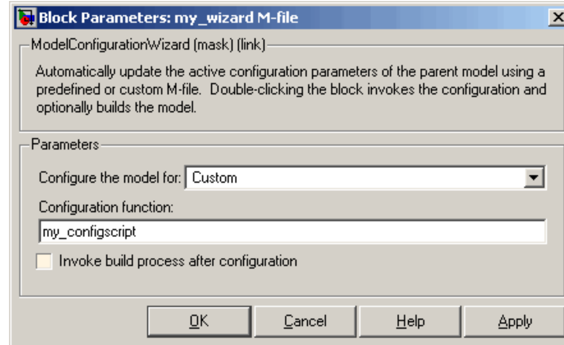
5 Select **Save as** from the **File** menu of the new library window; save the library to the `/my_wizards` directory, under your library name of choice. In the figure below, the library has been saved as `ex_custom_button`, and the block has been labeled `my_wizard MATLAB-file`.



The next step is to link the custom block to the custom script:

1 Right-click on the block in your model, and select **Mask Parameters** from the context menu. Notice that the **Configure the model for** menu set to `Custom`. When `Custom` is selected, the **Configuration function** edit field is enabled, so you can enter the name of a custom script.

2 Enter the name of your custom script into the **Configuration function** field. (Do not enter the `.m` filename extension, which is implicit.) In the figure below, the script name `my_configscript` has been entered into the **Configuration function** field. This establishes the linkage between the block and script.



3 Note that by default, the **Invoke build process after configuration** option is deselected. You can change the default for your custom block by selecting this option. For now, leave this option deselected.

4 Click **Apply** and close the Mask Parameters dialog box.

5 Save the library.

6 Close the Embedded Coder library and the Configuration Wizards sublibrary. Leave your custom library open for use in the next step.

Now, test your block and script in a model. Do this as follows:

1. Open the `vdp` demo model by typing the command:

   ```
   vdp
   ```

2. Open the Configuration Parameters dialog box and view the options by clicking on the **Code Generation** in the list in the left pane of the dialog box.

3. Observe that the `vdp` demo is configured, by default, for the GRT target. Close the Configuration Parameters dialog box.

4. Select your custom block from your custom library. Drag and drop the block into the `vdp` model.

5. In the `vdp` model, double-click your custom block.

6. In the MATLAB window, you should see the test message you previously added to your script:

   ```
   Custom Configuration Wizard Script completed.
   ```

   This indicates that the custom block successfully executed the script.

7. Reopen the Configuration Parameters dialog box and view the **Code Generation** pane again. You should now see that the model is configured for the ERT target.

Before applying further edits to your custom script, proceed to the next section to learn about the operation and conventions of Configuration Wizard scripts.

Creating a Configuration Wizard Script

You should create your custom Configuration Wizard script by copying and modifying the example script, rtwsampleconfig.m. This section provides guidelines for modification.

The Configuration Function. The example script implements a single function without a return value. The function takes a single argument cs:

```
function rtwsampleconfig(cs)
```

The argument cs is a handle to a proprietary object that contains information about the model's active configuration set. The Simulink software obtains this handle and passes it in to the configuration function when the user double-clicks a Configuration Wizard block.

Your custom script should conform to this prototype. Your code should use cs as a "black box" object that transmits information to and from the active configuration set, using the accessor functions described below.

Accessing Configuration Set Options. To set options or obtain option values, use the Simulink set_param and get_param functions (if you are unfamiliar with these functions, see the Simulink Reference document).

Option names are passed in to set_param and get_param as strings specifying an *internal option name*. The internal option name is not always the same as the corresponding option label on the GUI (for example, the Configuration Parameters dialog box). The example configuration accompanies each set_param and get_param call with a comment that correlates internal option names to GUI option labels. For example:

```
set_param(cs,'LifeSpan','1'); % Application lifespan (days)
```

To obtain the current setting of an option in the active configuration set, call get_param. Pass in the cs object as the first argument, followed by the internal option name. For example, the following code excerpt tests the setting of the **Create code generation report** option:

```
if strcmp(get_param(cs, 'GenerateReport'), 'on')
    ...
```

To set an option in the active configuration set, call `set_param`. Pass in the `cs` object as the first argument, followed by one or more parameter/value pairs that specify the internal option name and its value. For example, the following code excerpt turns off the **Support absolute time** option:

```
set_param(cs,'SupportAbsoluteTime','off');
```

Selecting a Target. A Configuration Wizard script must select a target configuration. The example script uses the ERT target as a default. The script first stores string variables that correspond to the required **System target file**, **Template makefile**, and **Make command** settings:

```
stf = 'ert.tlc';
tmf = 'ert_default_tmf';
mc  = 'make_rtw';
```

The system target file is selected by passing the `cs` object and the `stf` string to the `switchTarget` function:

```
switchTarget(cs,stf,[]);
```

The template makefile and make command options are set by `set_param` calls:

```
set_param(cs,'TemplateMakefile',tmf);
set_param(cs,'MakeCommand',mc);
```

To select a target, your custom script needs only to set up the string variables `stf`, `tmf`, and `mc` and pass them to the appropriate calls, as above.

Obtaining Target and Configuration Set Information. The following utility functions and properties are provided so that your code can obtain information about the current target and configuration set, with the `cs` object:

- `isValidParam(cs, 'option')`: The `option` argument is an internal option name. `isValidParam` returns true if `option` is a valid option in the context of the active configuration set.
- `getPropEnabled(cs, 'option')`: The `option` argument is an internal option name. Returns true if this `option` is enabled (that is, writable).
- `IsERTTarget` property: Your code can detect whether or not the currently selected target is derived from the ERT target is selected by checking the `IsERTTarget` property, as follows:

```
isERT = strcmp(get_param(cs,'IsERTTarget'),'on');
```

This information can be used to determine whether or not the script should configure ERT-specific options, for example:

```
if isERT
  set_param(cs,'ZeroExternalMemoryAtStartup','off');
  set_param(cs,'ZeroInternalMemoryAtStartup','off');
  set_param(cs,'InitFltsAndDblsToZero','off');
  set_param(cs,'InlinedParameterPlacement',...
               'NonHierarchical');
     set_param(cs,'NoFixptDivByZeroProtection','on')
end
```

Invoking a Configuration Wizard Script from the MATLAB Command Prompt

Like any other MATLAB file, Configuration Wizard scripts can be run from the MATLAB command prompt. (The Configuration Wizard blocks are provided as a graphical convenience, but are not essential.)

Before invoking the script, you must open a model and instantiate a `cs` object to pass in as an argument to the script. After running the script, you can invoke the build process with the `rtwbuild` command. The following example opens, configures, and builds a model.

```
open my_model;
cs = getActiveConfigSet ('my_model');
rtwsampleconfig(cs);
rtwbuild('my_model');
```

Tips for Optimizing the Generated Code

In this section...

"Introduction" on page 21-19

"Using Configuration Wizard Blocks" on page 21-19

"Setting Hardware Implementation Parameters Correctly" on page 21-20

"Removing Unnecessary Initialization Code" on page 21-22

"Generating Pure Integer Code If Possible" on page 21-23

"Disabling MAT-File Logging" on page 21-23

"Using Virtualized Output Ports Optimization" on page 21-24

"Controlling Signal Storage" on page 21-25

"Using External Mode with the ERT Target" on page 21-26

"Optimizing Generated Code Using Specified Minimum and Maximum Values" on page 21-27

Introduction

The Embedded Coder software features a number of code generation options that can help you further optimize the generated code. This section highlights code generation options you can use to improve performance and reduce code size.

Most of the tips in this section apply specifically to the ERT target. See also the "Optimizing Generated Code" section of the Simulink Coder documentation for optimization techniques that are common to all target configurations.

Using Configuration Wizard Blocks

The Embedded Coder software provides a library of *Configuration Wizard* blocks and scripts to help you configure and optimize code generation from your models quickly and easily.

When you add one of the preset Configuration Wizard blocks to your model and double-click it, a MATLAB file script executes and configures all

parameters of the model's active configuration set without user intervention. The preset blocks configure the options optimally for common fixed- and floating-point code generation scenarios.

You can also create custom Configuration Wizard scripts and blocks.

See "Optimizing Your Model with Configuration Wizard Blocks and Scripts" on page 21-7 for detailed information.

Setting Hardware Implementation Parameters Correctly

Correct specification of target-specific characteristics of generated code (such as word sizes for `char`, `short`, `int`, and `long` data types, or desired rounding behaviors in integer operations) can be critical in embedded systems development. The **Hardware Implementation** category of options in a configuration set provides a simple and flexible way to control such characteristics in both simulation and code generation.

Before generating and deploying code, you should become familiar with the options on the **Hardware Implementation** pane of the Configuration Parameters dialog box. See "Hardware Implementation Pane" in the Simulink documentation and "Configuring the Hardware Implementation" in the Simulink Coder documentation for full details on the **Hardware Implementation** pane.

By configuring the **Hardware Implementation** properties of your model's active configuration set to match the behaviors of your compiler and hardware, you can generate more efficient code. For example, if you specify the **Byte ordering** property, you can avoid generation of extra code that tests the byte ordering of the target CPU.

You can use the `rtwdemo_targetsettings` demo model to determine some implementation-dependent characteristics of your C or C++ compiler, as well as characteristics of your target hardware. By using this model in conjunction with your target development system and debugger, you can observe the behavior of the code as it executes on the target. You can then use this information to configure the **Hardware Implementation** parameters of your model.

To use this model, type the command

```
rtwdemo_targetsettings
```

Follow the instructions in the model window.

Removing Unnecessary Initialization Code

Consider selecting the **Remove internal state zero initialization** and **Remove root level I/O zero initialization** options on the **Optimziation > General** pane.

These options (both off by default) control whether internal data (block states and block outputs) and external data (root inports and outports whose value is zero) are initialized. Initializing the internal and external data whose value is zero is a precaution and may not be necessary for your application. Many embedded application environments initialize all RAM to zero at startup, making generation of initialization code redundant.

However, be aware that if you select **Remove internal state zero initialization**, it is not guaranteed that memory is in a known state each time the generated code begins execution. If you turn the option on, running a model (or a generated S-function) multiple times can result in different answers for each run.

This behavior is sometimes desirable. For example, you can turn on **Remove internal state zero initialization** if you want to test the behavior of your design during a warm boot (that is, a restart without full system reinitialization).

In cases where you have turned on **Remove internal state zero initialization** but still want to get the same answer on every run from a S-function generated by the Embedded Coder software, you can use either of the following MATLAB commands before each run:

```
clear SFcnName
```

where *SFcnName* is the name of the S-function, or

```
clear mex
```

A related option, **Use memset to initialize floats and doubles**, lets you control the representation of zero used during initialization. See "Use memset to initialize floats and doubles to 0.0" in the Simulink reference documentation.

Note that the code still initializes data structures whose value is not zero when **Remove internal state zero initialization** and **Remove root level I/O zero initialization** are selected.

Note also that data of `ImportedExtern` or `ImportedExternPointer` storage classes is never initialized, regardless of the settings of these options.

Generating Pure Integer Code If Possible

If your application uses only integer arithmetic, deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane to ensure that generated code contains no floating-point data or operations. When this option is deselected, an error is raised if any noninteger data or expressions are encountered during code generation. The error message reports the offending blocks and parameters.

Disabling MAT-File Logging

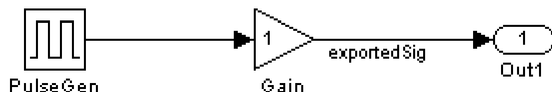
Clear the **MAT-file logging** option in the **Verification** section of the **Interface** pane. This setting is the default, and is recommended for embedded applications because it eliminates the extra code and memory usage required to initialize, update, and clean up logging variables. In addition to these efficiencies, clearing the **MAT-file logging** option lets you exploit further efficiencies under certain conditions. See "Using Virtualized Output Ports Optimization" on page 21-24 for information.

Note also that code generated to support MAT-file logging invokes `malloc`, which may be undesirable for your application.

Using Virtualized Output Ports Optimization

The *virtualized output ports* optimization lets you store the signal entering the root output port as a global variable. This eliminates code and data storage associated with root output ports when the **MAT-file logging** option is cleared and the TLC variable `FullRootOutputVector` equals `0`, both of which are defaults for Embedded Coder targets.

To illustrate this feature, consider the model shown in the following block diagram. Assume that the signal `exportedSig` has `exportedGlobal` storage class.



In the default case, the output of the Gain block is written to the signal storage location, `exportedSig`. No code or data is generated for the `Out1` block, which has become, in effect, a virtual block. This is shown in the following code fragment.

```
/* Gain Block: <Root>/Gain */
  exportedSig = rtb_PulseGen * VirtOutPortLogOFF_P.Gain_Gain;
```

In cases where either the **MAT-file logging** option is enabled, or `FullRootOutputVector = 1`, the generated code represents root output ports as members of an external outputs vector.

The following code fragment was generated from the same model shown in the previous example, but with **MAT-file logging** enabled. The output port is represented as a member of the external outputs vector `VirtOutPortLogON_Y`. The Gain block output value is copied to both `exportedSig` and to the external outputs vector.

```
/* Gain Block: <Root>/Gain */
  exportedSig = rtb_PulseGen * VirtOutPortLogON_P.Gain_Gain;

/* Outport Block: <Root>/Out1 */
  VirtOutPortLogON_Y.Out1 = exportedSig;
```

The overhead incurred by maintenance of data in the external outputs vector can be significant for smaller models being used to perform benchmarks.

Note that you can force root output ports to be stored in the external outputs vector (regardless of the setting of **MAT-file logging**) by setting the TLC variable `FullRootOutputVector` to `1`. You can do this by adding the statement

```
%assign FullRootOutputVector = 1
```

to the Embedded Coder system target file. Alternatively, you can enter the assignment with **TLC options** on the **Code Generation** pane of the Configuration Parameters dialog box.

For more information on how to control signal storage in generated code, see the "Defining Data Representation and Storage for Code Generation" section of the Simulink Coder documentation.

Controlling Signal Storage

There are a number of options that let you control how signals in your model are stored and represented in the generated code. You can control whether signal storage is declared in global memory space, or locally in functions (that is, in stack variables).

For a complete discussion of signal storage options, see the "Defining Data Representation and Storage for Code Generation" section of the Simulink Coder documentation.

If you want to store signals in stack space, you must turn the **Enable local block outputs** option on. To do this

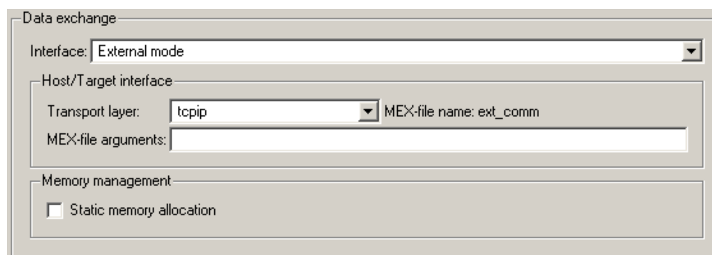
1. Select the **Optimization > Signals and Parameters** of the Configuration Parameters dialog box. Make sure that **Signal storage reuse** is selected. If **Signal storage reuse** is cleared, **Enable local block outputs** is not available.

2. Select the **Enable local block outputs** option. Click **Apply** if necessary.

Using External Mode with the ERT Target

Selecting the **External mode** option turns on generation of code to support external mode communication between host (Simulink) and target systems. The Embedded Coder software supports all features of Simulink external mode, as described in the "Communicating With Code Executing on a Target System Using Simulink External Mode" section of the Simulink Coder documentation.

This section discusses external mode options that may be of special interest to embedded systems designers. The next figure shows the **Data Exchange** subpane of the Configuration Parameters dialog box, **Interface** pane, with `External mode` selected.



Memory Management

Consider the **Memory management** option **Static memory allocation** before generating external mode code for an embedded target. Static memory allocation is generally desirable, as it reduces overhead and promotes deterministic performance.

When you select the **Static memory allocation** option, static external mode communication buffers are allocated in the target application. When **Static memory allocation** is deselected, communication buffers are allocated dynamically (with `malloc`) at run time.

Generation of Pure Integer Code with External Mode

The Embedded Coder software supports generation of pure integer code when external mode code is generated. To do this, select the **External mode** option, and deselect the **Support floating-point numbers** option in the **Software environment** section of the **Interface** pane.

This enhancement lets you generate external mode code that is free of any storage definitions of double or float data type, and allows your code to run on integer-only processors

If you intend to generate pure integer code with **External mode** on, note the following requirements:

- All trigger signals must be of data type `int32`. Use a Data Type Conversion block if needed.
- When pure integer code is generated, the simulation stop time specified in the **Solver** options is ignored. To specify a stop time, run your target application from the MATLAB command line and use the `-tf` option. (See "Running the External Program" in the Simulink Coder documentation.) If you do not specify this option, the application executes indefinitely (as if the stop time were `inf`).

  When executing pure integer target applications, the stop time specified by the `-tf` command line option is interpreted as the number of base rate ticks to execute, rather than as an elapsed time in seconds. The number of ticks is computed as

  ```
  stop time in seconds / base rate step size in seconds
  ```

Optimizing Generated Code Using Specified Minimum and Maximum Values

To optimize the generated code for your model, you can choose an option to use input range information, also known as *design minimum and maximum*, that you specify on signals and parameters. These minimum and maximum values usually represent environmental limits, such as temperature, or mechanical and electrical limits, such as output ranges of sensors.

When you select the **Optimize using specified minimum and maximum values** configuration parameter, the software uses the minimum and

maximum values to derive range information for downstream signals in the model. It then uses this derived range information to determine if it is possible to streamline the generated code by, for example:

- Reducing expressions to constants
- Removing dead branches of conditional statements
- Eliminating unnecessary mathematical operations

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

How to Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks
  - Block outputs
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
  - `Simulink.Signal` objects
- Ensure that the minimum and maximum values for signals and parameters are accurate and trustworthy. Otherwise, optimization might result in numerical mismatch with simulation.

  Before generating code, test the values by simulating your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

How to Enable Simulation Range Checking

1. In your model, select **Simulation > Configuration Parameters** to open the Configuration Parameters dialog box.

2. In the Configuration Parameters dialog box, select **Diagnostics > Data Validity**.
3. On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to `warning` or `error`.

- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream. For more information, see "Example: Optimizing Generated Code Using Specified Minimum and Maximum Values" on page 21-29.

How to Enable Optimization

1. Set the **Code Generation > System target file** configuration parameter to select an Embedded Real-Time (`ERT`) target (requires a Embedded Coder license).

2. Specify design minimum and maximum values for signals and parameters in your model using the tips in "How to Configure Your Model" on page 21-28.

3. Select the **Optimization > General Optimize using specified minimum and maximum values** configuration parameter.

   For more information, see "Optimize using the specified minimum and maximum values" in the Simulink documentation.

Example: Optimizing Generated Code Using Specified Minimum and Maximum Values

This example demonstrates how the software uses specified input range information to determine whether it can eliminate unnecessary utility functions from the generated code. It uses the `rtwdemo_minmax` demo model.

Generate Code Without Using Specified Minimum and Maximum Values. First, generate code without taking into account the minimum and maximum values for the inputs to the Sum and Gain blocks or the minimum and maximum values for the Gain block parameter to see the code generated without the optimization.

1 Open the model. At the MATLAB command line, enter:

```
rtwdemo_minmax
```

2 Double-click the **View Optimization Configuration** button.

The **Optimization** pane of the Configuration Parameters dialog box appears.

On the **Code generation** panel, note that the **Optimize using specified minimum and maximum values** parameter is cleared.

3 Double-click the **Generate Code** button.

The code generation report appears.

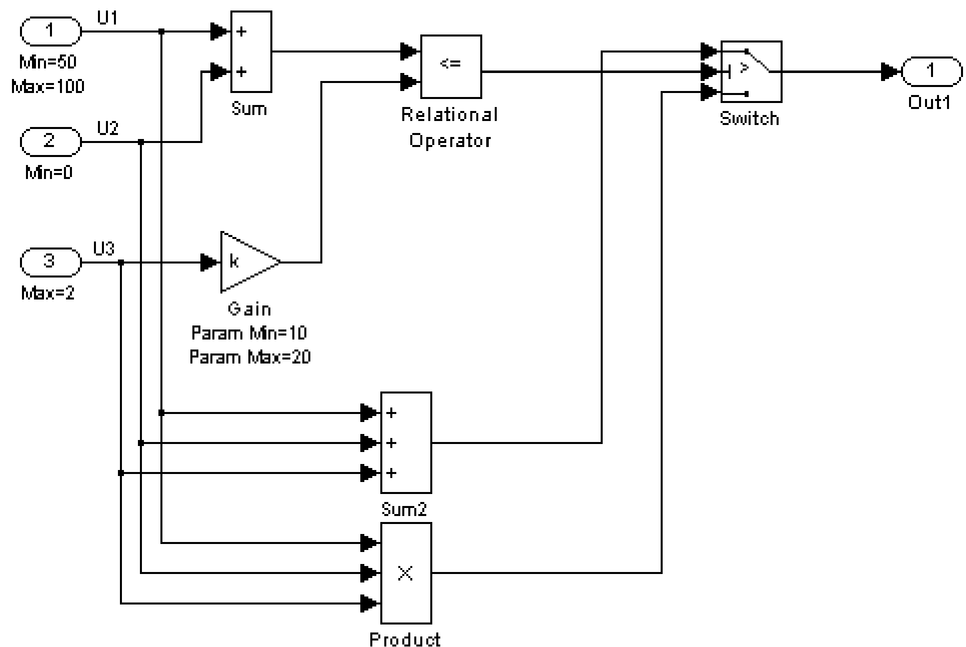
4 In the left pane of the report, click the `rtwdemo_minmax.c` link.

The report displays the C code in the right pane.

The generated code for this model includes a branch for each of the Relational Operator block inputs.

```
void rtwdemo_minmax_step(void)
{
  if (U1 + U2 <= k * U3) {
       rtY.Out1 = (U1 + U2) + U3;
  } else {
    rtY.Out1 = U1 * U2 * U3;
  }
 }
```

Generate Code Using Minimum and Maximum Values. Next, enable the optimization and generate code for the model again, this time taking into account the design minimum and maximum values for the inputs to the Sum and Gain blocks and the minimum and maximum values for the Gain block parameter.



Note that:

- The minimum value of the first input to the Relational Operator block is 50 because this value is the minimum output from the Sum block.
- The maximum value of the second input to the Relational Operator block is 40 because this value is the maximum output of the Gain block.

Therefore, the output of the Relational Operator block is always false, and so the output of the Switch block is always the product of the three inputs.

1. Double-click the **View Optimization Configuration** button.

   The Optimization pane of the Configuration Parameters dialog box appears.

2. On the **Code generation** panel, select the **Optimize using specified minimum and maximum values** parameter and click **Apply**.
3. Double-click the **Generate Code** button.

The code generation report appears.

4 In the left pane of the report, click the `rtwdemo_minmax.c` link and inspect the generated code. Using the minimum and maximum values, the software optimized the generated code by eliminating the conditional statement.

```
void rtwdemo_minmax_step(void)
{
  rtY.Out1 = U1 * U2 * U3;
}
```

Limitations

- This optimization does not take into account minimum and maximum values for:
  - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.
- If you use Polyspace® software to verify code generated using this optimization, it might mark code that was previously green as orange. For example, if your model contains a division where the range of the denominator does not include zero, the generated code does not include protection against division by zero. Polyspace might mark this code orange because it does not have information about the minimum and maximum values for the inputs to the division.

  The Polyspace Model Link products do automatically capture some minimum and maximum values specified in the MATLAB workspace, for example, for `Simulink.Signal` and `Simulink.Parameter` objects. In this example, to provide range information to the Polyspace software, use a `Simulink.Signal` object on the input of the division and specify a range that does not include zero.

The Polyspace Model Link products store these values in a Data Range Specification (DRS) file. However, they do not capture all minimum and maximum values in your Simulink model. To provide additional minimum and maximum information to Polyspace, you can manually define a DRS file. For more information, see the Polyspace Model Link documentation.

- If you are using double-precision data types and the **Code Generation > Interface > Support non-finite numbers** configuration parameter is selected, this optimization does not occur.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, code is generated once for the subsystem and shares this code among the multiple instances of the subsystem.
- The Model Advisor **Check safety-related optimization settings** check generates a warning if this option is selected. For many safety-critical applications, removing dead code automatically is unacceptable because doing so might make code untraceable. For more information, see Check safety-related optimization settings.
- If a block uses a `Simulink.Parameter` object to define the value of a parameter, this optimization takes into account the minimum and maximum values specified for the `Simulink.Parameter` object only if it is used on its own. It does not use these minimum and maximum values if the object is part of an expression. For example, if a Gain block has a gain parameter specified as `K1`, where `K1` is defined as a `Simulink.Parameter` object in the base workspace, the optimization takes the minimum and maximum values of `K1` into account. However, if the Gain block has a gain parameter of `K1+5` or `K1+K2+K3`, where `K2` and `K3` are also `Simulink.Parameter` objects, the optimization does not use the minimum and maximum values of `K1`, `K2` or `K3`.

Developing Models and Code That Comply with Industry Standards and Guidelines

- "What Are the Standards and Guidelines?" on page 22-2
- "Developing Models and Code That Comply with MAAB Guidelines" on page 22-4
- "Developing Models and Code That Comply with MISRA C Guidelines" on page 22-5
- "Developing Models and Code That Comply with the IEC 61508 Standard" on page 22-6
- "Developing Models and Code That Comply with the ISO 26262 Standard" on page 22-8
- "Developing Models and Code That Comply with the DO-178B Standard" on page 22-10

What Are the Standards and Guidelines?

If your application has mission-critical development and certification goals, your models or subsystems and the code generated for them might need to comply with one or more of the standards and guidelines listed in the following table.

Standard or Guidelines	Organization	For More Information, See...
Guidelines: Use of MATLAB, Simulink, and Stateflow software for control algorithm modeling – MathWorks Automotive Advisory Board (MAAB) Guidelines	MAAB	• Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow Software: PDF, Word • "Developing Models and Code That Comply with MAAB Guidelines" on page 22-4
Guidelines: Use of the C Language in Critical Systems (MISRA C[5])	Motor Industry Software Reliability Association (MISRA)	• MISRA C Web site • Technical Solution 1-1IFP0W on the MathWorks Web site • "Developing Models and Code That Comply with MISRA C Guidelines" on page 22-5
Standard: AUTomotive Open System ARchitecture (AUTOSAR)	AUTOSAR Development Partnership	• Publications and specifications available from the AUTOSAR Web site • Technical Solution 1-2WFS27 on the MathWorks Web site • Chapter 24, "Generating Code for AUTOSAR Software Components"

5. MISRA® and MISRA C® are registered trademarks of MISRA® Ltd., held on behalf of the MISRA® Consortium.

Standard or Guidelines	Organization	For More Information, See...
Standard: IEC 61508, Functional safety of electrical/electronic/ programmable electronic safety-related systems	International Electrotechnical Commission	• IEC functional safety zone Web site • Model-Based Design for IEC 61508 (Excerpts) — For the complete document, see Technical Solution 1-32COJP on the MathWorks Web site. • "Developing Models and Code That Comply with the IEC 61508 Standard" on page 22-6
Standard: ISO 26262, Road Vehicles - Functional Safety	International Organization for Standardization	• ISO 26262 Support in MATLAB and Simulink • "Developing Models and Code That Comply with the ISO 26262 Standard" on page 22-8
Standard: DO-178B, Software Considerations in Airborne Systems and Equipment Certification	Radio Technical Commission for Aeronautics (RTCA)	• Model-Based Design for DO-178B (Excerpts) — For the complete document, see Technical Solution 1-1ZLDDE on the MathWorks Web site. • "Developing Models and Code That Comply with the DO-178B Standard" on page 22-10

For information on whether Simulink Coder technology is certified or qualified and whether safety-critical software has been developed with MathWorks tools, see Embedded Coder — Code Certification with MathWorks Tools.

Developing Models and Code That Comply with MAAB Guidelines

The MathWorks Automotive Advisory Board (MAAB) involves major automotive OEMs and suppliers in the process of evolving MathWorks controls, simulation, and code generation products, including Simulink, Stateflow, and Simulink Coder. An important result of the MAAB has been the MAAB Guidelines.

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem, and the code that you generate from it, complies with MAAB guidelines. To check your model or subsystem, open the Simulink Model Advisor. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks** and run the MathWorks Automotive Advisory Board checks.

For more information on using the Model Advisor, see "Consulting the Model Advisor" in the Simulink documentation.

Developing Models and Code That Comply with MISRA C Guidelines

The Motor Industry Software Reliability Association (MISRA[6]) has established "Guidelines for the Use of the C Language in Critical Systems" (MISRA C). For general information about MISRA C, see `www.misra-c.com`.

To configure a model or subsystem so that the code generator is most likely to produce MISRA-C:2004 compliant code, use the Code Generation Advisor. For more information, refer to:

- "Defining High-Level Code Generation Objectives" on page 15-3
-

The Model Advisor also checks that you developed your model or subsystem to increase the likelihood of generating MISRA-C:2004 compliant code. To check your model or subsystem:

1. Open the Model Advisor.
2. Navigate to **By Product > Embedded Coder**.
3. Run the following checks:

- "Check for blocks not recommended for MISRA-C:2004 compliance"
- "Check configuration parameters for MISRA-C:2004 compliance"

For more information, see "Consulting the Model Advisor" in the Simulink documentation.

For information about using Embedded Coder software within MISRA C guidelines, see Technical Solution 1-1IFP0W on the MathWorks Web site.

6. MISRA® and MISRA C® are registered trademarks of MISRA® Ltd., held on behalf of the MISRA® Consortium.

Developing Models and Code That Comply with the IEC 61508 Standard

In this section...
"Applying Simulink and Embedded Coder to the IEC 61508 Standard" on page 22-6
"Checking for IEC 61508 Standard Compliance Using the Model Advisor" on page 22-6
"Validating Traceability" on page 22-11

Applying Simulink and Embedded Coder to the IEC 61508 Standard

Applying Model-Based Design successfully to a safety-critical system requires extra consideration and rigor to ensure the system adheres to defined safety standards. IEC 61508, Functional safety of electrical/electronic/programmable electronic safety related systems, is such a standard. Because the standard was published when most software was coded by hand, the standard needs to be mapped to Model-Based Design technologies. Model-Based Design for IEC 61508 (Excerpts) provides a sampling of information available from a document that offers recommendations on how to apply Simulink, Simulink Coder, and third-party products for Model-Based Design to IEC 61508 measures and techniques. For the complete version of Model-Based Design for IEC 61508, see Technical Solution 1-32COJP on the MathWorks Web site

Checking for IEC 61508 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the IEC 61508 standard by running the Simulink Model Advisor. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508 and ISO 26262 Checks** or **By Task > Modeling Standards for IEC 61508** and run the "IEC 61508 and ISO 26262 Checks".

For more information on using the Model Advisor, see "Consulting the Model Advisor" in the Simulink documentation.

Validating Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The Requirements Management Interface (RMI) that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option when generating an HTML report during the code generation or build process.

Developing Models and Code That Comply with the ISO 26262 Standard

In this section...
"Applying Simulink and Embedded Coder to the ISO 26262 Standard" on page 22-8
"Checking for ISO 26262 Standard Compliance Using the Model Advisor" on page 22-8
"Validating Traceability" on page 22-11

Applying Simulink and Embedded Coder to the ISO 26262 Standard

Applying Model-Based Design successfully to a safety-critical system requires extra consideration and rigor to ensure the system adheres to defined functional safety standards. ISO 26262, Road Vehicles - Functional Safety, is such a standard. For further information about MathWorks support for ISO 26262, see ISO 26262 Support in MATLAB and Simulink.

Checking for ISO 26262 Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the ISO 26262 standard by running the Simulink Model Advisor. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > IEC 61508 and ISO 26262 Checks** or **By Task > Modeling Standards for ISO 26262** and run the "IEC 61508 and ISO 26262 Checks".

For more information on using the Model Advisor, see "Consulting the Model Advisor" in the Simulink documentation.

Validating Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The Requirements Management Interface (RMI) that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option when generating an HTML report during the code generation or build process.

Developing Models and Code That Comply with the DO-178B Standard

In this section...

"Applying Simulink and Embedded Coder to the DO-178B Standard" on page 22-10

"Checking for Standard Compliance Using the Model Advisor" on page 22-10

"Validating Traceability" on page 22-11

Applying Simulink and Embedded Coder to the DO-178B Standard

Applying Model-Based Design successfully to a safety-critical system, requires extra consideration and rigor to ensure the system adheres to defined safety standards. DO-178B, Software Considerations in Airborne Systems and Equipment Certification, is such a standard. Because the standard was published when most software was coded by hand, the standard needs to be mapped to Model-Based Design technologies. Model-Based Design for DO-178B (Excerpts) provides a sampling of information available from a document that offers recommendations on how to apply Simulink, Simulink Coder, and third-party products for Model-Based Design to DO-178B measures and techniques. For the complete version of Model-Based Design for DO-178B, see Technical Solution 1-1ZLDDE on the MathWorks Web site.

Checking for Standard Compliance Using the Model Advisor

If you have a Simulink Verification and Validation product license, you can check that your Simulink model or subsystem and the code that you generate from it complies with selected aspects of the DO-178B standard by running the Simulink Model Advisor. Navigate to **By Product > Simulink Verification and Validation > Modeling Standards > DO-178B Checks** or **By Task > Modeling Standards for DO-178B** and run the DO-178B checks.

For more information on using the Model Advisor, see "Consulting the Model Advisor" in the Simulink documentation.

Validating Traceability

Typically, applications that require certification require some level of traceability between requirements, models, and corresponding code.

To...	Use...
Associate requirements documents with objects in Simulink models	The Requirements Management Interface (RMI) that is available if you have a Simulink Verification and Validation license.
Trace model blocks and subsystems to generated code	The **Model-to-code** traceability option when generating an HTML report during the code generation or build process.
Trace generated code to model blocks and subsystems	The **Code-to-model** traceability option when generating an HTML report during the code generation or build process.

Generating Reentrant Code from MATLAB Code

- “What Is Reentrant Code?” on page 23-2
- “When to Generate Reentrant Code” on page 23-3
- “How to Generate Reentrant Code” on page 23-4
- “Generated Code API” on page 23-5
- “How to Call Reentrant Code in a Single-Thread Environment” on page 23-6
- “How to Call Reentrant Code in a Multithreaded Environment” on page 23-7
- “Example: Calling Reentrant Code with No Persistent or Global Data (UNIX Only)” on page 23-9
- “Example: Calling Reentrant Code — Multithreaded with Persistent Data (Windows Only)” on page 23-15
- “Example: Calling Reentrant Code — Multithreaded with Persistent Data (UNIX Only)” on page 23-21

What Is Reentrant Code?

Reentrant code is a reusable programming routine that multiple programs can use simultaneously. Operating systems and other system software that uses multithreading to handle concurrent events use reentrant code. Sharing code with persistent or static data in a concurrent environment is difficult because multiple threads or processes might attempt to simultaneously read and write the static data. Reentrant code does not contain any static data. Calling programs maintain their state variables and pass them into the function. Therefore, any number of threads or processes can share one copy of a reentrant routine.

With an Embedded Coder license, you can use `codegen` to generate reusable code. For more information, see "How to Generate Reentrant Code" on page 23-4.

When to Generate Reentrant Code

Generate reentrant code when you want to:

- Deploy your code in multi-threaded environments.
- Share the same algorithm with different persistent data.
- Compile code that uses function variables that are too large to fit on the stack.

If you do not choose to generate reentrant code, `codegen` generates code that uses statically allocated memory for function variables that are too large to fit on the stack, and for global and persistent variables. The use of static memory allocation for these variables means that you cannot deploy the generated code in environments that require code to be reentrant. In addition, the generated code can result in static memory size overflow if you cannot adjust the static memory allocation size to accommodate the static memory requirements of the program.

When you generate reentrant code, `codegen` creates input data structures for function variables that are too large to fit on the stack, and for persistent and global variables. You can then dynamically allocate memory for these input structures. The use of dynamic memory allocation means that you can deploy the code in reentrant environments.

How to Generate Reentrant Code

Prerequisites

This option requires an Embedded Coder license.

Procedure

Use the `MultiInstanceCode` option of the `coder.EmbeddedCodeConfig` code generation configuration object. For example, to compile the file `foo.m` and generate reusable code:

1. Create a code generation configuration object and enable the `MultiInstanceCode` option.

   ```
   cfg = coder.config('lib', 'ecoder', true);
   cfg.MultiInstanceCode = true;
   ```

2. Pass the configuration object to `codegen` using the `-config` option.

   ```
   codegen -config cfg  foo
   ```

Alternatively, you can set this parameter using the MATLAB® Coder™ Project Settings dialog box. On the **Interface** pane , select **Generate reusable code**.

Generated Code API

When you generate reusable code, `codegen` supports dynamic allocation of function variables that are too large for the stack, as well as persistent and global variables. It generates a header file, *`primary_function_name`*`_types.h`, which you must include when using the generated code. This header file contains the following structures:

- *`primary_function_name`*`StackData`

  This structure contains the user allocated memory. You must pass a pointer to this structure as the first parameter to all functions that use it either directly, because the function uses a field in the structure, or indirectly, because the function passes the structure to a called function.

  The *`primary_function_name`*`StackData` structure also contains a pointer to the *`primary_function_name`*`PersistentData` structure if the algorithm uses persistent or global data. Including this pointer means that you have to pass only one parameter to each calling function.

- *`primary_function_name`*`PersistentData`

  If your algorithm uses persistent or global variables, `codegen` provides a separate structure for them and adds a pointer to this structure to the memory allocation structure. Having a separate structure for persistent and global variables allows you to allocate memory for these variables once and share them with all threads if desired. However, if there is no communication between threads, you can choose to allocate memory for these variables per thread or per application.

For more information on using these global structures, see "Multithreaded Examples" on page 23-7.

How to Call Reentrant Code in a Single-Thread Environment

To call reentrant code in a single-thread environment, create a `main` function that:

- Includes the header file *primary_function_name*.h.
- Allocates memory for the global memory allocation structure *primary_function_name*`StackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure *primary_function_name*`PersistentData`, .
- Calls these functions:
  - *primary_function_name*`_initialize`.
  - *primary_function_name*.
  - *primary_function_name*`_terminate`.

  When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, `codegen` automatically generates two housekeeping functions that you must call with the C/C++ function. For more information, see "Calling Initialize and Terminate Functions".
- Frees the memory used for global structures.

How to Call Reentrant Code in a Multithreaded Environment

To call reentrant code, create a `main` function that:

- Includes the header file *`primary_function_name`*.h.
- For each thread, allocates memory for the global memory allocation structure *`primary_function_name`*`StackData`.
- If the algorithm uses persistent or global data, allocates memory for the global structure *`primary_function_name`*`PersistentData`. If there is communication between threads, you must allocate this memory once for the application. Otherwise, you can choose to allocate memory per thread or per application.
- Contains a thread function that calls these functions:
  - *`primary_function_name`*`_initialize`.
  - *`primary_function_name`*.
  - *`primary_function_name`*`_terminate`.

  When you convert a MATLAB function to a C/C++ library function or a C/C++ executable, `codegen` automatically generates two housekeeping functions that you must call with the C/C++ function. For more information, see "Calling Initialize and Terminate Functions" in the MATLAB Coder documentation.
- Initializes each thread and passes in a pointer to the memory allocation structure as the first parameter to the thread function.
- Frees the memory used for global structures.

Multithreaded Examples

Type of Reentrant Code	Platform	Reference
Multithreaded with no persistent or global data	Windows	"Generating Reentrant C Code from MATLAB Code"
	UNIX	"Example: Calling Reentrant Code with No Persistent or Global Data (UNIX Only)" on page 23-9

Type of Reentrant Code	Platform	Reference
Multithreaded with persistent or global data	Windows	"Example: Calling Reentrant Code — Multithreaded with Persistent Data (Windows Only)" on page 23-15
	UNIX	"Example: Calling Reentrant Code — Multithreaded with Persistent Data (UNIX Only)" on page 23-21

Example: Calling Reentrant Code with No Persistent or Global Data (UNIX Only)

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a simple multithreaded example that uses no persistent or global data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```
function Y = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential
    % of the input matrix using Taylor series and returns
    % the computed output.
    %
    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E;
```

Providing a main Function

To call the reentrant code, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see “Calling Initialize and Terminate Functions”.
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    matrix_expStackData* spillData;
} IODATA;

/* The thread_function calls the matrix_exp function written in MATLAB */
void *thread_function(void *dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize();
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out);
    matrix_exp_terminate();
}

int main() {
    pthread_t thread1, thread2;
    int  iret1, iret2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expStackData* sd1=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2=(matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

    data1.spillData = sd1;
    data2.spillData = sd2;
```

```
    for (i=0;i<NUMELEMENTS;i++) {
        data1.in[i] = 1;
        data1.out[i] = 0;
        data2.in[i] = 1.1;
        data2.out[i] = 0;
    }

    /*Initializing the 2 threads and passing appropriate data to the thread functions*/
    printf("Starting thread 1...\n");
    iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
    if (iret1 != 0){
  perror( "Thread 1 creation failed.");
  exit(EXIT_FAILURE);
 }

    printf("Starting thread 2...\n");
    iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
    if (iret2 != 0){
        perror( "Thread 2 creation failed.");
        exit(EXIT_FAILURE);
    }

    /*Wait for both the threads to finish execution*/
    iret1 = pthread_join(thread1, NULL);
    if (iret1 != 0){
  perror( "Thread 1 join failed.");
  exit(EXIT_FAILURE);
 }

    iret2 = pthread_join(thread2, NULL);
    if (iret2 != 0){
  perror( "Thread 2 join failed.");
  exit(EXIT_FAILURE);
 }

    free(sd1);
    free(sd2);

    printf("Finished Execution!\n");
    exit(EXIT_SUCCESS);
```

```
}
```

Generating Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```
% This example can only be run on Unix platforms
if ~isunix
  error('This example requires pthread libraries and can only be run on Unix.');
end

% Setting the correct options for the Config object

% Create a code gen configuration object
e = coder.config('exe','ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
e.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config e main.c matrix_exp.m -report -args ones(160,160)
```

These commands:

- Check that the example is running on UNIX platforms and generates an error message if not.
- Create a Simulink Coder configuration object for an ERT target.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Use the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

  ```
  function setbuildargs(buildInfo)
  ```

```
% The example being compiled requires pthread support.
% The -lpthread flag requests that the pthread library
% be included in the build
    linkFlags = {'-lpthread'};
    addLinkFlags(buildInfo, linkFlags);
```

For more information about the `PostCodeGenCommand` option, see "Customizing the Post-Code-Generation Build Process".

- Invoke `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `e`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with the correct class, size, and complexity.

  For more information on these options, see `codegen`.

Examining the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines the `matrix_expStackData` global structure. This structure contains local variables that are too large to fit on the stack.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
   struct {
     real_T F[25600];
     real_T Y[25600];
   } f0;
```

```
} matrix_expStackData;
#endif
/* End of MATLAB Coder code generation (matrix_exp_types.h) */
```

Running the Code

Finally, call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports successful completion.

Example: Calling Reentrant Code — Multithreaded with Persistent Data (Windows Only)

This example requires libraries that are specific to the Microsoft® Windows operating system and, therefore, runs only on Windows platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential
    % of the input matrix using Taylor series and returns
    % the computed output. It also returns the number of
    % times this function has been called.
    %
    persistent count;
    if isempty(count)
        count = 0;
    end
    count = count+1;

    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E ;

    numTimes = count;
```

Providing a main Function

To call reentrant code that uses persistent data, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see "Calling Initialize and Terminate Functions".
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
DWORD WINAPI thread_function(PVOID dummyPtr) {
    IODATA *myIOData = (IODATA*)dummyPtr;
    matrix_exp_initialize(myIOData->spillData);
```

```
    matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData->numTimes);
    printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
    matrix_exp_terminate();
    return 0;
}

void main() {
    HANDLE thread1, thread2;
    IODATA data1;
    IODATA data2;
    int32_T i;

    /*Initializing data for passing to the 2 threads*/
    matrix_expPersistentData* pd1 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expPersistentData* pd2 = (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
    matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
    matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

    sd1->pd = pd1;
    sd2->pd = pd2;
    data1.spillData = sd1;
    data2.spillData = sd2;

    for (i=0;i<NUMELEMENTS;i++) {
        data1.in[i] = 1;
        data1.out[i] = 0;
        data2.in[i] = 1.1;
        data2.out[i] = 0;
    }

    data1.numTimes = 0;
    data2.numTimes = 0;

    /*Initializing the 2 threads and passing appropriate data to the thread functions*/
    printf("Starting thread 1...\n");
    thread1 = CreateThread(NULL, 0, thread_function, (PVOID) &data1, 0, NULL);
    if (thread1 == NULL){
  perror( "Thread 1 creation failed.");
  exit(EXIT_FAILURE);
 }
```

```
    printf("Starting thread 2...\n");
    thread2 = CreateThread(NULL, 0, thread_function, (PVOID) &data2, 0, NULL);
    if (thread2 == NULL){
        perror( "Thread 2 creation failed.");
        exit(EXIT_FAILURE);
    }

    /*Wait for both the threads to finish execution*/
    if (WaitForSingleObject(thread1, INFINITE) != WAIT_OBJECT_0){
  perror( "Thread 1 join failed.");
  exit(EXIT_FAILURE);
 }

    if (WaitForSingleObject(thread2, INFINITE) != WAIT_OBJECT_0){
  perror( "Thread 2 join failed.");
  exit(EXIT_FAILURE);
 }

    free(sd1);
    free(sd2);
    free(pd1);
    free(pd2);

    printf("Finished Execution!\n");
    exit(EXIT_SUCCESS);

}
```

Generating Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```
% This example can only be run on Windows platforms
if ~ispc
 error...
   ('This example requires Windows-specific libraries and can only be run on Windows.');
end

% Setting the correct options for the Config object
% Create a code gen configuration object
e = coder.config('exe', 'ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Compiling
codegen -config e main.c -report matrix_exp.m -args ones(160,160)
```

These commands:

- Check that the example is running on Windows platforms and generates an error message if not.
- Create a code generation configuration object for an ERT target.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Invoke `codegen` with the following options:
  - `-config` to pass in the code generation configuration object `e`.
  - `main.c` to include this file in the compilation.
  - `-report` to create a code generation report.
  - `-args` to specify an example input with the correct class, size, and complexity.

  For more information on these options, see `codegen`.

Examining the Generated Code

`codegen` generates a header file `matrix_exp_types.h`, which defines:

- The `matrix_expStackData` global structure that contains local variables that are too large to fit on the stack and a pointer to the `matrix_expPersistentData` global structure.
- The `matrix_expPersistentData` global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
  real_T count;
} matrix_expPersistentData;
typedef struct {
   struct {
      real_T F[25600];
      real_T Y[25600];
   } f0;
   matrix_expPersistentData *pd;
} matrix_expStackData;
#endif
/* End of code generation (matrix_exp_types.h) */
```

Running the Code

Finally, call the code using the command:

```
system('matrix_exp.exe')
```

The executable runs and reports successful completion.

Example: Calling Reentrant Code — Multithreaded with Persistent Data (UNIX Only)

This example requires POSIX thread (pthread) libraries and, therefore, runs only on UNIX platforms. It is a multithreaded example that uses persistent data. Two threads call the MATLAB function `matrix_exp` with different sets of input data.

MATLAB Code Used for This Example

```
function [Y,numTimes] = matrix_exp(X) %#codegen
    %
    % The function matrix_exp computes matrix exponential
    % of the input matrix using Taylor series and returns
    % the computed output. It also returns the number of
    % times this function has been called.
    %

    persistent count;
    if isempty(count)
        count = 0;
    end
    count = count+1;

    E = zeros(size(X));
    F = eye(size(X));
    k = 1;
    while norm(E+F-E,1) > 0
        E = E + F;
        F = X*F/k;
        k = k+1;
    end
    Y = E ;

    numTimes = count;
```

Providing a main Function

To call reentrant code that uses persistent data, you must provide a `main` function that:

- Includes the header file `matrix_exp.h`.
- For each thread, allocates memory for stack data.
- Allocates memory for persistent data, once per application if threads share data, and once per thread otherwise.
- Calls the `matrix_exp_initialize` housekeeping function. For more information, see "Calling Initialize and Terminate Functions".
- Calls `matrix_exp`.
- Calls `matrix_exp_terminate`.
- Frees the memory used for stack and persistent data.

For this example, `main.c` contains:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "matrix_exp.h"
#include "matrix_exp_initialize.h"
#include "matrix_exp_terminate.h"
#include "rtwtypes.h"
#define NUMELEMENTS (160*160)

typedef struct {
    real_T in[NUMELEMENTS];
    real_T out[NUMELEMENTS];
    real_T numTimes;
    matrix_expStackData* spillData;
} IODATA;

/*The thread_function calls the matrix_exp function written in MATLAB*/
void *thread_function(void *dummyPtr) {
  IODATA *myIOData = (IODATA*)dummyPtr;
  matrix_exp_initialize(myIOData->spillData);
```

```
  matrix_exp(myIOData->spillData, myIOData->in, myIOData->out, &myIOData>numTimes);
  printf("Number of times function matrix_exp is called is %g\n",myIOData->numTimes);
  matrix_exp_terminate();
}

int main() {
  pthread_t thread1, thread2;
  int  iret1, iret2;
  IODATA data1;
  IODATA data2;
  int32_T i;

  /*Initializing data for passing to the 2 threads*/
  matrix_expPersistentData* pd1 =
     (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
  matrix_expPersistentData* pd2 =
     (matrix_expPersistentData*)calloc(1,sizeof(matrix_expPersistentData));
  matrix_expStackData* sd1 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));
  matrix_expStackData* sd2 = (matrix_expStackData*)calloc(1,sizeof(matrix_expStackData));

  sd1->pd = pd1;
  sd2->pd = pd2;
  data1.spillData = sd1;
  data2.spillData = sd2;

  for (i=0;i<NUMELEMENTS;i++) {
      data1.in[i] = 1;
      data1.out[i] = 0;
      data2.in[i] = 1.1;
      data2.out[i] = 0;
  }

  data1.numTimes = 0;
  data2.numTimes = 0;

  /*Initializing the 2 threads and passing appropriate data to the thread functions*/
  printf("Starting thread 1...\n");
  iret1 = pthread_create(&thread1, NULL, thread_function, (void*) &data1);
  if (iret1 != 0){
   perror("Thread 1 creation failed.");
```

```
exit(EXIT_FAILURE);
 }

  printf("Starting thread 2...\n");
  iret2 = pthread_create(&thread2, NULL, thread_function, (void*) &data2);
  if (iret2 != 0){
      perror( "Thread 2 creation failed.");
      exit(EXIT_FAILURE);
  }

  /*Wait for both the threads to finish execution*/
  iret1 = pthread_join(thread1, NULL);
  if (iret1 != 0){
   perror( "Thread 1 join failed.");
exit(EXIT_FAILURE);
 }

  iret2 = pthread_join(thread2, NULL);
  if (iret2 != 0){
   perror( "Thread 2 join failed.");
exit(EXIT_FAILURE);
 }

  free(sd1);
  free(sd2);
  free(pd1);
  free(pd2);

  printf("Finished Execution!\n");
  return(0);

}
```

Generating Reentrant C Code

Run the following script at the MATLAB command line to generate code.

```
% This example can only be run on Unix platforms
if ~isunix
  error('This example requires pthread libraries and can only be run on Unix.');
end

% Setting the correct options for the Config object

% Specify an ERT target
e = coder.config('exe','ecoder', true);

% Enable reentrant code generation
e.MultiInstanceCode = true;

% Set the post code generation command to be the 'setbuildargs' function
e.PostCodeGenCommand = 'setbuildargs(buildInfo)';

% Compiling
codegen -config e main.c  -report matrix_exp.m -args ones(160,160)
```

These commands:

- Check that the example is running on UNIX platforms and generates an error message if not.
- Create a code generation configuration object.
- Enable the `MultiInstanceCode` option to generate reusable, reentrant code.
- Use the `PostCodeGenCommand` option to set the post-code-generation command to be the `setbuildargs` function. This function sets the `-lpthread` flag to specify that the build include the pthread library.

  ```
  function setbuildargs(buildInfo)
  % The example being compiled requires pthread support.
  % The -lpthread flag requests that the pthread library
  % be included in the build
      linkFlags = {'-lpthread'};
      addLinkFlags(buildInfo, linkFlags);
  ```

For more information about the PostCodeGenCommand option, see "Customizing the Post-Code-Generation Build Process".

- Invokes codegen with the following options:
  - -config to pass in the code generation configuration object e.
  - main.c to include this file in the compilation.
  - -report to create a code generation report.
  - -args to specify an example input with the correct class, size, and complexity.

  For more information on these options, see codegen.

Examining the Generated Code

codegen generates a header file matrix_exp_types.h, which defines:

- The matrix_expStackData global structure that contains local variables that are too large to fit on the stack and a pointer to the matrix_expPersistentData global structure.
- The matrix_expPersistentData global structure that contains persistent data.

```
/*
 * matrix_exp_types.h
 *
 * MATLAB Coder code generation for function 'matrix_exp'
 *
 */

#ifndef __MATRIX_EXP_TYPES_H__
#define __MATRIX_EXP_TYPES_H__

/* Type Definitions */
typedef struct {
  real_T count;
} matrix_expPersistentData;
typedef struct {
  struct {
```

```
    real_T F[25600];
    real_T Y[25600];
  } f0;
  matrix_expPersistentData *pd;
} matrix_expStackData;
#endif
/* End of code generation (matrix_exp_types.h) */
```

Running the Code

Finally, call the code using the command:

```
system('./matrix_exp')
```

The executable runs and reports successful completion.

Generating Code for AUTOSAR Software Components

- "Overview of AUTOSAR Support" on page 24-2
- "Simulink Modeling Patterns for AUTOSAR" on page 24-3
- "Workflow for AUTOSAR" on page 24-26
- "Importing an AUTOSAR Software Component" on page 24-28
- "Preparing a Simulink Model for AUTOSAR Code Generation" on page 24-31
- "Generating AUTOSAR Code and Description Files" on page 24-58
- "Configuring AUTOSAR Options Programmatically" on page 24-64
- "Verifying the AUTOSAR Code with SIL and PIL Simulations" on page 24-65
- "Limitations and Tips" on page 24-68
- "Demos and Further Reading" on page 24-73

Overview of AUTOSAR Support

Embedded Coder software supports AUTOSAR (*AUTomotive Open System ARchitecture*), an open and standardized automotive software architecture. AUTOSAR is developed jointly by automobile manufacturers, suppliers, and tool developers.

The AUTOSAR standard addresses:

- Architecture – Three layers, *Application*, *Runtime Environment (RTE)*, and *Basic Software*, enable decoupling of AUTOSAR Software Components from the execution platform. Standard interfaces between AUTOSAR Software Components and the Runtime Environment allow reuse or relocation of components within the Electronic Control Unit (ECU) topology of a vehicle.
- Methodology – Specification of code formats and description file templates, for example.
- Application Interfaces – Specification of interfaces for typical automotive applications.

For details on the AUTOSAR standard, go to www.autosar.org.

In Simulink, you can model AUTOSAR Software Components and related concepts. See "Simulink Modeling Patterns for AUTOSAR" on page 24-3.

Using Embedded Coder software, you can generate AUTOSAR-compliant code and description files. See "Workflow for AUTOSAR" on page 24-26.

Simulink Modeling Patterns for AUTOSAR

In this section...

"About Simulink Modeling Patterns for AUTOSAR" on page 24-3
"AUTOSAR Software Components" on page 24-3
"AUTOSAR Communication" on page 24-9
"Calibration Parameters" on page 24-15
"Inter-Runnable Variables" on page 24-16
"Data Types" on page 24-17
"Per-Instance Memory" on page 24-22
"AUTOSAR Terminology" on page 24-23

About Simulink Modeling Patterns for AUTOSAR

This section describes how you model AUTOSAR Software Components and related concepts in Simulink.

AUTOSAR Software Components

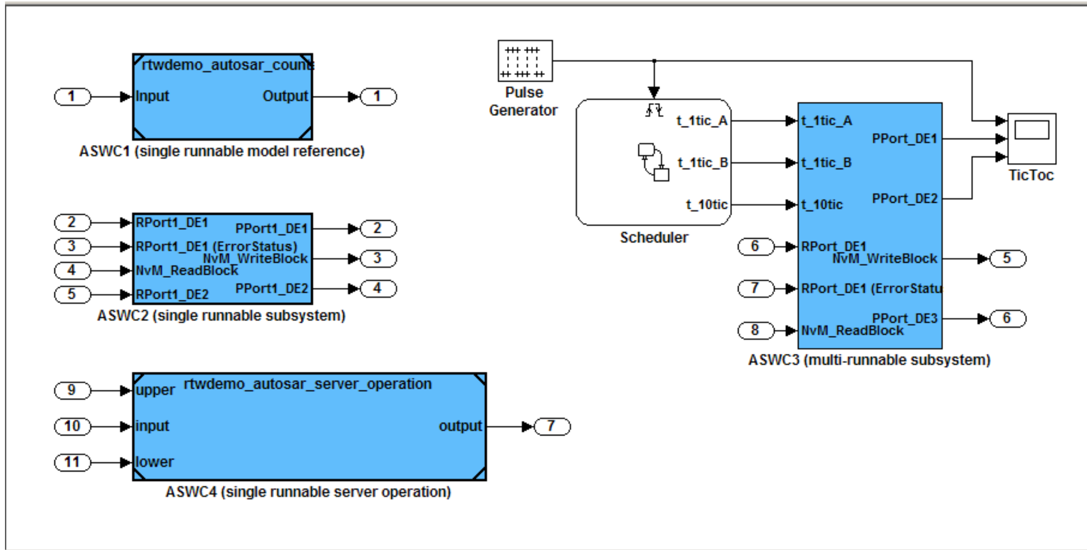
In AUTOSAR, application software consists of separate units, *AUTOSAR Software Components*.

Note An AUTOSAR Software Component is sometimes referred to as *atomic* because it is never split across more than one Electronic Control Unit (ECU). Do not confuse *atomic* in this context with the concept of Simulink atomic subsystems.

The behavior of an AUTOSAR Software Component is implemented by a single or multiple *runnable entities* (runnables), which expose well-defined connection points, *ports*.

In Simulink, you can represent an AUTOSAR Software Component using a model or a subsystem. For example, the following figure shows modeling

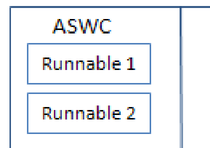
patterns for AUTOSAR Software Components (ASWC) labeled `ASWC1`, `ASWC2`, `ASWC3`, and `ASWC4`.



Runnables

AUTOSAR Software Components contain runnables that are directly or indirectly scheduled by the underlying AUTOSAR operating system.

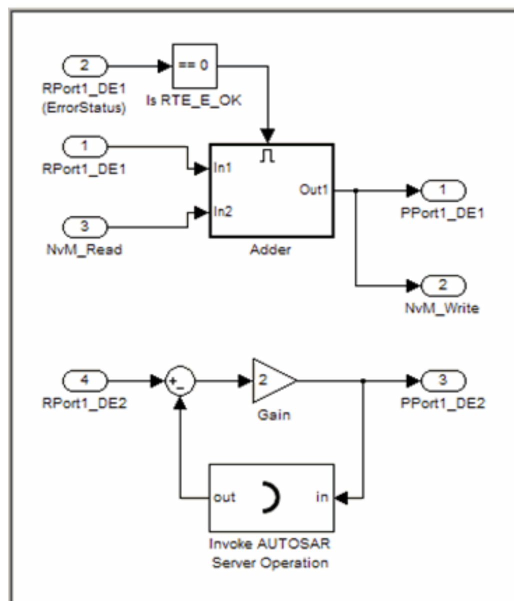
The following figure shows an AUTOSAR Software Component with two runnables, `Runnable 1` and `Runnable 2`. Each runnable is triggered by RTEEvents, events generated by the AUTOSAR Runtime Environment (RTE). For example, TimingEvent is an RTEEvent that is generated periodically.



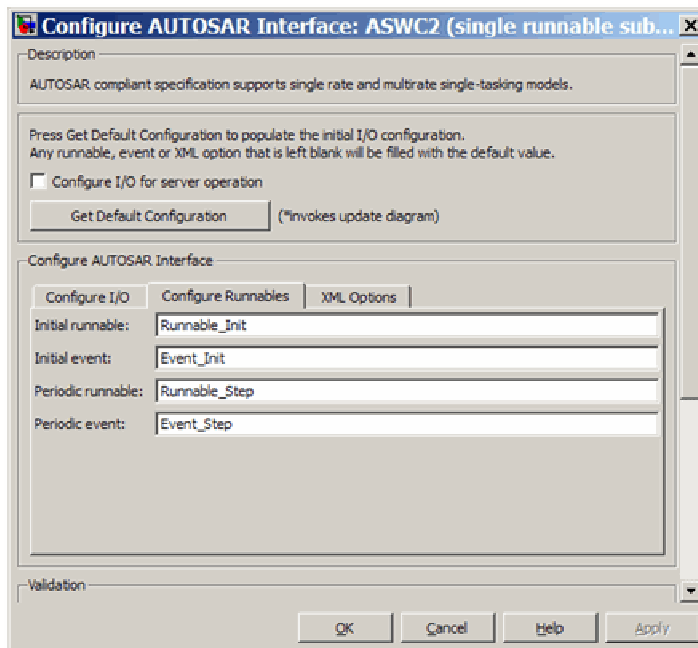
The components ASWC1, ASWC2 and ASWC4 contain single runnables. These components are represented by a subsystem or a model, and can be single- or multirate. However, the software implements each component as a single-tasking operation.

Note The software generates an additional runnable for the initialization function regardless of the modeling pattern.

ASWC2 is modeled as a single-rate, single-tasking atomic subsystem.

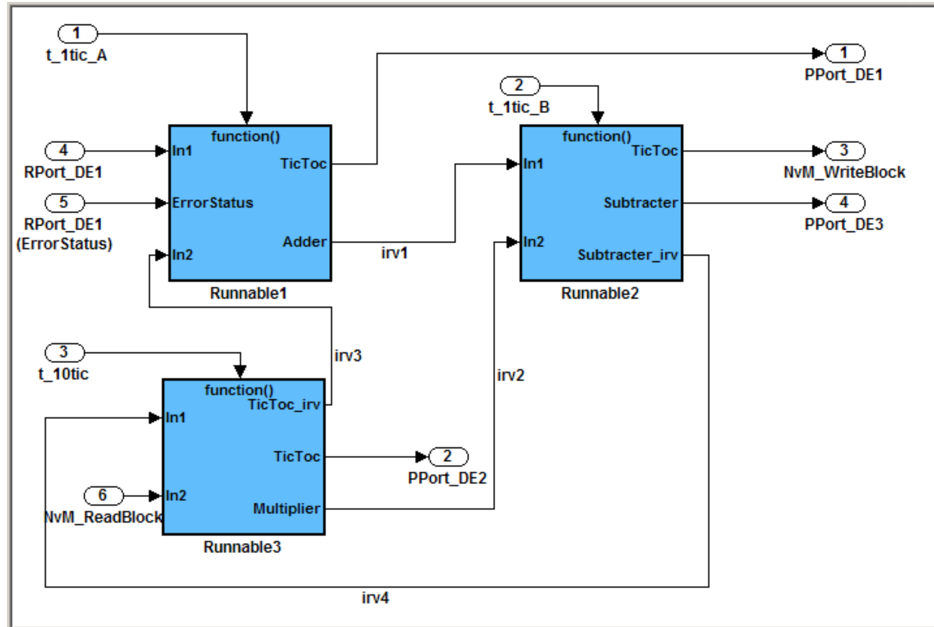


You can generate the ASWC2 runnable, which corresponds to the step function of the subsystem. Use the Configure AUTOSAR Interface dialog box to specify the names of the initial and periodic runnables, as shown by the following figure.



The software generates TimingEvents for the runnables. The TimingEvent period for the periodic runnable is the fundamental sample time of the model or atomic subsystem. Specify this sample time in the Subsystem Parameters dialog box, in the **Sample time (-1 for inherited)** field.

The component `ASWC3` contains multiple runnables.



Use the Export Functions feature to map the runnables to Simulink function-call subsystems. See "Configuring Multiple Runnables" on page 24-47 and "Exporting AUTOSAR Software Component" on page 24-61. The software also generates an initialization runnable for the initialization function.

Use the Configure AUTOSAR Interface dialog box to specify the names of the multiple runnables and the periods of TimingEvents.

Description

AUTOSAR compliant specification supports single rate and multirate single-tasking models.

Press Get Default Configuration to populate the initial I/O configuration.
Any runnable, event or XML option that is left blank will be filled with the default value.

☐ Configure I/O for server operation

Get Default Configuration (*invokes update diagram)

Configure AUTOSAR Interface

Input/Output | Inter-Runnable Variables | Runnables | XML Options

Initial runnable name: Runnable_Init

Runnable
Runnable1
Runnable2
Runnable3

Properties - Runnable1

Runnable name: Runnable1_Step

Event Type	Event Name	Execution Period	Trigger Port
TimingEvent	Event_Runnable1	0.001	

Add Event Delete Event

Validation

Validate (*invokes update diagram)

Multiple Instantiation

AUTOSAR supports multiple instantiations of software components. However, Simulink supports multiple instantiations (reentrant code) only if a model is configured as a server operation. See “Configuring a Server Operation” on page 24-40.

To generate reentrant code for a model configured as a server operation, on the **Code Generation > Interface** pane, select the **Generate reusable code** check box.

AUTOSAR Communication

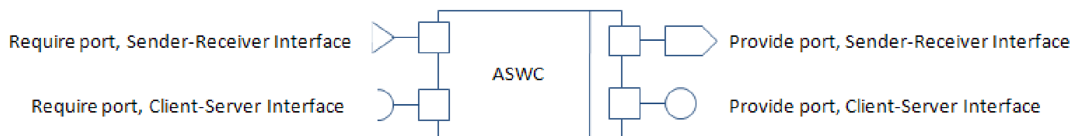
AUTOSAR Software Components provide well-defined connection points, ports. There are two types of AUTOSAR ports:

- `Require`
- `Provide`

In addition, these AUTOSAR ports can reference two kinds of interfaces:

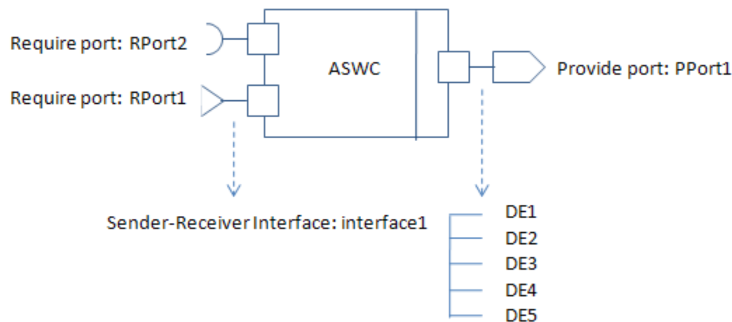
- Sender-Receiver
- Client-Server

The following figure shows an AUTOSAR Software Component with four ports representing all port and interface combinations.



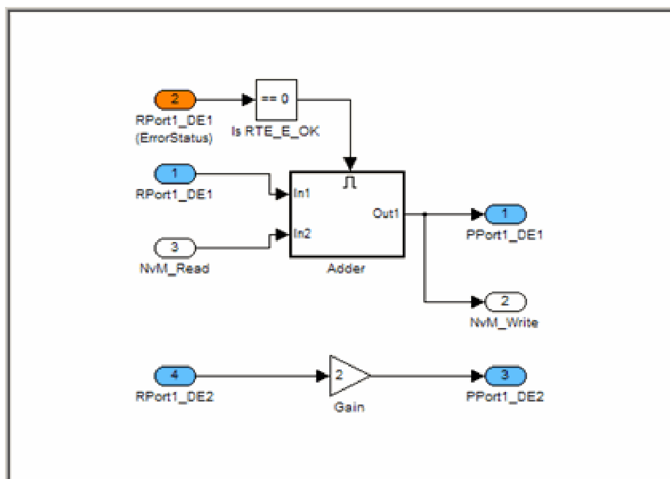
Sender-Receiver Interface

A Sender-Receiver Interface consists of one or more data elements. Although a `Require` or `Provide` port may reference a Sender-Receiver Interface, the AUTOSAR Software Component does not necessarily access all the data elements. For example, consider the following figure.



The AUTOSAR Software Component has a `Require` and `Provide` port that references the same Sender-Receiver Interface, `interface1`. Although this interface contains data elements `DE1`, `DE2`, `DE3`, `DE4`, and `DE5`, the component does not utilize all the data elements.

The following figure is an example of how you model, in Simulink, an AUTOSAR Software Component that accesses data elements.

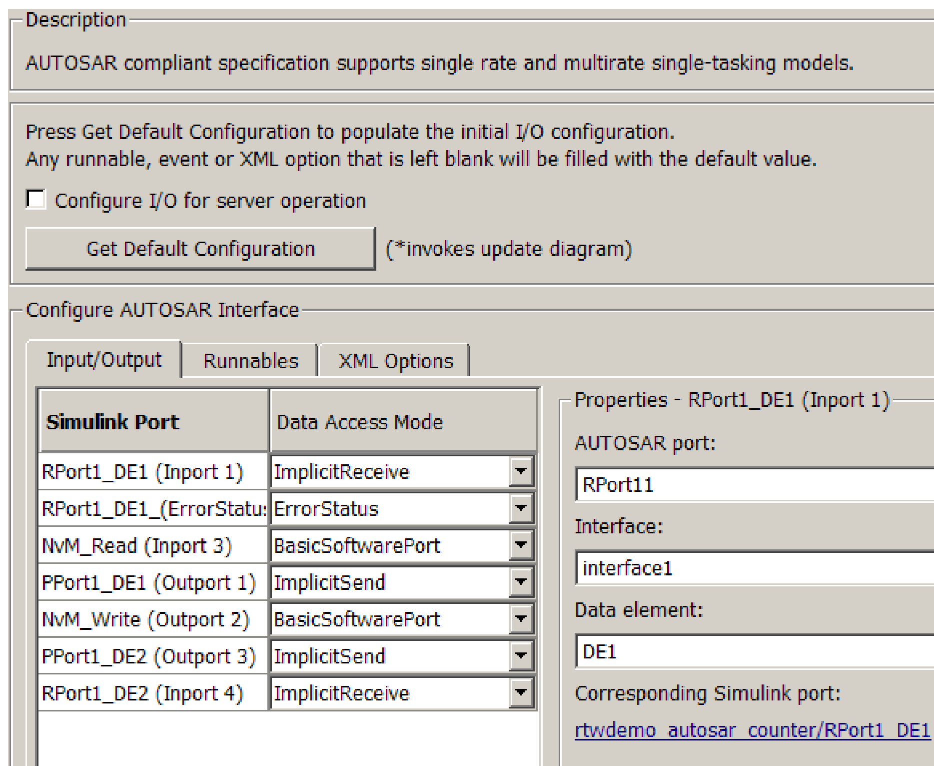


`ASWC` accesses data elements `DE1` and `DE2`. You model data element access as follows:

- For `Require` ports, use Simulink inports. For example, `RPort1_DE1` and `RPort1_DE2`.
- For `Provide` ports, use Simulink outports. For example, `PPort1_DE1` and `PPort1_DE2`.

ErrorStatus is a value that the AUTOSAR Runtime Environment (RTE) returns to indicate errors that the communication system detects for each data element. You can use a Simulink inport to model error status, for example, `RPort1_DE1 (ErrorStatus)`.

Use the Configure AUTOSAR Interface dialog box to specify the AUTOSAR settings for each inport and outport. The following figure shows settings for `ASWC`.

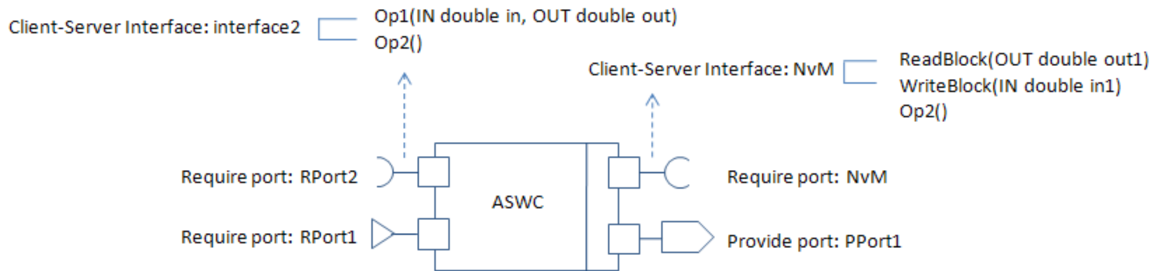


For example, the Data Access Mode for `RPort1_DE1` is set to `ImplicitReceive`. For information on how you specify settings, see "Using the Configure AUTOSAR Interface Dialog Box" on page 24-31.

Client-Server Interface

A Client-Server Interface consists of one or more operation prototypes. An operation prototype contains one or more arguments of specific data types. A Client-Server Interface can be referenced by either a `Require` or `Provide` port.

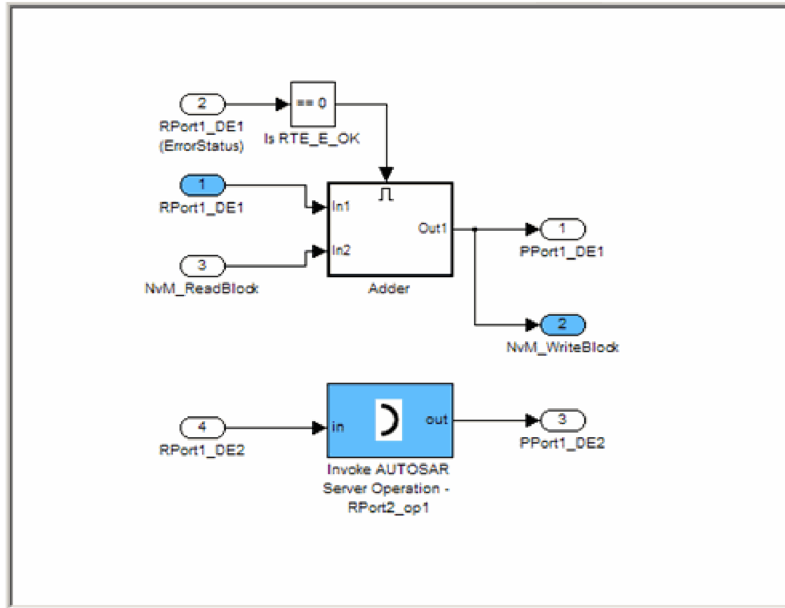
The following figure shows an AUTOSAR Software Component with `Require` ports (`RPort2` and `NvM`) that reference Client-Server Interfaces (`Interface2` and `NvM`).



Simulink provides the following modeling patterns for Client-Server Interfaces:

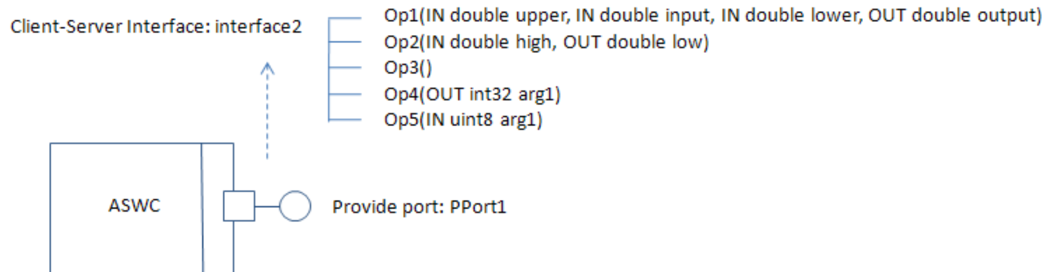
- If you want to invoke a Basic Software interface with operations that have only one argument, for example, `Client-Server Interface: NvM`, use an inport or outport.
- If you want to invoke Basic Software or application software interfaces that contain operations with any number of arguments, for example, `Client-Server Interface: Interface2`, use the Invoke AUTOSAR Server Operation block. See "Configuring the Invoke AUTOSAR Server Operation Block" on page 24-43

The following figure shows the use of the Invoke AUTOSAR Server Operation block in modeling an AUTOSAR Software Component in Simulink.

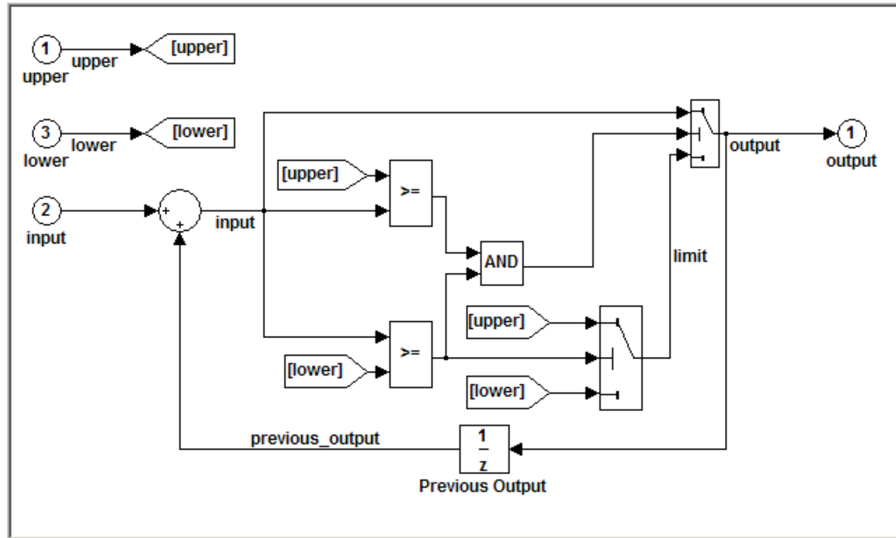


Use the Configure AUTOSAR Interface dialog box to specify the AUTOSAR settings for each inport and outport. See "Using the Configure AUTOSAR Interface Dialog Box" on page 24-31.

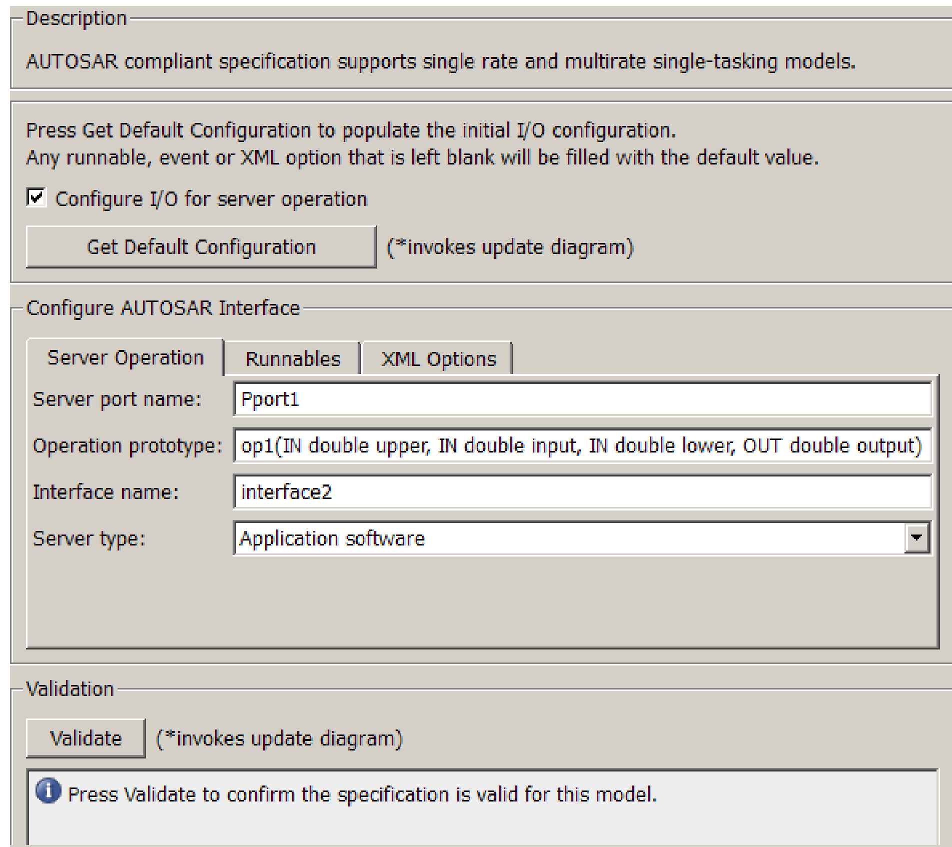
The following figure shows an AUTOSAR Software Component with a `Provide` port that references a Client-Server Interface.



In Simulink, you can model a single operation of an AUTOSAR Software Component that is referenced by a Client-Server Interface. Consider the following model.



Use the Configure AUTOSAR Interface dialog box to map the inports and outports to the arguments of the operation prototype. For example, the inports map to arguments `upper`, `input`, and `lower`.



For more information, see "Configuring a Server Operation" on page 24-40 .

Calibration Parameters

About Calibration Parameters

A calibration parameter is a value in an Electronic Control Unit (ECU). You tune or modify these parameters using a calibration data management tool or an offline calibration tool.

The AUTOSAR standard specifies the following types of calibration parameters:

- Calibration parameters that belong to a *calibration component*, which can be accessed by all AUTOSAR Software Components.

  You define calibration components using an AUTOSAR authoring tool.
- Internal calibration parameters, which are defined and accessed by only one AUTOSAR Software Component.

The software supports import, export, and code generation for both types of calibration parameters.

Importing and Exporting Calibration Parameters

You can import calibration parameters into the MATLAB base workspace.

For example, to import parameters from an AUTOSAR calibration component description, use `arxml.importer.createCalibrationComponentObjects`.

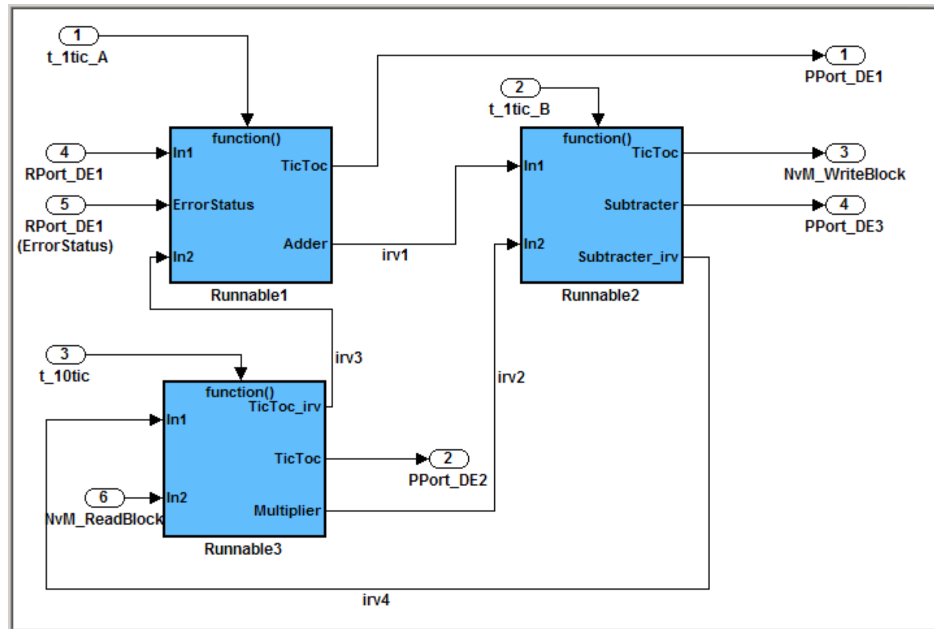
To provide your Simulink model with access to these parameters, assign the imported parameters to block parameters.

For more information, see "Importing an AUTOSAR Software Component" on page 24-28.

You can specify the type of calibration parameter exported by configuring properties of the corresponding block parameter in the base workspace. See "Configuring Calibration Parameters" on page 24-53 and rtwdemo_autosar_legacy_script.

Inter-Runnable Variables

In AUTOSAR, *inter-runnable* variables are used to communicate primitive type data between runnables in the same component. You define these variables in a Simulink model by the signal lines that connect subsystems (runnables). For example, in the following figure, `irv1`, `irv2`, `irv3`, and `irv4` are inter-runnable variables.



You can specify the names and data access modes of the inter-runnable variables that you export. See "Configuring Inter-Runnable Variables" on page 24-48.

Data Types

AUTOSAR specifies data types that apply to:

- Data elements of a Sender-Receiver Interface
- Operation arguments of a Client-Server Interface
- Calibration parameters
- Inter-runnable variables

The data types fall into two categories:

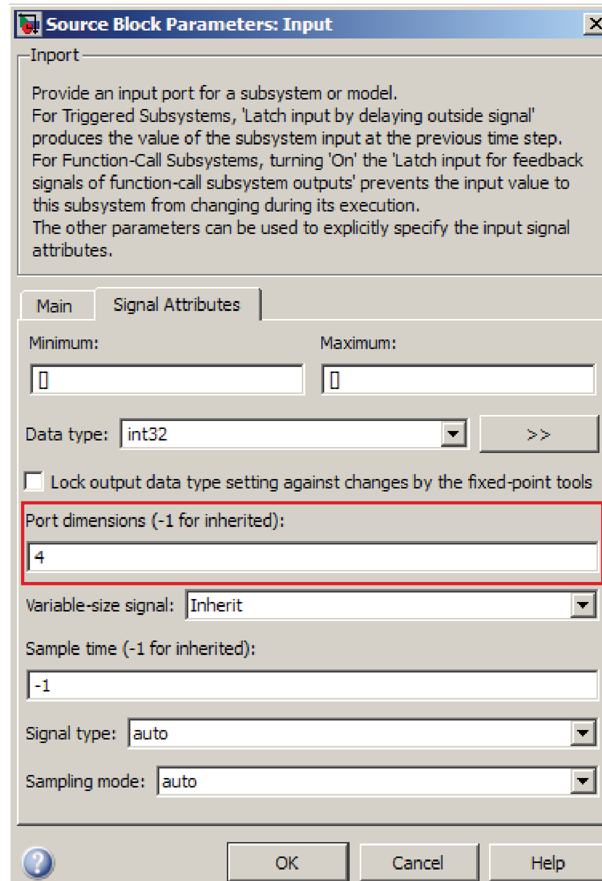
- Primitive data types, which allow a direct mapping to C intrinsic types.
- Composite data types, which map to C arrays and structures.

You can use Simulink data types to define AUTOSAR primitive types.

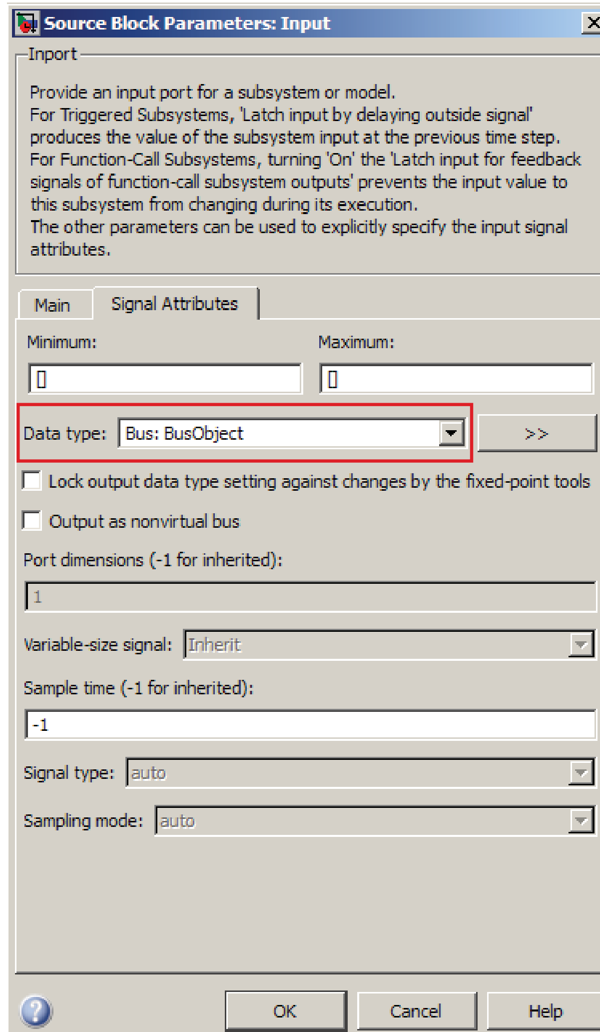
AUTOSAR Data Type	Simulink Data Type
UInt4	uint8
SInt4	int8
UInt8	uint8
SInt8	int8
UInt16	uint16
SInt16	int16
UInt32	uint32
SInt32	int32
Float_with_NaN	single
Float	single
Double_with_NaN	double
Double	double
Boolean	boolean
Char8	uint8
Char16	Not supported

AUTOSAR composite data types are arrays and records, which are represented in Simulink by wide signals and bus objects, respectively. In the Inport or Outport Block Parameters dialog box, use the **Signal Attributes** pane to configure wide signals and bus objects.

The following figure shows how to specify a wide signal, which corresponds to an AUTOSAR composite array.

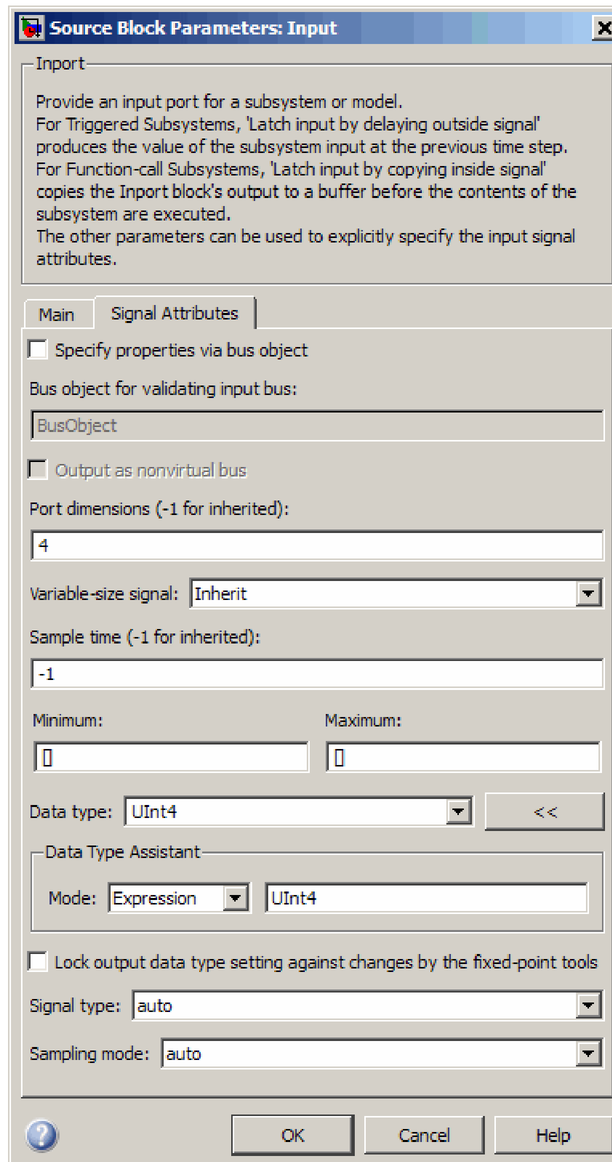


The following figure shows how to specify a bus object, which corresponds to an AUTOSAR composite record.



You can use the **Data Type Assistant** on the **Signal Attributes** pane of the Inport or Outport Block Parameters dialog box to specify the data types of data elements and arguments of an operation prototype. If you select **Mode** to be `Built in`, then you can specify the data type to be, for example, `single` or `boolean`. Alternatively, if you select **Mode** to be `Expression`, you can specify

an (alias) expression for data type. As an example, the following figure shows an alias `UInt4` in the **Data type** field.



Enumerated Data Types

AUTOSAR supports enumerated data types. For the import process, if there is a corresponding Simulink enumerated data type, then the software uses this data type. Through a check, the software ensures that the two data types are consistent. However, if there is no corresponding Simulink data type, then the software automatically creates the enumerated data type using the `Simulink.defineIntEnumType` class. This automatic creation of data types is useful when you want to import a large number of enumerated data types.

Consider the following example:

```
<SHORT-NAME>BasicColors</SHORT-NAME>
  <COMPU-INTERNAL-TO-PHYS>
  <COMPU-SCALES>
     <COMPU-SCALE>
        <LOWER-LIMIT>0</LOWER-LIMIT>
        <UPPER-LIMIT>0</UPPER-LIMIT>
        <COMPU-CONST>
           <VT>Red</VT>
....
```

The software creates an enumerated data type using:

```
Simulink.defineIntEnumType( 'BasicColors', ...
     {'Red', 'Green', 'Blue'}, ...
     [0;1;2], ...
     'Description', 'Type definition of BasicColors.', ...
     'HeaderFile', 'Rte_Type.h', ...
     'AddClassNameToEnumNames', false);
```

Per-Instance Memory

AUTOSAR supports per-instance memory, which allows you to specify instance-specific global memory within a software component. An AUTOSAR run-time environment generator allocates this memory and provides an API through which you access this memory.

In Simulink, you can model per-instance memory through the use of Data Store Memory and Data Store Read/Write blocks together with an `AUTOSAR.Signal` data object that specifies, for example, the `PerInstanceMemory` custom storage class.

AUTOSAR also allows you to use per-instance memory as a RAM mirror for data in non-volatile RAM (NVRAM), which enables you to access and use NVRAM in your AUTOSAR application.

Once an `AUTOSAR.Signal` data object specifies the `PerInstanceMemory` custom storage class, you can configure this per-instance memory to be a mirror block for a specific NVRAM block by setting the attribute `needsNVRAMAccess` to true.

For detailed information about how you model per-instance memory, see the demo rtwdemo_autosar_PIM_script. For an outline, see "Using Data Store Memory Blocks to Specify Per-Instance Memory" on page 24-55.

AUTOSAR Terminology

Term	Notes
AUTOSAR Runtime Environment (RTE)	• Layer between Application and Basic Software layers • Realizes communication between: - AUTOSAR Software Components - AUTOSAR Software Components and Basic Software
AUTOSAR Software Component	• A software component containing one or more algorithms, which communicates with its environment through ports • Connected to the AUTOSAR Runtime Environment (RTE) • Relocatable (not tied to a particular ECU)
Characteristics	Values of characteristics can be changed on an ECU through a calibration data management tool or an offline calibration tool.
Client-Server Interface	• PortInterface for client-server communication • Defines operations provided by server and used by client

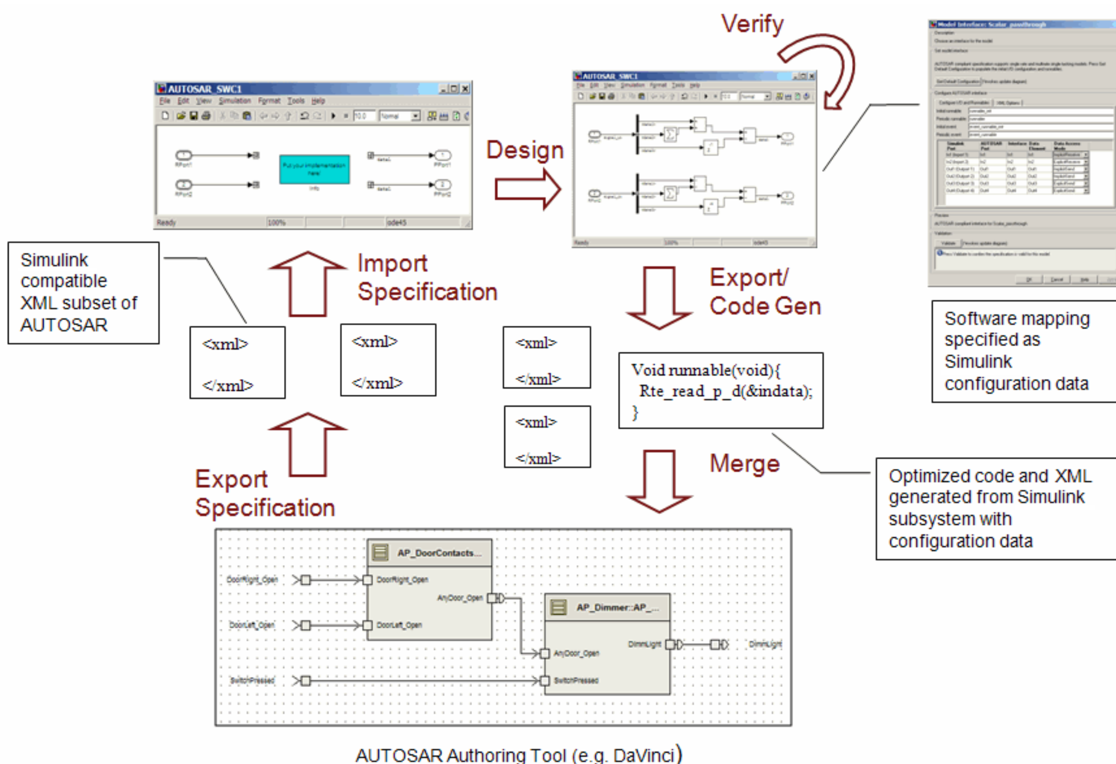
Term	Notes
Composite data types	Category of data types, such as one of the following: • Array — Contains more than one element of the same type, and has zero-based indexing • Record — Non-empty set of objects, where each object has a unique identifier
ComSpec	Defines specific communication attributes.
DataElementPrototype (data element)	Data value (signal) exchanged between a sender and a receiver.
Data types	• Either primitive or composite • Types data elements, arguments of operations in a Client-Server Interface, and constants
ErrorStatus	Indicates errors detected by communication system. Runtime Environment defines the following macros for sender-receiver communication: • `RTE_E_OK: no errors` • `RTE_E_INVALID: data element invalid` • `RTE_E_MAX_AGE_EXCEEDED: data element outdated`
OperationPrototype (operation)	• Invoked by a client • Provides value for each argument with direction `in` or `inout`, which must be of the correct data type • Client expects to receive a response to the invoked operation, part of which is a value (of correct data type) with direction `out` or `inout`
PortInterface	• Characterizes information provided or required by a port • Can be either Sender-Receiver Interface or Client-Server Interface

Term	Notes
Primitive data types	Category of data types that allow a direct mapping to C intrinsic types.
Provide port (PPort)	Port providing data or service of a server.
Require port (RPort)	Port requiring data or service of a server.
RTEEvent	Event or situation that triggers execution of a runnable by the Runtime Environment (RTE). The software supports the following RTEEvents: • OperationInvokedEvent (applicable to server operations) • TimingEvent • DataReceivedEvent
Runnable entity (runnable)	Part of AUTOSAR Software-Component that can be executed and scheduled independently of other runnable entities (runnables).
Sender-Receiver Interface	• PortInterface for sender-receiver communication • Defines data elements sent by sending component (with `Provide` port providing Sender-Receiver Interface) or received by receiving component (with `Require` requiring Sender-Receiver Interface)
Sender Receiver Annotation	Annotation of data elements in a port that implements Sender-Receiver Interface.
Sensor Actuator Software Component	AUTOSAR Software Component dedicated to the control of a sensor or actuator.
Service	Logical entity of Basic Software that offers functionality, which is used by various AUTOSAR Software Components.

Workflow for AUTOSAR

This section describes how you use Embedded Coder software to generate AUTOSAR-compliant code.

The following diagram shows a workflow that you can follow.



In this *round-trip* workflow, you perform the following tasks:

1. Import previously specified AUTOSAR Software Components, including definitions of calibration parameters, into Simulink. See "Importing an AUTOSAR Software Component" on page 24-28 and "Configuring Calibration Parameters" on page 24-53.

2. Incorporate your Simulink design into the skeleton model or subsystem created by the import process.

3. Export, generating code and description files. This process involves configuring the AUTOSAR interface, validating this interface, and then building your Simulink models. See:

   - "Using the Configure AUTOSAR Interface Dialog Box" on page 24-31
   - "Configuring Ports for Basic Software and Error Status Receivers" on page 24-37
   - "Configuring Client-Server Communication" on page 24-38
   - "Configuring AUTOSAR Options Programmatically" on page 24-64
   - "Modifying and Validating an Existing AUTOSAR Interface" on page 24-57
   - "Exporting AUTOSAR Software Component" on page 24-61
   - "Configuring Multiple Runnables" on page 24-47

   You can also verify your generated code in a simulation. See "Verifying the AUTOSAR Code with SIL and PIL Simulations" on page 24-65.

4. Merge generated code and description files with other systems using an AUTOSAR authoring tool, for example, the DaVinci tool suite from Vector Informatik GmbH. See demo rtwdemo_autosar_roundtrip_script.

   You can use the authoring tool to export specifications, which can be imported back into Simulink.

Importing an AUTOSAR Software Component

Use the `arxml.importer` class to:

- Parse an AUTOSAR Software Component description file, for example, exported from the DaVinci tool suite from Vector Informatik GmbH
- Import the software component into a Simulink model for configuration, code generation, and XML export

For a complete list of methods, see “AUTOSAR” in the Embedded Coder Function Reference documentation.

Use `arxml.importer` methods in the following order:

1. Call the constructor `arxml.importer`, for example, `arxml.importer('mySoftwareComponentFile.arxml')`, to create an importer object that looks for atomic software components in the specified XML file. In the Command Window, you see reports describing identified atomic software components. You can have multiple components. For example:

   ```
   The file "mySoftwareComponentFile.arxml" contains:
     1 Atomic-Software-Component-Type:
       '/ComponentType/complex_type_component'
     3 CalPrm-Component-Type:
       '/ComponentType/MyCalibComp1'
       '/ComponentType/MyCalibComp2'
       '/ComponentType/MyCalibComp3'
   ```

   To change the main file and update the list of components, use `arxml.importer.setFile`.

   Each software component requires an `arxml.importer` object. For each `arxml.importer` object, specify the file that contains the software component that you want.

2. Use `arxml.importer.setDependencies` if you need to specify additional dependent XML files containing the information that completes the software component description (for example, data types, interfaces). You can specify a cell array of files or a single file.

Complete specifying dependencies only for components that you intend to import into Simulink.

3. To import a parsed atomic software component into a Simulink model, call one of the following methods. If you have not specified all dependencies for the components, you will see errors.

   - `arxml.importer.createComponentAsSubsystem` — Creates and configures a Simulink subsystem skeleton corresponding to the specified atomic software component description.
   - `arxml.importer.createComponentAsModel` — Creates and configures a Simulink model skeleton corresponding to the specified atomic software component description.

     For example:

     ```
     importer_obj.createComponentAsModel('/ComponentType/complex_type_component')
     ```

   - `arxml.importer.createCalibrationComponentObjects` — Creates Simulink calibration objects corresponding to the specified AUTOSAR calibration component description.

     For example:

     ```
     [success] = createCalibrationComponentObjects(importer_obj,
     'CreateSimulinkObject', true)
     ```

   See also the limitation, “Cannot Import Internal Behavior” on page 24-68.

After you import your software component into Simulink, you can modify the skeleton model or subsystem. For parameters from a calibration component, after importing the parameters into the MATLAB workspace, assign the calibration parameters to block parameters in your model.

To configure AUTOSAR code generation options and XML export options, see:

- “Preparing a Simulink Model for AUTOSAR Code Generation” on page 24-31
- “Generating AUTOSAR Code and Description Files” on page 24-58
- “Configuring AUTOSAR Options Programmatically” on page 24-64

To see how to import, modify, and export AUTOSAR Software Components, view the demo Import and Export an AUTOSAR Software Component.

Preparing a Simulink Model for AUTOSAR Code Generation

In this section...

"Using the Configure AUTOSAR Interface Dialog Box" on page 24-31

"Configuring Ports for Basic Software and Error Status Receivers" on page 24-37

"Configuring Client-Server Communication" on page 24-38

"Configuring Multiple Runnables" on page 24-47

"Configuring Calibration Parameters" on page 24-53

"Using Data Store Memory Blocks to Specify Per-Instance Memory" on page 24-55

"Modifying and Validating an Existing AUTOSAR Interface" on page 24-57

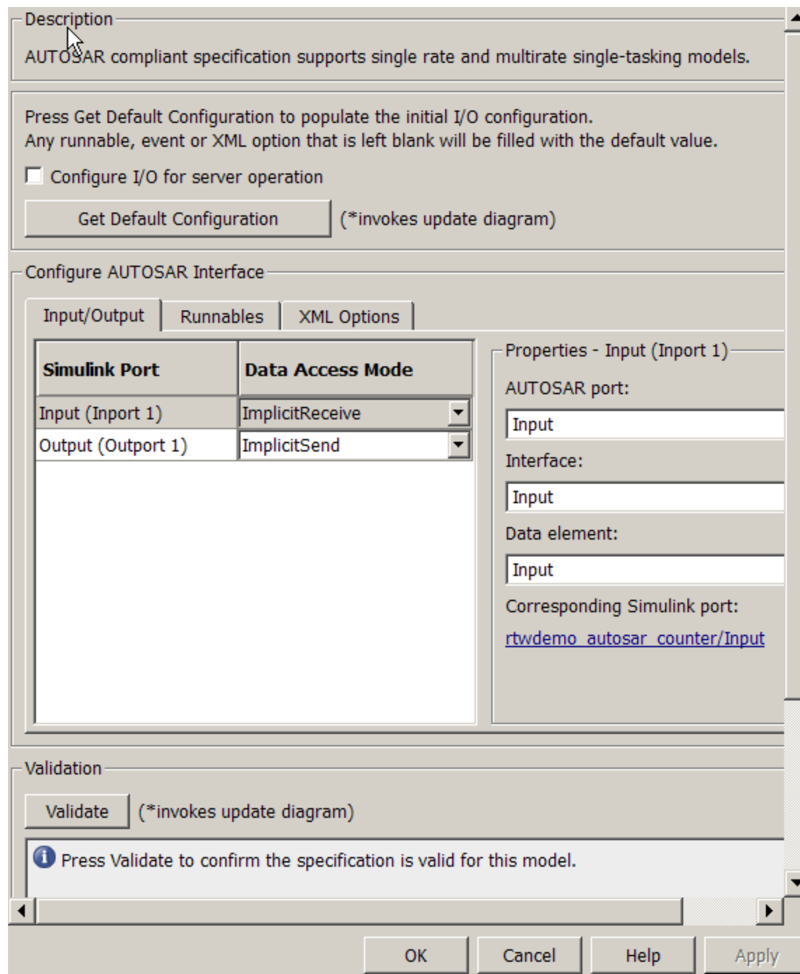
Using the Configure AUTOSAR Interface Dialog Box

Use the Configure AUTOSAR Interface dialog box to configure your AUTOSAR code generation and XML import and export options. Alternatively, you can control all AUTOSAR options programmatically. See "Configuring AUTOSAR Options Programmatically" on page 24-64.

In any model using the `autosar.tlc` system target file, you can open the Configure AUTOSAR Interface dialog box by right-clicking a subsystem and selecting **Code Generation > AUTOSAR *`Single or Multi`*-Runnable Component > Configure**.

Single-Runnable menu options are enabled for only atomic or function-call subsystems.

Multi-Runnable menu options are enabled for only virtual subsystems.



To configure your AUTOSAR options:

1 If the **Configure I/O for server operation** check box is selected, clear it. Select this check box only when you want to configure your Simulink model as a server operation (see "Configuring a Server Operation" on page 24-40).

2 Click **Get Default Configuration** to populate the controls for your model.

The runnable names, XML properties, and I/O configuration are initialized. If you click **Get Default Configuration** again later, only the I/O configurations are reset to default values.

3. Under **Configure AUTOSAR Interface**, use the controls to change your AUTOSAR code generation options and XML export options. For example, send and receive communication options such as port and interface names, data access modes, and initial and periodic runnable names.

   - On the **Input/Output** tab, designate inports and outports as data sender/receiver ports, error status receivers, or as access points to basic software.

     To designate inports and outports as sender or receiver ports, set each port's **Data Access Mode** to one of the following:

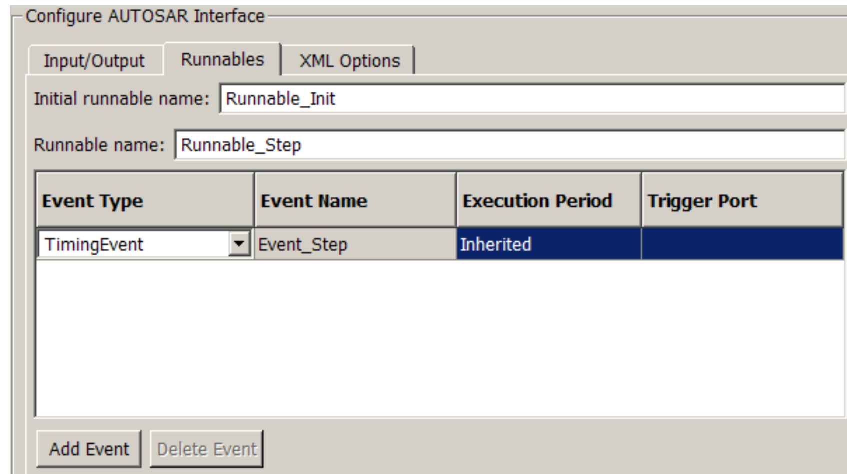
     - `Implicit` (recommended). Data is buffered by the Runtime Environment (RTE).
     - `Explicit`. Data is not buffered.

     Use the port interface settings to reflect your AUTOSAR port best practices. For example, some AUTOSAR users like to group related data into the same AUTOSAR port. You can achieve this arrangement in the GUI by duplicating AUTOSAR port names. Alternatively, you can use the AUTOSAR port to group information individually. In this case, a common approach is to set all of the data element settings to something neutral, for example, `'data'`, and leave the AUTOSAR port names as they are. You can also use the AUTOSAR interface name for any best practices that you might have. For example, you can set up interfaces for individual AUTOSAR ports by ensuring that the interface names change when the AUTOSAR port name changes, for example, by prefixing the AUTOSAR interface of the corresponding AUTOSAR port name with an `'if_'`.

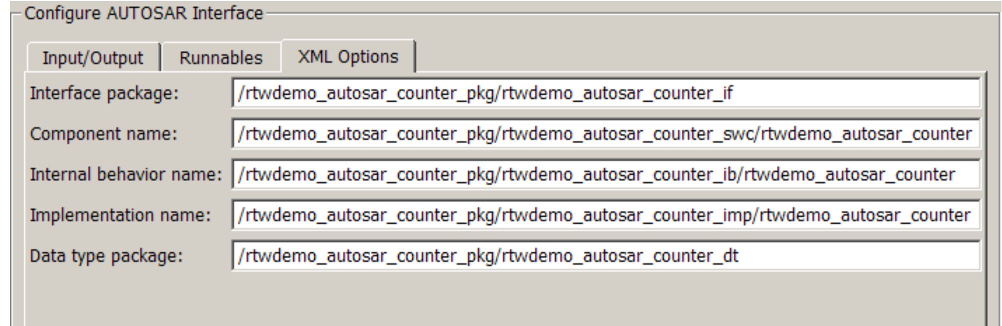
     For more information on all these options, see "AUTOSAR" in the Embedded Coder Function Reference documentation.

     You also use **Data Access Mode** to designate ports to access basic software or error status. See "Configuring Ports for Basic Software and Error Status Receivers" on page 24-37.

   - On the **Runnables** tab, specify the names of your initial and periodic runnables, for example, `Runnable_Init` and `Runnable_Step`.



- On the **XML Options** tab, specify the names and package paths of the XML files that you publish when you generate code. See also "Exporting AUTOSAR Software Component" on page 24-61.



4. After you configure your options, click **Validate**, which calls `runValidation`. If there are problems, you see messages describing why the configuration is invalid.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Embedded Coder Function Reference documentation.

---

5 If validation succeeds, click **OK** to return to the Configuration Parameters dialog box.

6 Save your model and then generate code to export your AUTOSAR component.

Configuring Single Runnables for DataReceivedEvents

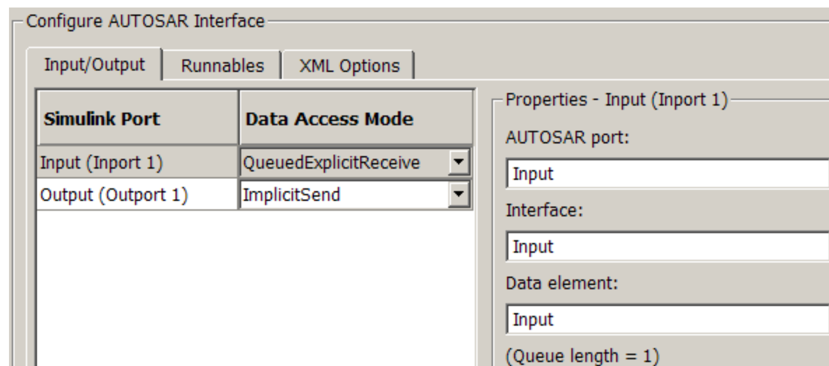
The AUTOSAR Runtime Environment uses the event type *DataReceivedEvent* to trigger a runnable only when the value of a received data element is updated.

The software supports two data access modes that enable DataReceivedEvents to act as triggers, `ExplicitReceive` and `QueuedExplicitReceive`. The latter, in principle, allows the queuing of events. However, by default, the software restricts the queue length to one event only. If you want a different queue length, you must edit the generated XML file.

To create a runnable trigger with a DataReceivedEvent:

1 Under **Configure AUTOSAR Interface**, select the **Input/Output** tab.

2 If you want an input data signal to be a trigger (for example, `Input`) set **Data Access Mode** for the corresponding inport to `ExplicitReceive` or `QueuedExplicitReceive`.



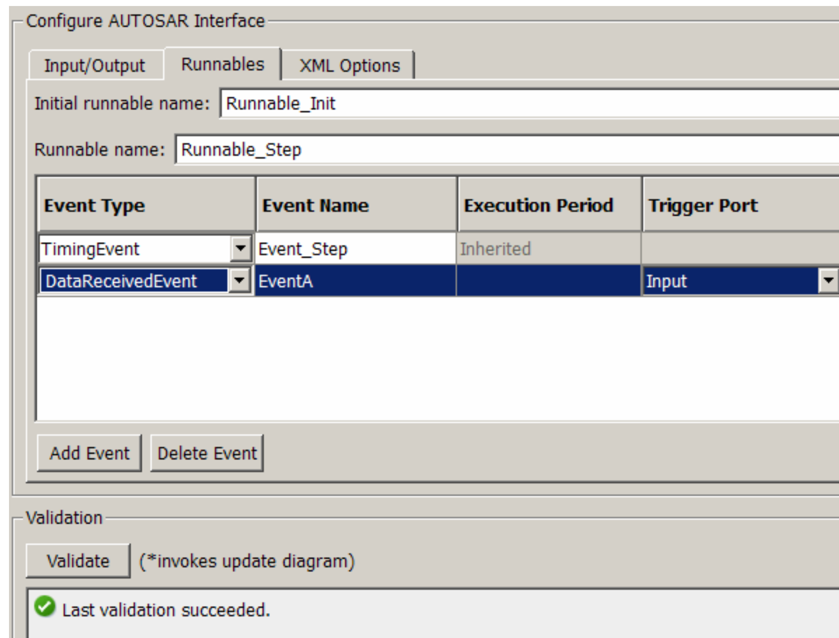
3 Select the **Runnables** tab. To create a new trigger event, click **Add Event**. By default, from the **Event Type** drop-down list, the software selects `DataReceivedEvent`.

4 In the **Event Name** column, specify an appropriate event name.

5 In the **Trigger Port** column, from the drop-down list, select the Simulink port, for example, `Input`.

6 To create an additional trigger event, repeat steps 3 – 5. You can remove a trigger event by selecting the event row and clicking **Delete Event**.

7 To verify that you have configured the event triggers correctly, click **Validate**.



Note If you define a DataReceivedEvent in a top-model or right-click build configuration, MathWorks recommends that you specify sample time independance for the model, that is, you set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane to `Ensure sample time independent`. This action ensures that the generated code can be executed at non-periodic rates, for example, asynchronously. However, if you know the execution context, for example, the data triggers periodically, then you do not have to specify sample time independence.

Configuring Ports for Basic Software and Error Status Receivers

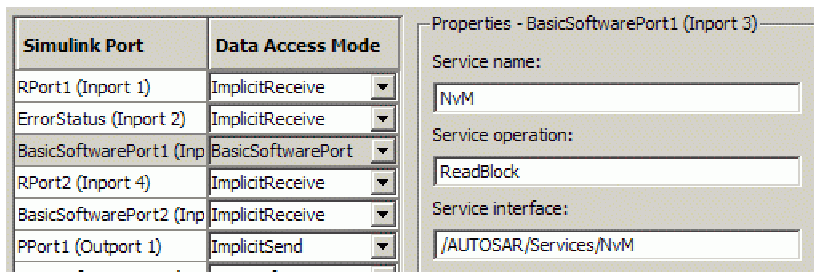
You can configure ports to access AUTOSAR services and device drivers (AUTOSAR basic software), and to access communication error status in your model. You can configure ports programmatically or by using the AUTOSAR Model Interface dialog box. To open the dialog box, right-click a subsystem

and select **Code Generation > AUTOSAR *`Single or Multi`*runnable Component > Configure**.

In the dialog box, you can specify the **Data Access Mode** of every port.

- Designate inports and outports as access points to basic software.

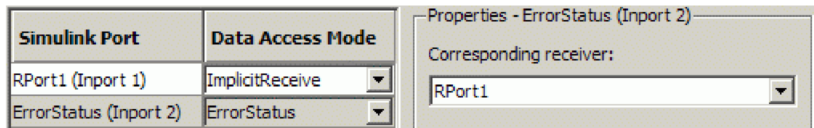
  If you select **Basic Software**, specify the service name, operation, and interface. The service name and operation must be valid AUTOSAR identifiers, and the service interface must be a valid path of the form `AUTOSAR/Service/`*`servicename`*.

  After you export your AUTOSAR components, you must include your service interface definition XML file to import these components correctly into an authoring tool.

- Designate inports to receive error status.

  If you select **Error Status** for an inport, you must select the other port (of mode Implicit or Explicit Receive) to listen for error status. Error status ports must use `uint8` data type (or an alias).

Configuring Client-Server Communication

- "Configuring a Server Operation" on page 24-40
- "Configuring the Invoke AUTOSAR Server Operation Block" on page 24-43

- “Creating Configurable Subsystems from a Client-Server Interface” on page 24-45
- “Simulating and Generating Code for Client-Server Communication” on page 24-46

AUTOSAR allows client-server communication between:

- Application software components
- An application software component and Basic Software

An AUTOSAR Client-Server Interface defines the interaction between a software component that *provides* the interface and a software component that *requires* the interface. The component that provides the interface is the server. The component that requires the interface is the client.

In Simulink, you can:

- Configure your model to implement a server operation. When you build your model, you generate AUTOSAR-compliant code and XML description files, including a client-server interface. See “Configuring a Server Operation” on page 24-40.
- Configure a client port for your model using an Invoke AUTOSAR Server Operation block that references a client-server interface. When you build your model, you generate AUTOSAR-compliant code and XML description files for your client port. See “Configuring the Invoke AUTOSAR Server Operation Block” on page 24-43.

Once you create a client-server interface, you can generate a Simulink library of configurable, client-server subsystems that reference the:

- Invoke AUTOSAR Server Operation block for code generation
- Server operation model block for simulation

For information on how to generate this library, see “Creating Configurable Subsystems from a Client-Server Interface” on page 24-45

You can deploy the client-server subsystem in a Simulink model and, using the Mode Switch for Invoke AUTOSAR Server Operation, run the model in

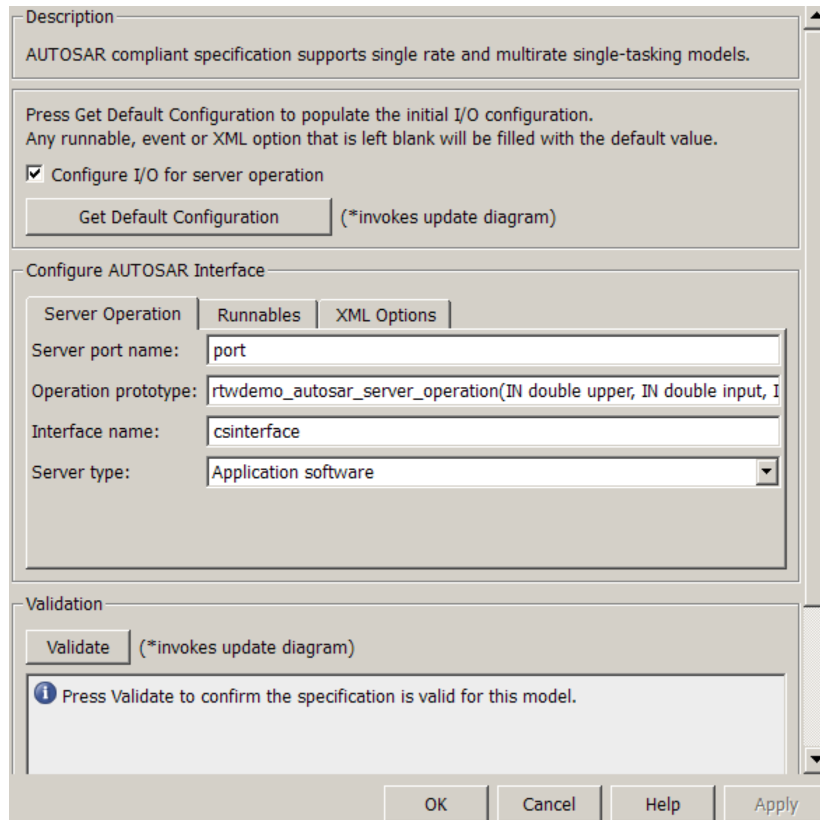
either a simulation or code-generation mode. See "Simulating and Generating Code for Client-Server Communication" on page 24-46.

For a demo on generating and using an AUTOSAR Client-Server Interface, see rtwdemo_autosar_clientserver_script.

Configuring a Server Operation

In the Configure AUTOSAR Interface dialog box, you can configure your Simulink model as a server operation. Then you can generate AUTOSAR-compliant code and XML files, including the client-server interface.

1 Select the **Configure I/O for server operation** check box. The **Input/Output** tab becomes the **Server Operation** tab.



2 Click **Get Default Configuration** to populate the controls for your model.

The runnable names, XML properties, and I/O configuration are initialized. If you click **Get Default Configuration** again later, only the I/O configurations are reset to default values.

On the **Configure AUTOSAR Interface** pane, use the controls to change your AUTOSAR code generation options and XML export options.

3 On the **Server Operation** tab, specify the following:

- **Server port name**. Use a valid AUTOSAR short-name identifier.
- **Operation prototype** . The names of the prototype and its arguments must be valid AUTOSAR short-name identifiers, for example

`rtwdemo_autosar_server_operation(IN double upper, IN double input, IN double lower, OUT double output)`.

- **Interface name**. The path reference of the client-server interface. Use a valid AUTOSAR short-name path, for example, `csinterface`.
- **Server type**. From the drop-down list, select either `Application software` or `Basic software`.

4 On the **Runnables** tab, specify the names of your initial and periodic runnables, for example, `Runnable_Init` and `Runnable_Step`.

5 On the **XML Options** tab, specify the names and package paths of the XML files that you publish when you generate code. For more details about these files, see “Exporting AUTOSAR Software Component” on page 24-61.

6 After you configure your options, click **Validate**, which calls `runValidation`. If there are problems, you see messages describing why the configuration is invalid.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Embedded Coder Function Reference documentation.

7. If validation succeeds, click **OK** to return to the Configuration Parameters dialog box.

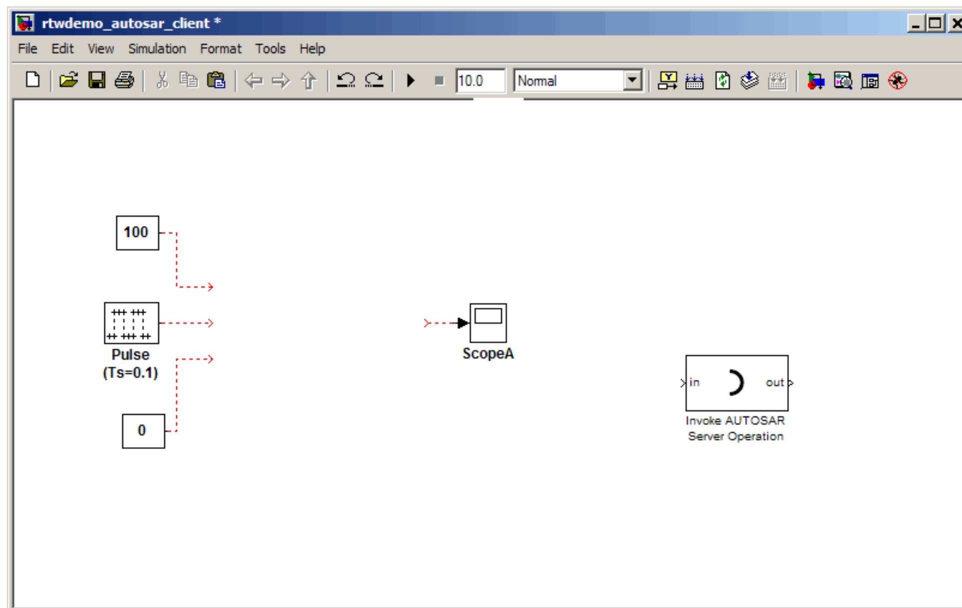
8. Save your model.

9. To generate AUTOSAR-compliant code and XML files, select **Tools > Code Generation > Build Model**.

Configuring the Invoke AUTOSAR Server Operation Block

You can use the Invoke AUTOSAR Server Operation block in your Simulink model to configure a client port (that accesses either application software or AUTOSAR Basic Software). You can then build the model to generate AUTOSAR-compliant code and XML files.

1. Drag an Invoke AUTOSAR Server Operation block into your model.

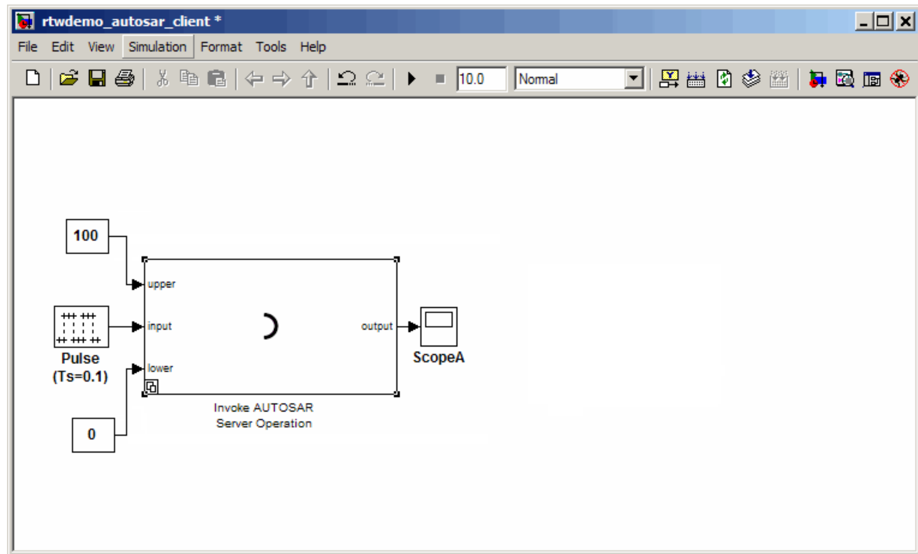


2 Double-click the block to open the Invoke AUTOSAR Server Operation dialog box. Specify the following:

- **Client port name**. A valid AUTOSAR short-name identifier.
- **Operation prototype**. The names of the prototype and its arguments must be valid AUTOSAR short-name identifiers, for example, `rtwdemo_autosar_server_operation(IN double upper, IN double input, IN double lower, OUT double output)`.
- **Interface path**. The path reference of the client-server interface. You must use a valid AUTOSAR short-name path, for example, `/AUTOSAR/Interface`.
- **Server type**. From the drop-down list, select either `Application software` or `Basic software`.
- **Show error status**. If you want the client port to receive the error status of client-server communication, select this check box.
- **Sample time**. Set this parameter to -1 to inherit the sample time.

3 Click **OK**. Your Invoke AUTOSAR Server Operation block is updated.

4 Connect the updated Invoke AUTOSAR Server Operation block to your model.



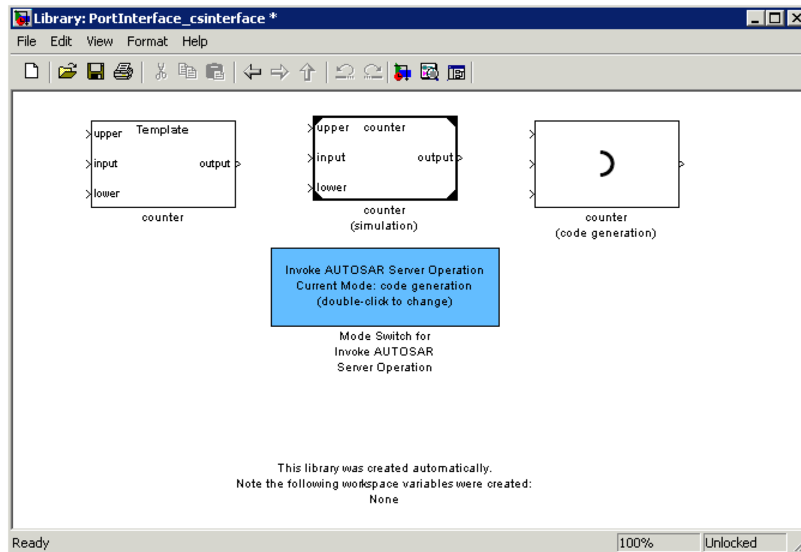
5 Select **Tools > Code Generation > Build Model**. AUTOSAR-compliant code and XML files for the client port are generated.

Creating Configurable Subsystems from a Client-Server Interface

You can generate a Simulink library of configurable subsystems by applying the `createOperationAsConfigurableSubsystems` method to the `arxml.importer` object with the client-server interface. For example:

```
% Create an AUTOSAR importer object
obj = arxml.importer('rtwdemo_autosar_csinterface.arxml');

% Create the client-server operation configurable subsystem library
obj.createOperationAsConfigurableSubsystems('/PortInterface/csinterface', ...
                                            'CreateSimulinkObject', false);
```

yield the following `PortInterface_csinterface` library.

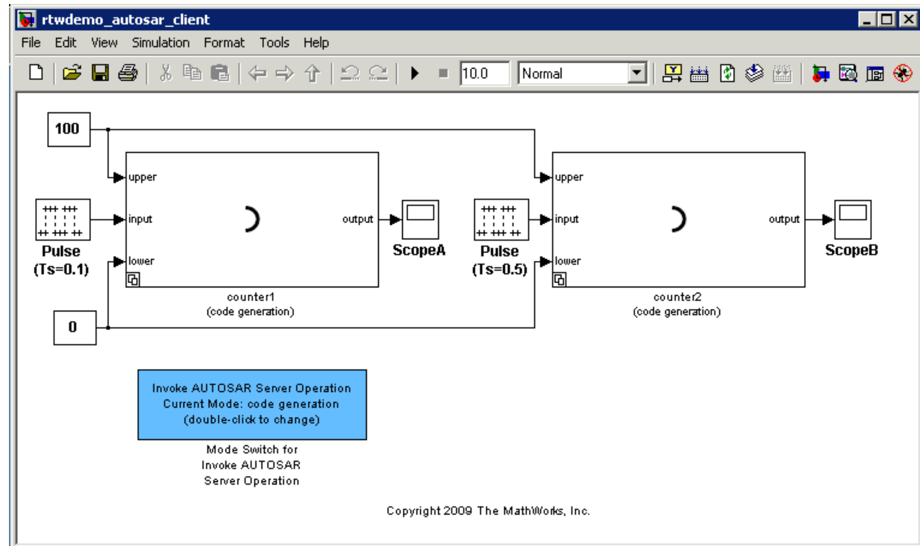


Simulating and Generating Code for Client-Server Communication

Use the Template block from the client-server subsystem library to construct a model that can be run in either code-generation or simulation mode.

1. Drag the Template block from the subsystem library into your model window and connect it to other blocks.

2. Place the Mode Switch for Invoke AUTOSAR Server Operation in your model window.



To simulate the model:

1 Double-click the Mode Switch for AUTOSAR Server Operation to change the current mode from `code generation` to `simulation`.

2 Select **Simulation > Start**.

To generate code for the model:

1 Double-click the Mode Switch for AUTOSAR Server Operation to change the current mode from `simulation` to `code generation`.

2 Select **Tools > Code Generation > Build Model**.

Configuring Multiple Runnables

You can use function-call subsystems within a wrapper subsystem to represent multiple runnables in a single AUTOSAR Software Component, and export each function-call subsystem as an AUTOSAR runnable.

If you group function-call subsystems within your wrapper subsystem into virtual subsystems, for example, to improve the graphical layout of your

model, you can still export the function-call subsystems as AUTOSAR runnables. For information about virtual subsystems, see "Creating Subsystems" and "Virtual Blocks" in the Simulink documentation.

In addition to function-call subsystems, the software supports the following blocks within a wrapper subsystem:

- Data Store
- Display
- DocBlock
- From
- Goto
- Merge
- Model Info
- Scope
- Signal Specification

Use the Configure AUTOSAR Interface dialog box to specify an AUTOSAR interface for each function-call subsystem that you want to export as a runnable. To open this dialog box, right-click the top-level wrapper subsystem and select **Code Generation > AUTOSAR Multi-Runnable Component > Configure**. For information on how you configure multiple runnables, see:

- "Using the Configure AUTOSAR Interface Dialog Box" on page 24-31
- "Configuring Inter-Runnable Variables" on page 24-48
- "Specifying Execution Period" on page 24-50
- "Configuring Multiple Runnables for DataReceivedEvents" on page 24-51

See also the AUTOSAR Code Generation for Multiple Runnable Entities demo.

Configuring Inter-Runnable Variables

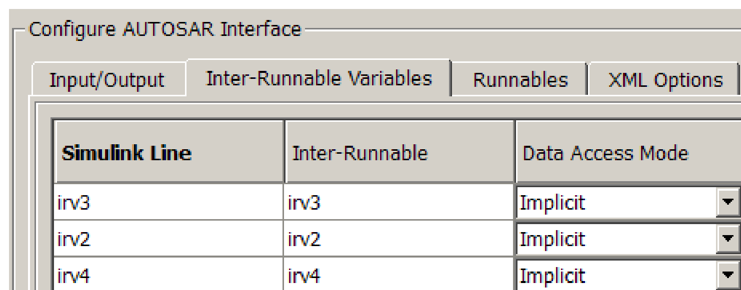
Inter-runnable variables communicate primitive type data between runnables in a component. You define these inter-runnable variables by the signal lines

that connect subsystems. For an example, see "Inter-Runnable Variables" on page 24-16.

By default, the software assigns the signal name to the exported inter-runnable variable. If you want to edit the name, before generating code, double-click the signal name and enter a new name. However, you can specify a different name for the exported variable. In addition, you can specify the data access mode of the inter-runnable variable.

To configure an inter-runnable variable:

1 In the Configure AUTOSAR Interface dialog box, select the **Inter-Runnable Variables** tab. You see Simulink signals with (default) inter-runnable names and data access modes.



2 For each signal that you want to configure:

   a In the **Inter-Runnable** cell, specify your AUTOSAR name for the exported variable.

   b In the **Data Access Mode** cell, from the drop-down list, select either `Explicit` or `Implicit` (recommended).

3 Click **OK**.

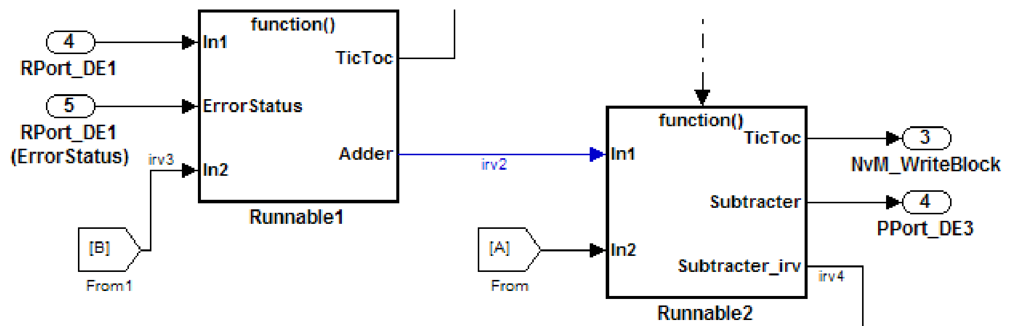
When you select a signal, links appear under **Source ports for signal** .

Simulink Line	Inter-Runnable	Data Access Mode
irv3	irv3	Implicit
irv2	irv2	Implicit
irv4	irv4	Implicit

Source ports for signal irv2

rtwdemo_autosar_multirunnables/ASWC/Runnable1/2

Each link corresponds to an instance of the signal and is associated with a source port in the Simulink model. Click a link to go to the corresponding model signal. For example, clicking `rtw.../ASWC/Runnable1/2` takes you to the following.



Specifying Execution Period

You may need to use blocks that depend on time for a software component with multiple runnables, for example, the Discrete-Time Integrator block. In this case, you can specify a timer for each AUTOSAR runnable. The timer increments at each execution of the runnable.

Use the Configure AUTOSAR Interface dialog box to specify the execution period:

1. Select the **Runnables** tab.

2. Under **Runnable**, select a runnable, for example, `Runnable1`.

3. In the **Execution Period** cell for the runnable's TimingEvent, enter the execution period.

4. Click **OK**.

Note The timer value in an AUTOSAR runnable is valid only if the runnable runs at a periodic rate that corresponds to the execution period that you specify. If the runnable runs at different rate, or does not begin executing at t = 0, then the timer value will be incorrect.

The timer data type generated depends on the execution period and the application life span. To specify the application life span:

1. Select **Configuration Parameters > Optimization**.

2. In the **Application lifespan (days)** field, enter the required value.

Configuring Multiple Runnables for DataReceivedEvents

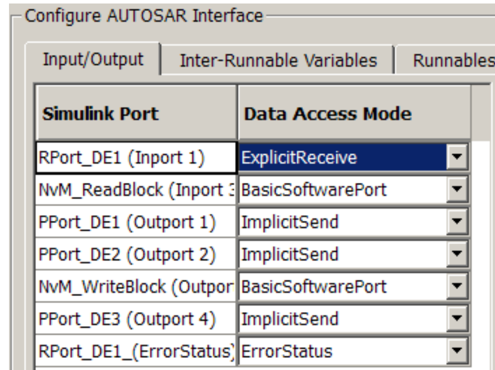
The AUTOSAR Runtime Environment uses the event type DataReceivedEvent to trigger runnables only when the value of a received data element is updated.

The software supports two data access modes that enable DataReceivedEvents to act as triggers, `ExplicitReceive` and `QueuedExplicitReceive`. The latter, in principle, allows the queuing of events. However, by default, the software restricts the queue length to one event only. If you want a different queue length, you must edit the generated XML file.

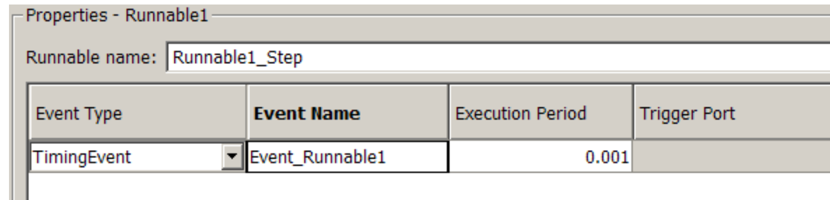
To create runnable triggers with DataReceivedEvents:

1. Under **Configure AUTOSAR Interface**, select the **Input/Output** tab.

2. If you want an input data signal to be a trigger, for example, `RPort_DE1`, from the corresponding **Data Access Mode** drop-down list, select either `ExplicitReceive` or `QueuedExplicitReceive`.



3 Select the **Runnables** tab. Under **Runnable**, select the runnable that you want to configure, for example, `Runnable1`.



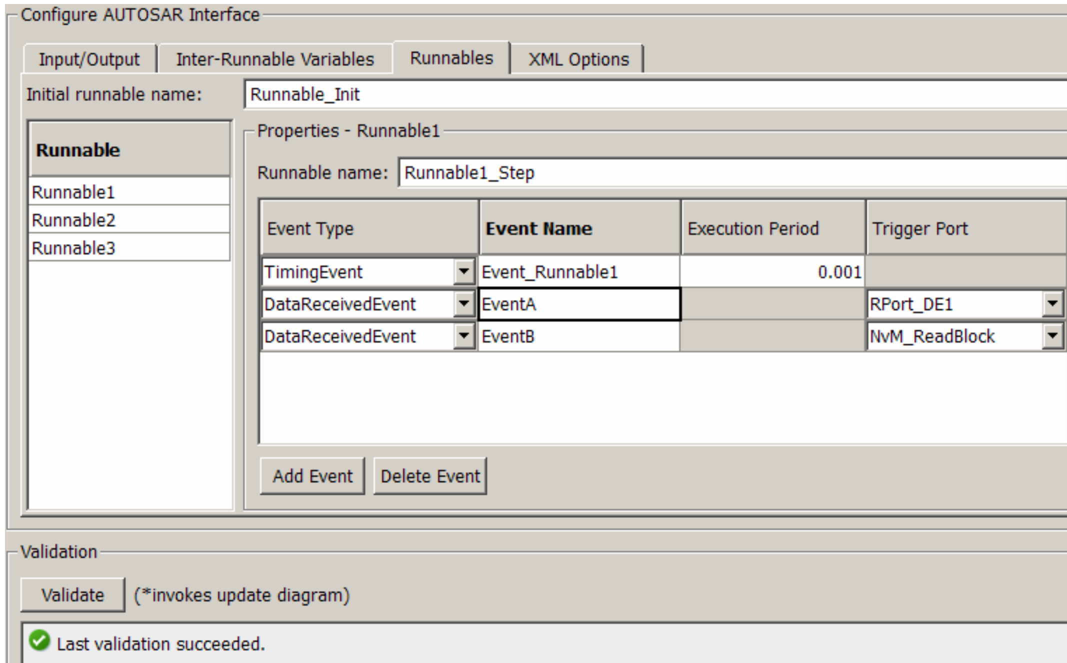
4 Click **Add Event** to create a new trigger event. By default, from the **Event Type** drop-down list, the software selects `DataReceivedEvent` .

5 In the **Event Name** column, specify an appropriate event name.

6 In the **Trigger Port** column, from the drop-down list, select the Simulink port, for example, `RPort_DE1`.

7 To create an additional trigger event, for example, using `NvM_ReadBlock`, repeat steps 4 – 6. You can remove a trigger event by selecting the event row and clicking **Delete Event**.

8 To verify that you have configured the trigger events correctly, click **Validate**.



Note If a runnable contains blocks that use absolute time, for example, a discrete-time integrator, then MathWorks recommends that you:

- Use a timing event to trigger the runnable
- Specify the execution period of the timing event to be the same as the sample time of the function-call trigger.

Configuring Calibration Parameters

You can specify the type of calibration parameter that you export by configuring properties of the corresponding block parameter in the base workspace.

For example, to configure an *internal calibration parameter* for your AUTOSAR model:

1 Create an `AUTOSAR.Parameter` object.

   a Open the Model Explorer (**Ctrl+H**).

   b In the **Model Heirarchy** view, under **Simulink Root**, select **Base Workspace**.

   c Select **Add > Add Custom**. The Model Explorer – Select Object dialog box opens.

   d Specify a value in the **Object Name(s)** field, for example, `myPrm`.

   e From the **Object class** drop-down list, select `AUTOSAR.Parameter`.

   f Click **OK**. A new object `myPrm` appears in the base workspace.

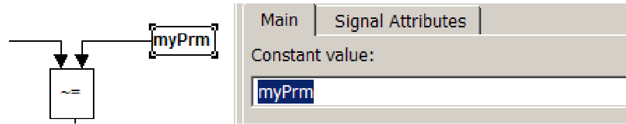
2 In the **Contents** pane, select the object, for example, `myPrm`.

3 Using the **Dialog** pane, configure the following properties of this data object:

   - **Value** — Specify a value for the calibration parameter. For an internal calibration parameter, this value represents the initial value.
   - **Data type**. For information about a creating data type, for example, a bus object data type, see "Using the Data Type Assistant" in the Simulink documentation.
   - **Storage class** — To specify an internal calibration parameter, from the drop-down list, select `InternalCalPrm`. You must then specify **Per instance behavior**. Select one of the following:
     - `Parameter shared by all instances of the Software Component`
     - `Each instance of the Software Component has its own copy of the parameter`

   For information about the **Dialog** pane, see "The Model Explorer: Dialog Pane" in the Simulink documentation.

4 In the Block Parameters dialog box, assign the data object to your model, for example:



Before you generate code, you must:

- Select **Configuration Parameters > Optimization > Signals and Parameters > Inline parameters**.
- Clear **Configuration Parameters > Code Generation > Ignore custom storage classes**.

These actions ensure that the software exports calibration parameters correctly. See "Generating Code with Custom Storage Classes" on page 8-58.

For calibration component parameters, after you export your AUTOSAR components, you must include your calibration interface definition XML file to import the parameters correctly into an authoring tool.

Note The software does not support the use of AUTOSAR calibration parameters within Model blocks.

Using Data Store Memory Blocks to Specify Per-Instance Memory

You can model per-instance memory through the use of Data Store Memory blocks together with an `AUTOSAR.Signal` data object. For a detailed example, see the `rtwdemo_autosar_PIM_script` demo. The following is an outline of the required steps:

1. In the base workspace, create an `AUTOSAR.Signal` object.
2. Set the storage class of this object to `PerInstanceMemory`.
3. If required, set `needsNVRAMAccess` property to true.
4. Create a Data Store Memory block that references the `AUTOSAR.Signal` object. See Data Store Memory in the Simulink Reference documentation.

Note The software does *not* support per-instance memory modeling within a submodel.

When you build your model, the XML files that are generated define an exclusive area for each Data Store Memory block that references per-instance memory. Every runnable that accesses per-instance memory runs inside the corresponding exclusive area. If multiple AUTOSAR runnables have access to the same Data Store Memory block, the exported AUTOSAR specification enforces data consistency by using an AUTOSAR exclusive area. This specification ensures that runnables have mutually exclusive access to the per-instance memory global data, preventing data corruption.

If you set `needsNVRAMAccess` to true, then a `SERVICE-NEEDS` entry (schema version 3.0) or `NVRAM-MAPPINGS` entry (schema version 2.1) is declared in XML files to indicate that the per-instance memory is a RAM mirror block and must be serviced by the `NvM` manager module.

Creating an `AUTOSAR.Signal` Object

To create an `AUTOSAR.Signal` object in the base workspace:

1. Open the Model Explorer (**Ctrl+H**).
2. In the **Model Heirarchy** view, under **Simulink Root**, select **Base Workspace**.
3. Select **Add > Add Custom**. The Model Explorer – Select Object dialog box opens.
4. Specify a value in the **Object Name(s)** field, for example, `nvmImplicitRW`.
5. From the **Object class** drop-down list, select `AUTOSAR.Signal`.
6. Click **OK**. A new object `nvmImplicitRW` appears in the base workspace.

Modifying and Validating an Existing AUTOSAR Interface

You can validate your AUTOSAR interface using the Configure AUTOSAR Interface dialog box. See "Using the Configure AUTOSAR Interface Dialog Box" on page 24-31. The following steps show how you can modify and validate your AUTOSAR interface *programmatically*:

1 Get the handle to an existing model-specific `RTW.AutosarInterface` object that is attached to your loaded Simulink model. Enter:

```
obj = RTW.getFunctionSpecification(modelName)
```

`modelName` is a string specifying the name of a loaded Simulink model, and *`obj`* returns a handle to an `RTW.AutosarInterface` object attached to the specified model.

Test the AUTOSAR interface object. Enter:

```
isa(obj,'RTW.AutosarInterface')
```

This test must return 1. If the model does not have an AUTOSAR interface object, the function returns `[]`.

2 To view and change items, use the AUTOSAR `get` and `set` functions listed in "AUTOSAR" in the Embedded Coder Function Reference documentation.

3 Validate the function prototype using `RTW.AutosarInterface.runValidation`.

Note For information on all validation checks, see `RTW.AutosarInterface.runValidation` in the Embedded Coder Function Reference documentation.

---

4 If validation succeeds, save your model and then generate code.

Generating AUTOSAR Code and Description Files

In this section...

"Selecting an AUTOSAR Schema" on page 24-58

"Specifying Maximum SHORT-NAME Length" on page 24-58

"Configuring AUTOSAR Compiler Abstraction Macros" on page 24-59

"Root-Level Matrix I/O" on page 24-61

"Exporting AUTOSAR Software Component" on page 24-61

Selecting an AUTOSAR Schema

The default AUTOSAR schema version is 3.1. If you need to change the schema version, you must do so before exporting your AUTOSAR Software Component. Embedded Coder supports the following AUTOSAR schema versions:

- 3.1 (3.1.4 r65277)
- 3.0 (3.0.2 r21527)
- 2.0 (XSD build 49632)
- 2.1 (XSD rev 0017)

To select a schema version, open the Configuration Parameters dialog box:

1. In any model using the `autosar.tlc` system target file, the **AUTOSAR Code Generation Options** component appears in the tree.

   Click **AUTOSAR Code Generation Options** to open the **AUTOSAR Code Generation Options** pane.

2. From the drop-down list for **Generate XML file for schema version**, select the schema version that you require.

Specifying Maximum SHORT-NAME Length

The AUTOSAR standard specifies that `SHORT-NAME` XML elements must not be greater than 32 characters in length. However, your authoring tool may

support the use of longer elements, for example, to name ports and interfaces. The software allows you to specify the maximum length of your `SHORT-NAME` elements.

Before you build your model, on the **Code Generation > AUTOSAR Code Generation Options**pane, in the **Maximum SHORT-NAME length** field, specify the maximum length of your `SHORT-NAME` elements. You may specify a maximum length of up to 128 characters. The default is 32 characters.

Configuring AUTOSAR Compiler Abstraction Macros

Compilers for 16-bit platforms (for example, Cosmic and Metrowerks for S12X or Tasking for ST10) use special keywords to deal with the limited 16-bit addressing range. The location of data and code beyond the 64 k border is selected explicitly by special keywords. However, if such keywords are used directly within the source code, then software must be ported separately for each microcontroller family, that is, the software is not platform-independent.

AUTOSAR specifies C macros to abstract compiler directives (near/far memory calls) in a platform-independent manner. These compiler directives, derived from the 16-bit platforms, enable better code efficiencies for 16-bit micro-controllers without separate porting of source code for each compiler. This approach allows your system integrator, rather than your software component implementer, to choose the location of data and code for each software component.

For more information on AUTOSAR compiler abstraction, see www.autosar.org

Configuring AUTOSAR Compiler Macro Generation

Before you build your model, on the Simulink Coder **AUTOSAR Code Generation Options** pane, select the **Use AUTOSAR compiler abstraction macros** check box.

When you build the model, the software applies compiler abstraction macros to global data and function definitions in the generated code.

For data, the macros are in the following form:

- `CONST(`*`consttype`*`, `*`memclass`*`)` *`varname`*;
- `VAR(`*`type`*`, `*`memclass`*`)` *`varname`*;

where

- *`consttype`* and *`type`* are data types
- *`memclass`* is a macro string *`SWC`*`_VAR` (*`SWC`* is the software component identifier)
- *`varname`* is the variable identifier

For functions (model and subsystem), the macros are in the following form:

- `FUNC(`*`type`*`, `*`memclass`*`) funcname(void)`

where

- *`type`* is the data type of the return argument
- *`memclass`* is a macro string. This string can be either *`SWC`*`_CODE` for runnables (external functions), or *`SWC`*`_CODE_LOCAL` for internal functions (*`SWC`* is the software component identifier).

Example

If you do *not* select the **Use AUTOSAR compiler abstraction macros** check box, the software generates the following code:

```
/* Block signals (auto storage) */
BlockIO rtB;

/* Block states (auto storage) */
D_Work rtDWork;

/* Model step function */
void Runnable_Step(void)
```

However, if you select the **Use AUTOSAR compiler abstraction macros** check box, the software generates macros in the code:

```
/* Block signals (auto storage) */
```

```
VAR(BlockIO, SWC1_VAR) rtB;

/* Block states (auto storage) */
VAR(D_Work, SWC1_VAR) rtDWork;

/* Model step function */
FUNC(void, SWC1_CODE) Runnable_Step(void)
```

Root-Level Matrix I/O

The software supports matrix I/O at the root-level by generating code that implements matrices as one-dimensional arrays. However, this behavior is not the default. Before you build your model, on the **AUTOSAR Code Generation Options** pane, select the **Support root-level matrix I/O using one-dimensional arrays** check box.

Exporting AUTOSAR Software Component

After configuring your AUTOSAR export options, generate code to export your AUTOSAR Software Component.

To generate code and XML files:

- For a single runnable from a top model, build the model (**Ctrl+B**).
- For multiple runnables from subsystems:
  1. Right-click the top-level subsystem.
  2. Select **Code Generation > AUTOSAR Multi-Runnable Component > Export Functions**. The Build Code for Subsystem dialog box opens.
  3. Click **Build**.

     This command builds code for an AUTOSAR runnable for each subsystem. The build also creates an additional runnable to aggregate the initialization functions for each of the function-call subsystems.

The software component C code and the following XML files are exported to the build folder.

File Name	Description
*modelname*_behavior.arxml	Specifies the software component internal behavior
*modelname*_implementation.arxml	Specifies the software component implementation
*modelname*_interface.arxml	Specifies the software component interfaces, including extra interfaces
*modelname*_component.arxml	Specifies the software component type, including additional ports added to the Simulink model
*modelname*_datatype.arxml	Specifies the software component data types, including any modified or additional data types

Note In addition to the AUTOSAR software component C code, Embedded Coder creates the following header files in the `stub` subfolder of the build folder:

- `Rte_Type.h`
- `Rte_`*`SWC`*`.h`, where *SWC* is the name of the software component
- `Compiler.h`

These files contain dummy implementations of AUTOSAR functions, which the software uses for SIL and PIL simulations. You must *not* use these files outside Simulink. Your AUTOSAR RTE generator should produce the equivalent files.

You can merge the software component information back into an AUTOSAR authoring tool. This software component information is partitioned into separate files to facilitate merging. The partitioning attempts to minimize the number of merges that you must do. You do not need to merge the data type file into the authoring tool because data types are usually defined early in the design process. You must, however, merge the internal behavior file because this information is part of the model implementation.

For examples of how to generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model, see the following demos.

- AUTOSAR Code Generation
- AUTOSAR Code Generation for Multiple Runnable Entities

Configuring AUTOSAR Options Programmatically

To control AUTOSAR options programmatically, use the AUTOSAR functions listed in the following tables in the Embedded Coder Function Reference documentation.

- “AUTOSAR Component Import”
- “AUTOSAR Configuration”

Verifying the AUTOSAR Code with SIL and PIL Simulations

In this section...

"Overview" on page 24-65

"Using the SIL and PIL Simulation Modes" on page 24-65

"Using a SIL or PIL Block for AUTOSAR Verification" on page 24-66

Overview

You can carry out model-based verification of AUTOSAR software components using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. Use SIL for verification of generated source code on your host computer, and PIL for verification of object code on your production target.

Using the SIL and PIL Simulation Modes

You can run a top model or Model block that is configured for the AUTOSAR target (`autosar.tlc`) using the `Software-in-the-Loop (SIL)` and `Processor-in-the-Loop (PIL)` simulation modes.

For more information, see "Top-Model SIL or PIL Simulation" on page 39-16 and "Model Block SIL or PIL Simulation" on page 39-18.

AUTOSAR Top Model SIL and PIL Support

For a top model running in SIL or PIL simulation mode, the software does *not* support the following AUTOSAR features:

- AUTOSAR calibration parameters
- Client-server operations

Logging Invariant Output Signals. Through signal logging, you can configure your top model to log invariant output signals. However, the software will log these invariant signals as periodically sampled data.

AUTOSAR Model Block SIL and PIL Support

The software supports testing of AUTOSAR components that are modeled as model reference components. These model reference components are implemented as standard model reference Simulink Coder targets and do not contain any special AUTOSAR behavior.

Using a SIL or PIL Block for AUTOSAR Verification

To verify source code, you create a SIL block, which wraps the generated code in an S-function. The AUTOSAR target automatically configures the generated S-function to route simulation data using AUTOSAR run-time environment (RTE) API calls.

To verify the behavior of production-intent object code, you create a PIL block. You must provide an implementation of the target connectivity API for this block.

To carry out a verification using a SIL or PIL block:

1. In the Configuration Parameters dialog box, select **Code Generation**, and clear the check box **Generate code only**. If you select **Generate code only**, the software does not create a SIL or PIL block.

2. Select **Code Generation > SIL and PIL Verification**.

3. From the **Create block** drop-down list, select either `SIL` or `PIL`. Click **OK**.

4. To create your SIL or PIL block, generate code in the usual way. See “Exporting AUTOSAR Software Component” on page 24-61 and “Configuring Multiple Runnables” on page 24-47.

5. Once the SIL or PIL block is built, replace the existing component in your model with the new block.

6. Simulate the model and check the output to verify that the code produces the same data as the original subsystem.

Note The software does not propagate non-zero outport initialization inside an AUTOSAR model to the outports (via the RTE) until the step function executes. When you run the generated code in a SIL simulation, you do not see the outport initialization until the SIL wrapper executes the step function for the first time.

For more information about configuring and running simulations with SIL or PIL blocks, see "Using a SIL or PIL Block" on page 39-20.

AUTOSAR SIL and PIL Block Support

The following features are not supported:

- AUTOSAR calibration parameters
- Server operations

Runnable with Stateflow Chart Using Absolute Time. Consider a runnable (function-call subsystem) in a model, which contains a Stateflow chart using absolute-time temporal logic. Replace the runnable with a SIL block and run a simulation with the model. If the SIL block is executed conditionally in the model, then the results of the SIL simulation differ from the results of the Normal mode simulation.

Runnables in Feedback Loops. If your model has function-call subsystems and you export a runnable that has context-dependent inputs (for example, feedback signals), then the results of a SIL/PIL simulation with the generated code may not match the results of the Normal mode simulation of your model. See "Exported Functions in Feedback Loops" on page 39-83.

Limitations and Tips

In this section...

"Cannot Import Internal Behavior" on page 24-68

"Cannot Copy Subsystem Blocks Without Losing Interface Information" on page 24-68

"Error If No Default Configuration" on page 24-69

"The Generate Code Only Check Box" on page 24-69

"Specify Sample Time Independent Server Operation Model" on page 24-69

"Invoke AUTOSAR Server Operation Block in Referenced Model" on page 24-69

"Cannot Save Importer Objects in MAT-Files" on page 24-69

"Using the Merge Block for Inter-Runnable Variables" on page 24-70

"Using Goto and From Blocks Within Wrapper Subsystems" on page 24-70

"AUTOSAR Compiler Abstraction Macros" on page 24-70

"Intrinsic Fixed-Point Types for Model Configured as Server" on page 24-71

"Server Operation Model with Tunable Parameters" on page 24-71

"Migrating AUTOSAR Development Kit Models" on page 24-72

Cannot Import Internal Behavior

Internal behavior is not parsed. This means any I/O information stored at the runnable level (for example, implicit or explicit) is not imported, and all internal I/O settings default to implicit. You can subsequently configure these I/O ports with the `setIODataAccessMode` method or in the Configure AUTOSAR Interface dialog box.

Cannot Copy Subsystem Blocks Without Losing Interface Information

If you copy and paste a subsystem block to create a new block in either a new model or the same model, the AUTOSAR interface information stored with the original subsystem block does not copy to the new subsystem block.

Error If No Default Configuration

If you do not configure your model using the **Get Default Configuration** button or the `RTW.AutosarInterface.getDefaultConf` method, when you build the model the software produces an error message indicating this.

The Generate Code Only Check Box

If you do not select the **Generate code only** check box, the software produces an error message when you build the model. The message states that you can build an executable with the AUTOSAR target only if you:

- Configure the model to create a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block
- Run the model in SIL or PIL simulation mode
- Provide a custom template makefile

Specify Sample Time Independent Server Operation Model

For a server operation model, MathWorks recommends that you set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane to `Ensure sample time independent`. This action ensures that you can specify the sample time of the client model that invokes the server model (through an Invoke AUTOSAR Server Operation block) independently of the server model. If you do not specify this parameter, you must ensure that the client block calls the server block at the same sample time. Otherwise, the data returned from the server model may be invalid.

Invoke AUTOSAR Server Operation Block in Referenced Model

The software does not support the use of the Invoke AUTOSAR Server Operation block in a referenced model.

Cannot Save Importer Objects in MAT-Files

If you try to save an `arxml.importer` object in a MAT-file, you lose all the information. If you reload the MAT-file, then the object is null (handle = –1), because of the Java™ objects that compose the `arxml.importer` object.

Using the Merge Block for Inter-Runnable Variables

You can use the Merge block to merge inter-runnable variables. However, you must do the following:

- Ensure that the output signal of the Merge block is connected to either one root outport or one or more subsystems.
- If the output signal of the Merge block is connected to the inputs of one or more subsystems, assign the same signal name to the Merge block's output and inputs.

Using Goto and From Blocks Within Wrapper Subsystems

If your wrapper subsystem contains Goto and From blocks, you can generate code and XML files for multiple runnables. However, you must:

- Set the scope of the Goto block tag to `local`.
- Label, unambiguously, signals entering and leaving Goto and From blocks that connect subsystems. If the signal into the Goto block and the signal out of the corresponding From block connect two subsystems (runnables), then these signal segments represent an inter-runnable variable. You must provide a label for at least one signal segment, for example, the signal into the Goto block. In addition, if there are labels for other segments of the same signal, you must ensure that these labels are the same.

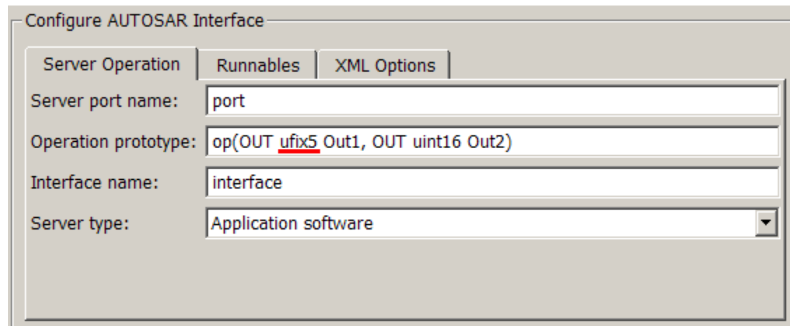
AUTOSAR Compiler Abstraction Macros

The software does not generate AUTOSAR compiler abstraction macros for data or functions arising from the following:

- Model blocks
- Stateflow
- MATLAB Coder
- Shared utility functions
- Custom storage classes
- Local or temporary variables

Intrinsic Fixed-Point Types for Model Configured as Server

The software does not support operation prototype arguments with intrinsic fixed-point data types. For example, `ufix5` shown in the following figure.



The software produces an error when you build the model.

To work around this limitation, before building the model, create a `Simulink.NumericType` base workspace object with the required property values. For example, to create a `Simulink.NumericType` object `ufix5`, enter the following in the Command Window:

```
>> ufix5 = Simulink.NumericType;
>> ufix5.DataTypeMode = 'Fixed-point: binary point scaling';
>> ufix5.Signedness = 'Signed'
>> ufix5.WordLength = 16
>> ufix5.FractionLength = 0
>> ufix5.IsAlias = 1;
>> ufix.HeaderFile = 'Rte_Type.h'
```

For more information, see Simulink.NumericType in the Simulink documentation.

Server Operation Model with Tunable Parameters

The software does not provide AUTOSAR support for a model that is configured as a server operation and has tunable parameters with storage class set to `SimulinkGlobal (Auto)`.

Migrating AUTOSAR Development Kit Models

Use the `autosar_adk_migrate` function to migrate an AUTOSAR Development Kit (ADK) model (from releases before R2008a) to the AUTOSAR interface.

Enter:

```
autosar_adk_migrate(PATHNAME)
```

to migrate the ADK model/system specified by the full path name `PATHNAME` from the ADK settings to the new AUTOSAR interface. The model must be open before you invoke this function. MathWorks recommends that you save the migrated model to a different file name.

Demos and Further Reading

AUTOSAR Demos

For detailed explanations of AUTOSAR workflows with Embedded Coder software, see the demos in the following table.

Demo	How to ...
AUTOSAR Code Generation: rtwdemo_autosar_legacy_script	Generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files from a Simulink model
Using an AUTOSAR Client-Server Interface rtwdemo_autosar_clientserver_script	Configure and generate AUTOSAR-compliant code and export AUTOSAR-compliant XML files for a Simulink model with an AUTOSAR client-server interface
AUTOSAR Code Generation for Multiple Runnable Entities: rtwdemo_autosar_multirunnables_script	Configure and generate AUTOSAR-compliant code and export AUTOSAR Software Component description XML files for a Simulink model with multiple runnables.
Import and Export an AUTOSAR Software Component: rtwdemo_autosar_roundtrip_script	Use an AUTOSAR authoring tool with Simulink to develop AUTOSAR Software Components. Learn how to import software component interfaces into Simulink, modify and export them, and merge the completed software component back into an AUTOSAR authoring tool.
Using Data Stores to Access Per-Instance Memory: rtwdemo_autosar_PIM_script	Publish an AUTOSAR Software Component with per-instance memory

Further Reading

For more information, see the AUTOSAR Web site:
`http://www.autosar.org/`

Integrating External Code and Generated C and C++ Code

- Chapter 25, "About External Code Integration Extensions"
- Chapter 26, "Generating S-Function Wrappers"
- Chapter 27, "Exporting Function-Call Subsystems"
- Chapter 28, "Nonvirtual Subsystem Modular Function Code Generation"
- Chapter 29, "Controlling Generation of Function Prototypes"
- Chapter 30, "Controlling Generation of Encapsulated C++ Model Interfaces"
- Chapter 31, "Replacing Math Functions and Operators Using Target Function Libraries"

25

About External Code Integration Extensions

The Simulink Coder documentation introduces capabilities for integrating external code with generated C and C++ code. Topics include

- “About External Code Integration”
- “Integrating External Code Using Model Configuration Parameters”
- “Integrating External Code Using Custom Code Blocks”
- “Integrating External Code Using S-Functions”

The Embedded Coder product extends the preceding capabilities to support

- Chapter 26, “Generating S-Function Wrappers”
- Chapter 27, “Exporting Function-Call Subsystems”
- Chapter 28, “Nonvirtual Subsystem Modular Function Code Generation”
- Chapter 29, “Controlling Generation of Function Prototypes”
- Chapter 30, “Controlling Generation of Encapsulated C++ Model Interfaces”
- Chapter 31, “Replacing Math Functions and Operators Using Target Function Libraries”

26

Generating S-Function Wrappers

- "About S-Function Wrapper Generation" on page 26-2
- "Creating a SIL Block" on page 26-3
- "S-Function Wrapper Generation Limitations" on page 26-4

About S-Function Wrapper Generation

An S-function wrapper is an S-function that calls your C or C++ code from within Simulink. S-function wrappers provide a standard interface between Simulink and externally written code, allowing you to integrate your code into a model with minimal modification. This is useful for software-in-the-loop (SIL) code verification (validating your generated code in Simulink), as well as for simulation acceleration purposes (see "Using a SIL or PIL Block" on page 39-20). For a complete description of wrapper S-functions, see the Simulink Writing S-Functions document.

Using the **Create block** parameter, on the **Code Generation > SIL and PIL Verification** pane, you can build in one automated step:

- A noninlined C or C++ MEX S-function wrapper that calls generated code
- A model with a SIL block. This block, which contains the generated S-function, is ready for use with other blocks or models

When the **Create block** parameter is set to `SIL`, the build process generates an additional source code file, *`model`*`_sf.c` or `.cpp`, in the build directory. This module contains the S-function that calls the generated code that you deploy. You can use this S-function within Simulink.

The build process then compiles and links *`model`*`_sf.c` or `.cpp` with *`model`*`.c` or `.cpp` and the other generated code modules, building a MEX-file. The MEX-file is named *`model`*`_sf.`*`mexext`*. (*`mexext`* is the file extension for MEX-files on your platform, as given by the MATLAB `mexext` command.) The MEX-file is stored in your working directory. Finally, the build process creates and opens an untitled model containing the SIL block with the generated S-function.

Note To generate a wrapper S-function for a subsystem, you can use a right-click subsystem build. Right-click the subsystem block in your model, select **Code Generation > Generate S-Function**, and in the Generate S-Function dialog box, select **Create Software-In-the-Loop (SIL) block** and click **Build**.

Creating a SIL Block

To create a SIL block with the S-function wrapper for your generated code, open your ERT-based Simulink model and do the following:

1. Open the Configuration Parameters dialog box.
2. Select the **Code Generation > SIL and PIL Verification** pane.
3. From the **Create block** drop-down list, select `SIL`.
4. Configure the other code generation options as required.
5. To ensure that memory for the S-function is initialized to zero, you must clear the following options in the **Data initialization** section of the **Optimization > General** pane:
   - **“Remove root level I/O zero initialization”**
   - **“Remove internal data zero initialization”**
   - **“Use memset to initialize floats and doubles to 0.0”**
6. Select the **Code Generation** pane, and click **Build**.
7. When the build process completes, an untitled model window opens. This model contains a SIL block with the generated S-function.
8. Save the new model.
9. You can now use the SIL block with other blocks or models in Simulink.

S-Function Wrapper Generation Limitations

The following limitations apply to Embedded Coder S-function wrapper generation:

- Continuous sample time is not supported. The **Support continuous time** option should not be selected when creating a SIL block.
- Models that contain S-function blocks for which the S-function is not inlined with a TLC file are not supported when creating a SIL block.
- You cannot use multiple instances of a SIL block within a model, because the code uses static memory allocation. Each instance potentially can overwrite global data values of the others.
- SIL blocks can be used with other blocks and models for SIL code verification and simulation acceleration, but they cannot be used for code generation.
- A MEX S-function wrapper must only be used in the version of MATLAB in which the wrapper is created.

Exporting Function-Call Subsystems

- “Overview” on page 27-2
- “Requirements for Exporting Function-Call Subsystems” on page 27-4
- “Techniques for Exporting Function-Call Subsystems” on page 27-7
- “Optimizing Exported Function-Call Subsystems” on page 27-10
- “Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 27-11
- “Function-Call Subsystem Export Example” on page 27-12
- “Function-Call Subsystems Export Limitations” on page 27-16

Overview

Embedded Coder software provides code export capabilities that you can use to

- Automatically generate code for
  - A function-call subsystem that contains only blocks that support code generation
  - A virtual subsystem that contains only such subsystems and a few other types of blocks
- Create a SIL block that represents the generated code

You can use these capabilities only if the subsystem and its interface to the Simulink model conform to certain requirements and constraints, as described in "Requirements for Exporting Function-Call Subsystems" on page 27-4. For limitations that apply, see "Function-Call Subsystems Export Limitations" on page 27-16.

Note For models designed in earlier releases, Embedded Coder software also supports the ability to export functions from triggered subsystems. In general, the requirements and limitations stated for exporting functions from function-call subsystems also apply to exporting functions from triggered subsystems, with the following exceptions:

- Triggered subsystems from which you intend to export functions must be encapsulated in a single top-level virtual subsystem.
- Triggered subsystems do not have to meet the requirements in "Trigger Signals Require a Common Source" on page 27-5 and "Requirements for Exported Virtual Subsystems" on page 27-5.
- The section "Exporting Function-Call Subsystems That Depend on Elapsed Time" on page 27-11 is not applicable to exporting functions from triggered subsystems.

Exported Subsystems Demo

To see a demo of exported function-call subsystems, type `rtwdemo_export_functions` in the MATLAB Command Window.

Additional Information

See the following in the Simulink documentation for additional information relating to exporting function-call subsystems:

- "Systems and Subsystems"
- "Signals"
- "Triggered Subsystems"
- "Function-Call Subsystems"
- *Developing S-Functions*

If you want to use Stateflow blocks to trigger exportable function-call subsystems, you may also need information from the *Stateflow User's Guide*.

Requirements for Exporting Function-Call Subsystems

To be exportable as code, a function-call subsystem, or a virtual subsystem that contains such subsystems, must meet certain requirements. Most requirements are similar for either type of export, but some apply only to virtual subsystems. The requirements that affect all Simulink code generation also apply.

For brevity, *exported subsystem* in this section means only an exported function-call subsystem or an exported virtual subsystem that contains such subsystems. The requirements listed do not necessarily apply to other types of exported subsystems.

Requirements for All Exported Subsystems

These requirements apply to both exported function-call subsystems and exported virtual subsystems that contain such subsystems.

Blocks Must Support Code Generation

All blocks within an exported subsystem must support code generation. However, blocks outside the subsystem need not support code generation unless they will be converted to code in some other context.

Blocks Must Not Use Absolute Time

Certain blocks use absolute time. Blocks that use absolute time are not supported in exported function-call subsystems. For a complete list of such blocks, see “Limitations on the Use of Absolute Time” in the Simulink Coder documentation.

Blocks Must Not Depend on Elapsed Time

Certain blocks, like the Sine Wave block and Discrete Integrator block, depend on elapsed time. If an exported function-call subsystem contains any blocks that depend on elapsed time, the subsystem must specify periodic execution. See “Exporting Function-Call Subsystems That Depend on Elapsed Time” on page 27-11 in the Simulink Coder documentation.

Trigger Signals Require a Common Source

If more than one trigger signal crosses the boundary of an exported system, all of the trigger signals must be periodic and originate from the same function-call initiator.

Trigger Signals Must Be Scalar

A trigger signal that crosses the boundary of an exported subsystem must be scalar. Input and output data signals that do not act as triggers need not be scalar.

Data Signals Must Be Nonvirtual

A data signal that crosses the boundary of an exported system cannot be a virtual bus, and cannot be implemented as a `Goto-From` connection. Every data signal crossing the export boundary must be scalar, muxed, or a nonvirtual bus.

Requirements for Exported Virtual Subsystems

These requirements apply only to exported virtual subsystems that contain function-call subsystems.

Virtual Subsystem Must Use Only Permissible Blocks

The top level of an exported virtual subsystem that contains function-call subsystem blocks can contain only the following other types of blocks:

- Input and Output blocks (ports)
- Constant blocks (including blocks that resolve to constants, such as Add)
- Merge blocks
- Virtual connection blocks (Mux, Demux, Bus Creator, Bus Selector, Signal Specification)
- Signal-viewer blocks, such as `Scope` blocks

These restrictions do *not* apply within function-call subsystems, whether or not they appear in a virtual subsystem. They apply only at the top level

of an exported virtual subsystem that contains one or more function-call subsystems.

Constant Blocks Must Be Inlined

When a constant block appears at the top level of an exported virtual subsystem, the containing model must check **Inline parameters** on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box.

Constant Outputs Must Specify a Storage Class

When a constant signal drives an output port of an exported virtual subsystem, the signal must specify a storage class.

Techniques for Exporting Function-Call Subsystems

In this section...

"General Workflow" on page 27-7
"Specifying a Custom Initialize Function Name" on page 27-8
"Specifying a Custom Description" on page 27-8

General Workflow

To export a function-call subsystem, or a virtual subsystem that contains function-call subsystems,

1. Ensure that the subsystem to be exported satisfies the "Requirements for Exporting Function-Call Subsystems" on page 27-4.

2. In the Configuration Parameters dialog box:

   a. On the **Code Generation** pane, specify an ERT code generation target such as `ert.tlc`.

   b. If you want a SIL block with the generated code, go to the **SIL and PIL Verification** pane and, from the **Create block** drop-down list, select `SIL`.

   c. Click **OK** or **Apply**.

3. Right-click the subsystem block and choose **Code Generation > Export Functions** from the context menu.

   The `Build code for subsystem:` *`Subsystem`* dialog box appears. This dialog box is not specific to exporting function-call subsystems, and generating code does not require entering information in the box.

4. Click **Build**.

   The MATLAB Command Window displays messages similar to those for any code generation sequence. Simulink generates code and places it in the working directory.

If you set **Create block** to `SIL` in step 2b, Simulink opens a new window that contains an S-function block that represents the generated code. This block has the same size, shape, and connectors as the original subsystem.

Code generation and optional block creation are now complete. You can test and use the code and optional block as you could any generated ERT code and S-function block.

Specifying a Custom Initialize Function Name

You can specify a custom name for the initialize function of your exported function as an argument to the `rtwbuild` command. When used for this purpose, the command takes the following form:

```
blockHandle = rtwbuild('subsystem', 'Mode', 'ExportFunctionCalls',..
              'ExportFunctionInitializeFunctionName', 'fcnname')
```

where *`fcnname`* specifies the desired function name. For example, if you specify the name `'myinitfcn'`, the build process emits code similar to the following:

```
/* Model initialize function */
void myinitfcn(void){
...
}
```

Specifying a Custom Description

You can enter a custom description for an exported function using the Block Properties dialog box of an Inport block. To do this, go to the subsystem that is to be exported as a function, right-click on the Inport block that drives the control port of the subsystem, and select **Block Properties**. In the **General** tab, use the **Description** field to enter your descriptive text. During function export, the text you enter is emitted to the generated code in the header for the Inport block. For example, if you open the demo program `rtwdemo_export_functions` and enter a description in the Block Properties dialog box for port `t_1tic_A`, code similar to the following is emitted:

```
/*
 * Output and update for exported function: t_1tic_A
 *
```

```
 *  My custom description of the exported function
*/
void t_1tic_A(void)
{
...
}
```

Optimizing Exported Function-Call Subsystems

To optimize the code generated for a function-call subsystem or virtual block that contains such subsystems, you can

- Specify a storage class for every input signal and output signal that crosses the boundary of the subsystem.
- For each function-call subsystem to be exported (whether directly or within a virtual subsystem):

  1. Right-click the subsystem and choose **Subsystem Parameters** from the context menu.
  2. Select the **Code Generation** tab and set the **Function packaging** parameter to `Auto`.
  3. Click **OK** or **Apply**.

Exporting Function-Call Subsystems That Depend on Elapsed Time

Some blocks, such as the Sine Wave block (if sample-based) and the Discrete-Time Integrator block, depend on elapsed time. See “Absolute and Elapsed Time Computation” in the Simulink Coder documentation for more information.

When a block that depends on elapsed time exists in a function-call subsystem, the subsystem cannot be exported unless it specifies periodic execution. To provide the necessary specification,

1. Right-click the trigger port block in the function-call subsystem and choose **TriggerPort Parameters** from the context menu.

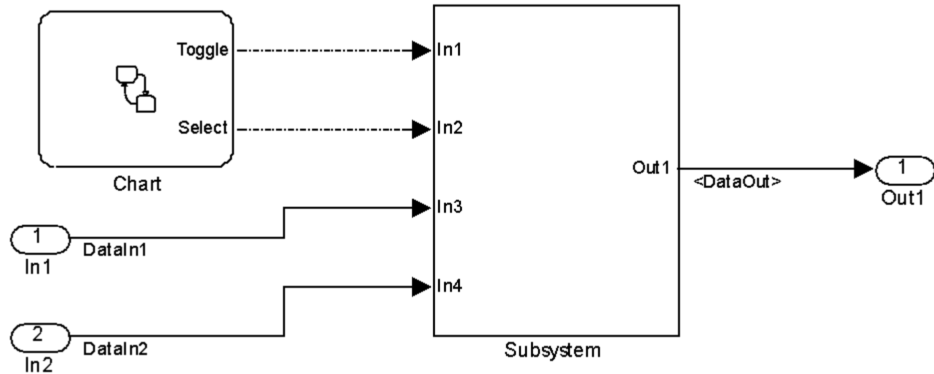
2. Specify **periodic** in the **Sample time type** field.

3. Set the **Sample time** to the same granularity specified (directly or by inheritance) in the function-call initiator.

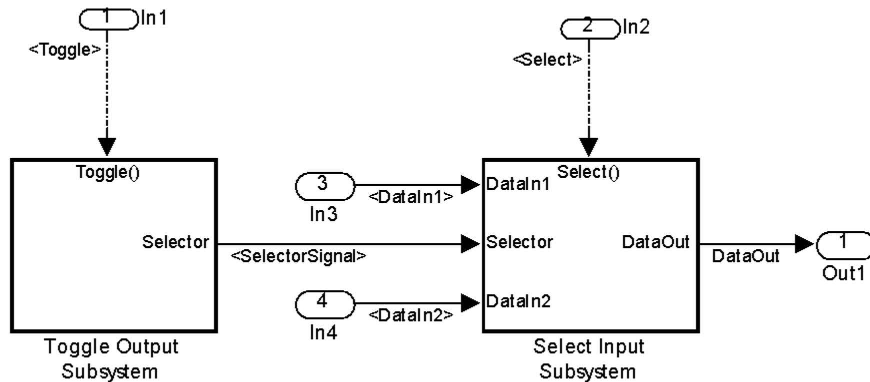
4. Click **OK** or **Apply**.

Function-Call Subsystem Export Example

The next figure shows the top level of a model that uses a Stateflow chart named `Chart` to input two function-call trigger signals (denoted by dash-dot lines) to a virtual subsystem named `Subsystem`.



The next figure shows the contents of `Subsystem` in the previous figure. The subsystem contains two function-call subsystems, each driven by one of the signals input from the top level.



In the preceding model, the Stateflow chart can assert either of two scalar signals, `Toggle` and `Select`.

- Asserting `Toggle` toggles the Boolean state of the function-call subsystem `Toggle Output Subsystem`.
- Asserting `Select` causes the function-call subsystem `Select Input Subsystem` to assign the value of `DataIn1` or `DataIn2` to its output signal. The value assigned depends on the current state of `Toggle Output Subsystem`.

The following generated code implements the subsystem named `Subsystem`. The code is typical for virtual subsystems that contain function-call subsystems. It specifies an initialization function and a function for each contained subsystem, and would also include functions to enable and disable subsystems if applicable.

```
#include "Subsystem.h"
#include "Subsystem_private.h"

/* Exported block signals */
real_T DataIn1;                         /* '<Root>/In3' */
real_T DataIn2;                         /* '<Root>/In4' */
real_T DataOut;                         /* '<S4>/Switch' */
boolean_T SelectorSignal;               /* '<S5>/Logical Operator' */

/* Exported block states */
boolean_T SelectorState;                /* '<S5>/Unit Delay' */

/* Real-time model */
RT_MODEL_Subsystem Subsystem_M_;
RT_MODEL_Subsystem *Subsystem_M = &Subsystem_M_;

/* Initial conditions for exported function: Toggle */

void Toggle_Init(void)
{
  /* Initial conditions for function-call system: '<S1>/Toggle Output Subsystem' */

  /* InitializeConditions for UnitDelay: '<S5>/Unit Delay' */
  SelectorState = Subsystem_P.UnitDelay_X0;
}

/* Output and update for exported function: Toggle */
```

```
void Toggle(void)
{
  /* Output and update for function-call system: '<S1>/Toggle Output Subsystem' */

  /* Logic: '<S5>/Logical Operator' incorporates:
   *  UnitDelay: '<S5>/Unit Delay'
   */
  SelectorSignal = !SelectorState;

  /* Update for UnitDelay: '<S5>/Unit Delay' */
  SelectorState = SelectorSignal;
}

/* Output and update for exported function: Select */

void Select(void)
{
  /* Output and update for function-call system: '<S1>/Select Input Subsystem' */

  /* Switch: '<S4>/Switch' incorporates:
   *  Inport: '<Root>/In3'
   *  Inport: '<Root>/In4'
   */
  if(SelectorSignal) {
    DataOut = DataIn1;
  } else {
    DataOut = DataIn2;
  }
}

/* Model initialize function */

void Subsystem_initialize(void)
{
  /* initialize error status */
  rtmSetErrorStatus(Subsystem_M, (const char_T *)0);

  /* block I/O */
```

```
  /* exported global signals */
  DataOut = 0.0;
  SelectorSignal = FALSE;

  /* states (dwork) */

  /* exported global states */
  SelectorState = FALSE;

  /* external inputs */
  DataIn1 = 0.0;
  DataIn2 = 0.0;

  Toggle_Init();
}

/* Model terminate function */

void Subsystem_terminate(void)
{
  /* (no terminate code required) */
}
```

Function-Call Subsystems Export Limitations

The function-call subsystem export capabilities have the following limitations:

- Subsystem block parameters do not control the names of the files containing the generated code. All such filenames begin with the name of the exported subsystem. Each filename is suffixed as appropriate to the file.
- Subsystem block parameters do not control the names of top-level functions in the generated code. Each function name reflects the name of the signal that triggers the function, or for an unnamed signal, the block from which the signal originates.
- The software cannot export reusable code for a function-call subsystem. Checking **Configuration Parameters > Code Generation > Interface > Generate reusable code** has no effect on the generated code for the subsystem.
- The software supports code generation for a SIL block provided the block does not have function-call input ports. However, the block will appear as a noninlined S-function in the generated code.
- The software supports a SIL block in accelerator mode only if its function-call initiator is noninlined in accelerator mode. Examples of noninlined initiators include all Stateflow charts.
- The SIL block must be driven by a Level-2 S-function initiator block, such as a Stateflow chart or the built-in Function-call Generator block.
- An asynchronous (sample-time) function-call system can be exported, but the software does not support the ERT S-function wrapper for an asynchronous system.
- The software does not support code generation for a SIL block if the block is generated for exported function calls.
- The output from a SIL block cannot be merged using the Merge block.
- The software does not support MAT-file logging for exported function calls. Any specification that enables MAT-file logging is ignored.
- The use of the TLC function `LibIsFirstInit` is deprecated for exported function calls.

- The *model*_initialize function generated in the code for an exported function-call subsystem never includes a firstTime argument, regardless of the value of the model configuration parameter IncludeERTFirstTime. Thus, you cannot call *model*_initialize at a time greater than start time, for example, to reset block states.

28

Nonvirtual Subsystem Modular Function Code Generation

- “Overview” on page 28-2
- “Configuring Nonvirtual Subsystems for Generating Modular Function Code” on page 28-4
- “Examples of Modular Function Code for Nonvirtual Subsystems” on page 28-9
- “Nonvirtual Subsystem Modular Function Code Limitations” on page 28-15

Overview

The Embedded Coder software provides a subsystem option, **Function with separate data**, that allows you to generate modular function code for nonvirtual subsystems, including atomic subsystems and conditionally executed subsystems.

By default, the generated code for a nonvirtual subsystem does not separate a subsystem's internal data from the data of its parent Simulink model. This can make it difficult to trace and test the code, particularly for nonreusable subsystems. Also, in large models containing nonvirtual subsystems, data structures can become large and potentially difficult to compile.

The Subsystem Parameters dialog box option **Function with separate data** allows you to generate subsystem function code in which the internal data for a nonvirtual subsystem is separated from its parent model and is owned by the subsystem. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the size of data structures throughout the model.

Note Selecting the **Function with separate data** option for a nonvirtual subsystem has no semantic effect on the parent Simulink model.

To be able to use this option,

- Your Simulink model must use an ERT-based system target file (requires a Embedded Coder license).
- Your subsystem must be configured to be atomic or conditionally executed (for more information, see "Systems and Subsystems" in the Simulink documentation).
- Your subsystem must use the `Function` setting for the **Code Generation > Function packaging**.

To configure your subsystem for generating modular function code, you invoke the Subsystem Parameters dialog box and make a series of selections to display and enable the **Function with separate data** option. See "Configuring Nonvirtual Subsystems for Generating Modular Function

Code" on page 28-4 and "Examples of Modular Function Code for Nonvirtual Subsystems" on page 28-9 for details. For limitations that apply, see "Nonvirtual Subsystem Modular Function Code Limitations" on page 28-15.

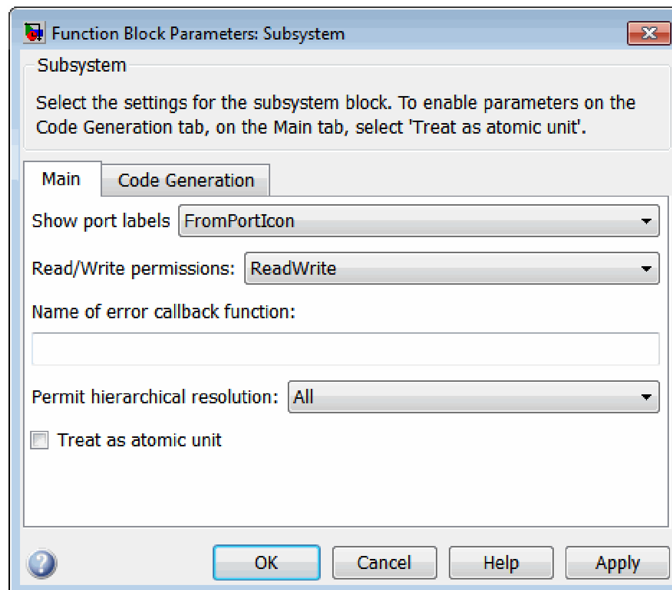
For more information about generating code for atomic subsystems, see the sections "Creating Subsystems" and "Generating Code and Executables from Subsystems" in the Simulink Coder documentation.

Configuring Nonvirtual Subsystems for Generating Modular Function Code

This section summarizes the steps needed to configure a subsystem in a Simulink model for modular function code generation.

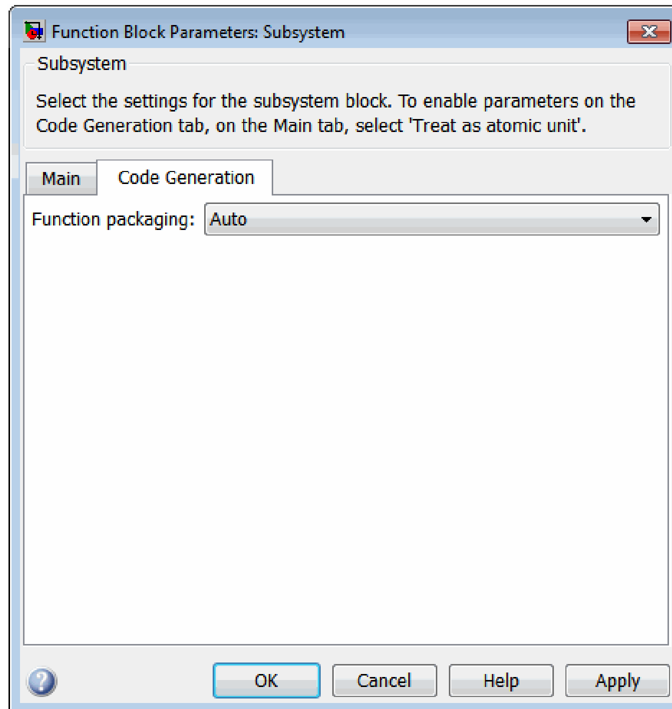
1. Verify that the Simulink model containing the subsystem uses an ERT-based system target file (see the **System target file** parameter on the **Code Generation** pane of the Configuration Parameters dialog box).

2. In your Simulink model, select the subsystem for which you want to generate modular function code and launch the Subsystem Parameters dialog box (for example, right-click the subsystem and select **Subsystem Parameters**). The dialog box for an atomic subsystem is shown below. (In the dialog box for a conditionally executed subsystem, the dialog box option **Treat as atomic unit** is greyed out, and you can skip Step 3.)

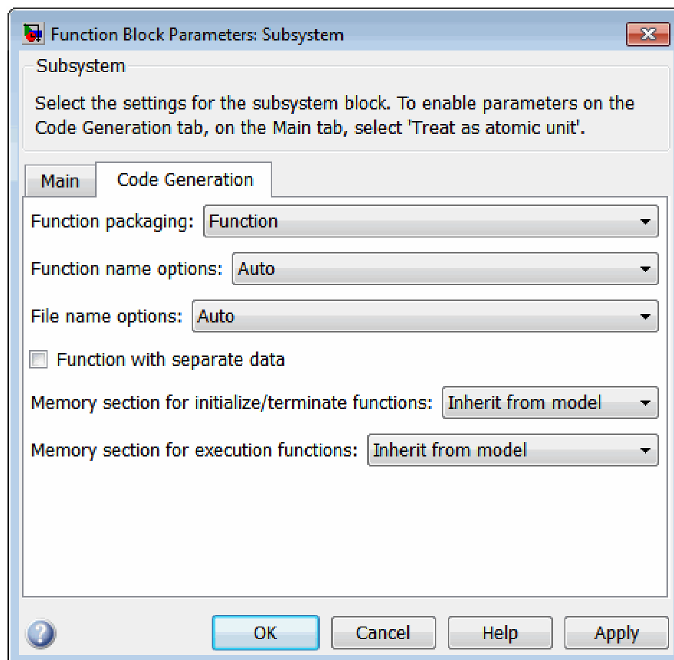


3. If the Subsystem Parameters dialog box option **Treat as atomic unit** is available for selection but not selected, the subsystem is neither atomic nor conditionally executed. Select the option **Treat as atomic unit**, which

enables **Function packaging** on the **Code Generation** tab. Select the **Code Generation** tab.



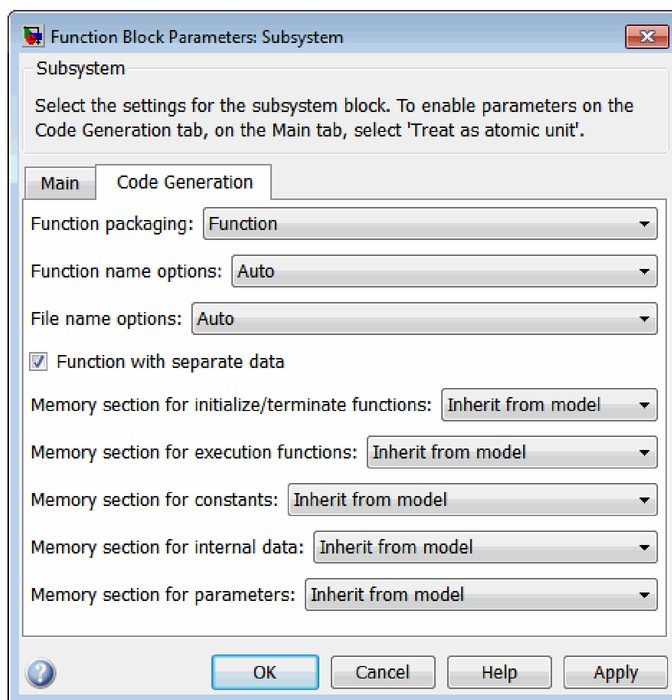
4 For the **Function packaging** parameter, select the value `Function`. After you make this selection, the **Function with separate data** option is displayed.



Note Before you generate nonvirtual subsystem function code with the **Function with separate data** option selected, you might want to generate function code with the option *deselected* and save the generated function `.c` and `.h` files in a separate directory for later comparison.

---

5 Select the **Function with separate data** option. After you make this selection, additional configuration parameters are displayed.



Note To control the naming of the subsystem function and the subsystem files in the generated code, you can modify the subsystem parameters **Function name options** and **File name options**.

---

6 To save your subsystem parameter settings and exit the dialog box, click **OK**.

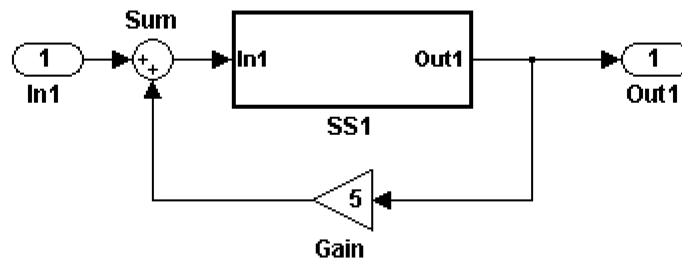
This completes the subsystem configuration needed to generate modular function code. You can now generate the code for the subsystem and examine the generated files, including the function `.c` and `.h` files named according to your subsystem parameter specifications. For more information on generating code for nonvirtual subsystems, see "Creating Subsystems" in the Simulink

Coder documentation. For examples of generated subsystem function code, see “Examples of Modular Function Code for Nonvirtual Subsystems” on page 28-9.

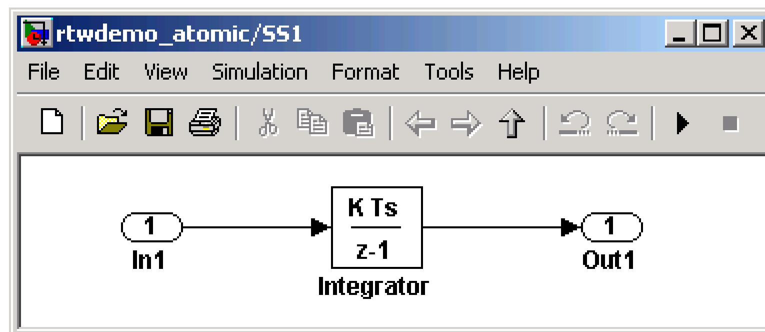
Examples of Modular Function Code for Nonvirtual Subsystems

To illustrate the effect of selecting the **Function with separate data** option for a nonvirtual subsystem, the following procedure generates atomic subsystem function code with and without the option selected and compares the results.

1 Open MATLAB and launch `rtwdemo_atomic.mdl` using the MATLAB command `rtwdemo_atomic`. Examine the Simulink model.



2 Double-click the SS1 subsystem and examine the contents. (You can close the subsystem window when you are finished.)



3 Use the Configuration Parameters dialog box to change the model's **System target file** from GRT to ERT. For example, from the Simulink window, select **Simulation > Configuration Parameters**, select the

Code Generation pane, select **System target file** `ert.tlc`, and click **OK** twice to confirm the change.

4 Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *without* data separation.

   a In the Simulink view of `rtwdemo_atomic.mdl`, right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that

   - On the **Main** tab, **Treat as atomic unit** is selected
   - On the **Code Generation** tab, `User specified` is selected for **Function name options**
   - On the **Code Generation** tab, `myfun` is specified for **Function name**

   b In the Subsystem Parameters dialog box, on the **Code Generation** tab

   i Select the value `Function` for the **Function packaging** parameter. After this selection, additional parameters and options will appear.

   ii Select the value `Use function name` for the **File name options** parameter. This selection is optional but simplifies the later task of code comparison by causing the atomic subsystem function code to be generated into the files `myfun.c` and `myfun.h`.

   Do *not* select the option **Function with separate data**. Click **Apply** to apply the changes and click **OK** to exit the dialog box.

   c Save this model variant to a personal work directory, for example, `d:/atomic/rtwdemo_atomic1.mdl`.

5 Create a variant of `rtwdemo_atomic.mdl` that illustrates function code *with* data separation.

   a In the Simulink view of `rtwdemo_atomic1.mdl` (or `rtwdemo_atomic.mdl` with step 3 reapplied), right-click the SS1 subsystem and select **Subsystem Parameters**. In the Subsystem Parameters dialog box, verify that

   - On the **Main** tab, **Treat as atomic unit** is selected
   - On the **Code Generation** tab, `Function` is selected for **Function packaging**

- On the **Code Generation** tab, `User specified` is selected for **Function name options**
- On the **Code Generation** tab, `myfun` is specified for **Function name**
- On the **Code Generation** tab, `Use function name` is specified for **File name options**

b In the Subsystem Parameters dialog box, on the **Code Generation** tab, select the option **Function with separate data**. Click **Apply** to apply the change and click **OK** to exit the dialog box.

c Save this model variant, using a different name than the first variant, to a personal work directory, for example, `d:/atomic/rtwdemo_atomic2.mdl`.

6. Generate code for each model, `d:/atomic/rtwdemo_atomic1.mdl` and `d:/atomic/rtwdemo_atomic2.mdl`.

7. In the generated code directories, compare the *`model`*`.c/.h` and `myfun.c/.h` files generated for the two models. (In this example, there are no significant differences in the generated variants of `ert_main.c`, *`model`*`_private.h`, *`model`*`_types.h`, or `rtwtypes.h`.)

H File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the H files generated for `rtwdemo_atomic1.mdl` and `rtwdemo_atomic2.mdl` help illustrate the effect of selecting the **Function with separate data** option for nonvirtual subsystems.

1. Selecting **Function with separate data** causes `typedefs` for subsystem data to be generated in the `myfun.h` file for `rtwdemo_atomic2`:

```
/* Block signals for system '<Root>/SS1' */
typedef struct {
  real_T Integrator;                    /* '<S1>/Integrator' */
} rtB_myfun;

/* Block states (auto storage) for system '<Root>/SS1' */
typedef struct {
  real_T Integrator_DSTATE;             /* '<S1>/Integrator' */
} rtDW_myfun;
```

By contrast, for rtwdemo_atomic1, typedefs for subsystem data belong to the model and appear in rtwdemo_atomic1.h:

```
/* Block signals (auto storage) */
typedef struct {
...
    real_T Integrator;                    /* '<S1>/Integrator' */
} BlockIO_rtwdemo_atomic1;

/* Block states (auto storage) for system '<Root>' */
typedef struct {
  real_T Integrator_DSTATE;               /* '<S1>/Integrator' */
} D_Work_rtwdemo_atomic1;
```

2 Selecting **Function with separate data** generates the following external declarations in the myfun.h file for rtwdemo_atomic2:

```
/* Extern declarations of internal data for 'system '<Root>/SS1'' */
extern rtB_myfun rtwdemo_atomic2_myfunB;

extern rtDW_myfun rtwdemo_atomic2_myfunDW;

extern void myfun_initialize(void);
```

By contrast, the generated code for rtwdemo_atomic1 contains model-level external declarations for the subsystem's BlockIO and D_Work data, in rtwdemo_atomic1.h:

```
/* Block signals (auto storage) */
extern BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
extern D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

C File Differences for Nonvirtual Subsystem Function Data Separation

The differences between the C files generated for rtwdemo_atomic1.mdl and rtwdemo_atomic2.mdl illustrate the key effects of selecting the **Function with separate data** option for nonvirtual subsystems.

1 Selecting **Function with separate data** causes a separate subsystem initialize function, `myfun_initialize`, to be generated in the `myfun.c` file for `rtwdemo_atomic2`:

```
void myfun_initialize(void) {
  {
    ((real_T*)&rtwdemo_atomic2_myfunB.Integrator)[0] = 0.0;
  }
  rtwdemo_atomic2_myfunDW.Integrator_DSTATE = 0.0;
}
```

The subsystem initialize function in `myfun.c` is invoked by the model initialize function in `rtwdemo_atomic2.c`:

```
/* Model initialize function */

void rtwdemo_atomic2_initialize(void)
{
...

  /* Initialize subsystem data */
  myfun_initialize();
}
```

By contrast, for `rtwdemo_atomic1`, subsystem data is initialized by the model initialize function in `rtwdemo_atomic1.c`:

```
/* Model initialize function */

void rtwdemo_atomic1_initialize(void)
{
...
  /* block I/O */
  {
 ...
    ((real_T*)&rtwdemo_atomic1_B.Integrator)[0] = 0.0;
  }

  /* states (dwork) */

  rtwdemo_atomic1_DWork.Integrator_DSTATE = 0.0;
```

```
...
}
```

2 Selecting **Function with separate data** generates the following declarations in the `myfun.c` file for `rtwdemo_atomic2`:

```
/* Declare variables for internal data of system '<Root>/SS1' */
rtB_myfun rtwdemo_atomic2_myfunB;

rtDW_myfun rtwdemo_atomic2_myfunDW;
```

By contrast, the generated code for `rtwdemo_atomic1` contains model-level declarations for the subsystem's `BlockIO` and `D_Work` data, in `rtwdemo_atomic1.c`:

```
/* Block signals (auto storage) */
BlockIO_rtwdemo_atomic1 rtwdemo_atomic1_B;

/* Block states (auto storage) */
D_Work_rtwdemo_atomic1 rtwdemo_atomic1_DWork;
```

3 Selecting **Function with separate data** generates identifier naming that reflects the subsystem orientation of data items. Notice the references to subsystem data in subsystem functions such as `myfun` and `myfun_update` or in the model's *`model`*`_step` function. For example, compare this code from `myfun` for `rtwdemo_atomic2`

```
/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic2_myfunB.Integrator = rtwdemo_atomic2_myfunDW.Integrator_DSTATE;
```

to the corresponding code from `myfun` for `rtwdemo_atomic1`.

```
/* DiscreteIntegrator: '<S1>/Integrator' */
rtwdemo_atomic1_B.Integrator = rtwdemo_atomic1_DWork.Integrator_DSTATE;
```

Nonvirtual Subsystem Modular Function Code Limitations

The nonvirtual subsystem option **Function with separate data** has the following limitations:

- The **Function with separate data** option is available only in ERT-based Simulink models (requires a Embedded Coder license).
- The nonvirtual subsystem to which the option is applied cannot have multiple sample times or continuous sample times; that is, the subsystem must be single-rate with a discrete sample time.
- The nonvirtual subsystem cannot contain continuous states.
- The nonvirtual subsystem cannot output function call signals.
- The nonvirtual subsystem cannot contain noninlined S-functions.
- The generated files for the nonvirtual subsystem will reference model-wide header files, such as `*model*.h` and `*model*_private.h`.
- The **Function with separate data** option is incompatible with the **GRT compatible call interface** option, located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Selecting both will generate an error.
- The **Function with separate data** option is incompatible with the **Generate reusable code** option (**Code Generation > Interface** pane). Selecting both will generate an error.
- Although the `*model*_initialize` function generated for a model containing a nonvirtual subsystem that uses the **Function with separate data** option may have a `firstTime` argument, the argument is not used. Thus, you cannot call `*model*_initialize` at a time greater than start time, for example, to reset block states. To suppress inclusion of the `firstTime` flag in the `*model*_initialize` function definition, set the model configuration parameter `IncludeERTFirstTime` to `off`.

29

Controlling Generation of Function Prototypes

- "Overview" on page 29-2
- "Configuring Model Function Prototypes" on page 29-4
- "Model Function Prototypes Example" on page 29-12
- "Configuring Model Function Prototypes Programmatically" on page 29-18
- "Sample Script for Configuring Model Function Prototypes" on page 29-22
- "Verifying Generated Code for Customized Functions" on page 29-23
- "Model Function Prototype Control Limitations" on page 29-24

Overview

The Embedded Coder software provides a **Configure Model Functions** button, located on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, that allows you to control the model function prototypes that are generated for ERT-based Simulink models.

By default, the function prototype of an ERT-based model's generated *model*_step function resembles the following:

```
void model_step(void);
```

The function prototype of an ERT-based model's generated *model*_init function resembles the following:

```
void model_init(void);
```

(For more detailed information about the default calling interface for the *model*_step function, see the model_step reference page.)

The **Configure Model Functions** button on the **Interface** pane provides you flexible control over the model function prototypes that are generated for your model. Clicking **Configure Model Functions** launches a Model Interface dialog box (see "Configuring Model Function Prototypes" on page 29-4). Based on the **Function specification** value you specify for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototypes. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

For more information about using the **Configure Model Functions** button and the Model Interface dialog box, see "Model Function Prototypes Example" on page 29-12 and the demo model `rtwdemo_fcnprotoctrl`, which is preconfigured to demonstrate function prototype control.

Alternatively, you can use function prototype control functions to programmatically control model function prototypes. For more information, see "Configuring Model Function Prototypes Programmatically" on page 29-18.

You can also control model function prototypes for nonvirtual subsystems, if you generate subsystem code using right-click build. To launch the **Model Interface for subsystem** dialog box, use the `RTW.configSubsystemBuild` function.

Right-click building the subsystem generates the step and initialization functions according to the customizations you make. For more information, see "Configuring Function Prototypes for Nonvirtual Subsystems" on page 29-9.

For limitations that apply, see "Model Function Prototype Control Limitations" on page 29-24.

Configuring Model Function Prototypes

In this section...

"Launching the Model Interface Dialog Boxes" on page 29-4
"Default Model Initialize and Step Functions View" on page 29-4
"Model Specific C Prototypes View" on page 29-5
"Configuring Function Prototypes for Nonvirtual Subsystems" on page 29-9

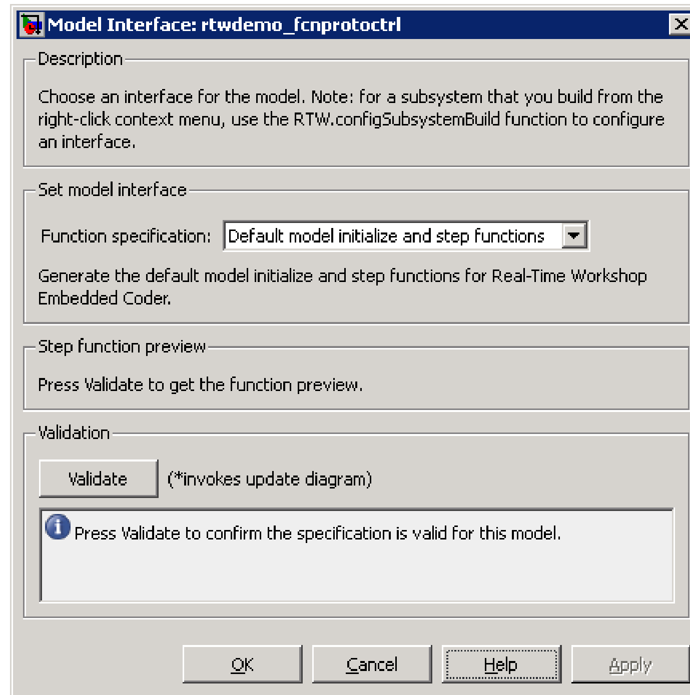
Launching the Model Interface Dialog Boxes

Clicking the **Configure Model Functions** button on the **Interface** pane of the Configuration Parameters dialog box launches the Model Interface dialog box. This dialog box is the starting point for configuring the model function prototypes that are generated during code generation for ERT-based Simulink models. Based on the **Function specification** value you select for your model function (supported values include `Default model initialize and step functions` and `Model specific C prototypes`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications.

To configure function prototypes for a right-click build of a nonvirtual subsystem, invoke the `RTW.configSubsystemBuild` function, which launches the Model Interface for subsystem dialog box. For more information, see "Configuring Function Prototypes for Nonvirtual Subsystems" on page 29-9

Default Model Initialize and Step Functions View

The figure below shows the Model Interface dialog box in the `Default model initialize and step functions` view.



The `Default model initialize and step functions` view allows you to validate and preview the predicted default model step and initialization function prototypes. To validate the default function prototype configuration against your model, click the **Validate** button. If the validation succeeds, the predicted step function prototype appears in the **Step function preview** subpane.

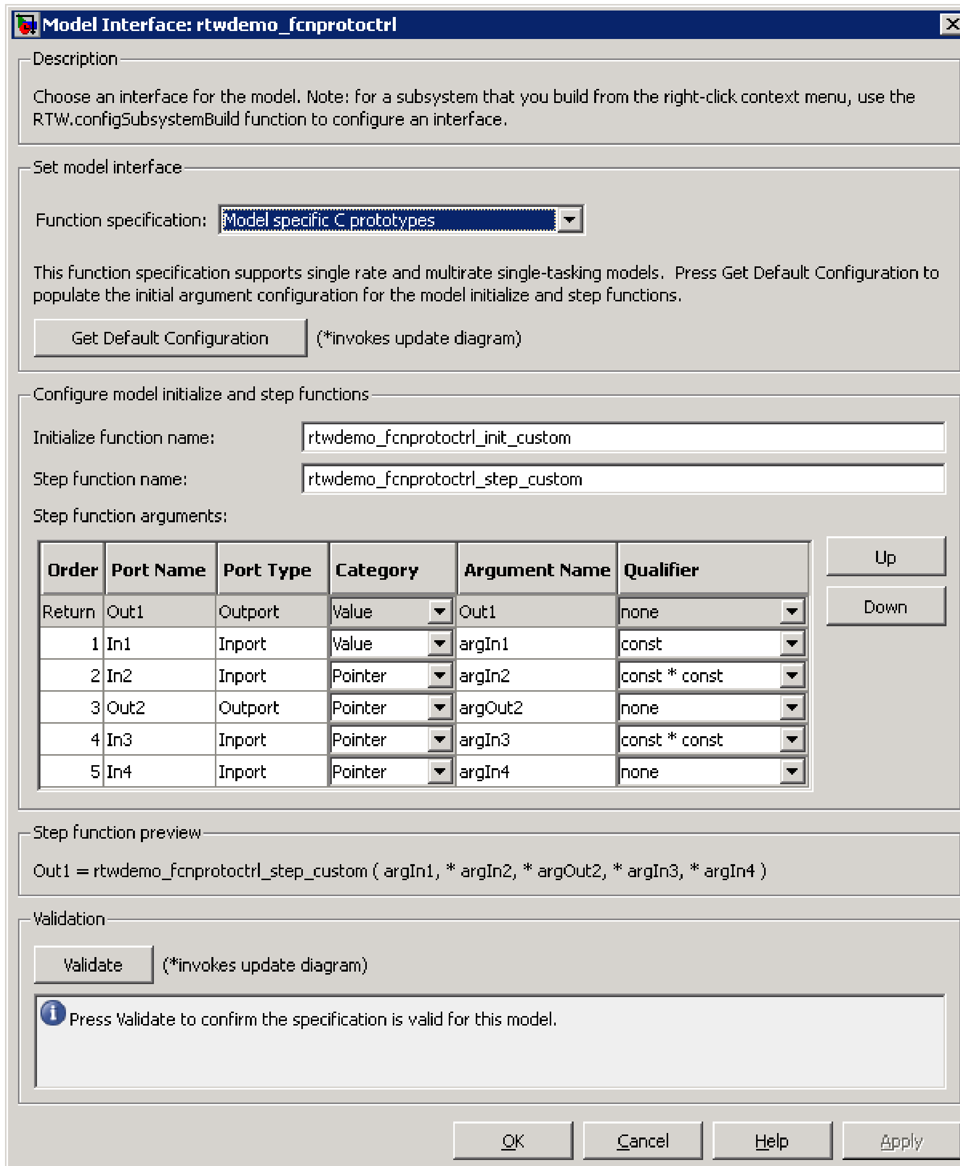
Note You cannot use the `Default model initialize and step functions` view to modify the function prototype configuration.

Model Specific C Prototypes View

Selecting `Model specific C prototypes` for the **Function specification** parameter displays the `Model specific C prototypes` view of your model function prototypes. This view provides controls that you can use to customize

the function names, the order of arguments, and argument attributes including name, passing mechanism, and type qualifier for each of the model's root-level I/O ports.

To begin configuring your function control prototype configuration, click the **Get Default Configuration** button. This activates and initializes the function names and properties in the **Configure model initialize and step functions** subpane, as shown below. If you click **Get Default Configuration** again later, only the properties of the step function arguments are reset to default values.



In the **Configure model initialize and step functions** subpane:

Parameter	Description
Step function name	Name of the `model_step` function.
Initialize function name	Name of the `model_init` function.
Order	Order of the argument. A return argument is listed as `Return`.
Port Name	Name of the port.
Port Type	Type of the port.
Category	Specifies how an argument is passed in or out from the customized step function, either by copying a value (`Value`) or by a pointer to a memory space (`Pointer`).
Argument Name	Name of the argument.
Qualifier (optional)	Specifies a `const` type qualifier for a function argument. The available values are dependent on the **Category** specified. When you change the **Category**, if the specified type is no longer available, the **Qualifier** changes to `none`. The possible values are: • `none` • `const` (value) • `const*` (value referenced by the pointer) • `const*const` (value referenced by the pointer and the pointer itself)

Parameter	Description
	Tip When a model includes a referenced model, the `const` type qualifier for the root input argument of the referenced model's specified step function interface is set to `none`, and the qualifier for the source signal in the referenced model's parent is set to a value other than `none`, code generation honors the referenced model's interface specification by generating a type cast that discards the `const` type qualifier from the source signal. To override this behavior, add a `const` type qualifier to the referenced model.

The **Step function preview** subpane provides a preview of how your step function prototype is interpreted in generated code. The preview is updated dynamically as you make modifications.

An argument `foo` whose **Category** is `Pointer` is previewed as `* foo`. If its **Category** is `Value`, it is previewed as `foo`. Notice that argument types and qualifiers are not represented in the **Step function preview** subpane.

Configuring Function Prototypes for Nonvirtual Subsystems

You can control step and initialization function prototypes for nonvirtual subsystems in ERT-based Simulink models, if you generate subsystem code using right-click build. Function prototype control is supported for the following types of nonvirtual blocks:

- Triggered subsystems
- Enabled subsystems
- Enabled trigger subsystems
- While subsystems
- For subsystems

- Stateflow blocks
- MATLAB function block

To launch the Model Interface for Subsystem dialog box, open the model containing the subsystem and invoke the `RTW.configSubsystemBuild` function.

The Model Interface dialog box for modifying the model-specific C prototypes for the `rtwdemo_counter/Amplifier` subsystem appears as follows:

Model Interface for subsystem: Amplifier

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: Model specific C prototypes

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: Amplifier_initialize

Step function name: Amplifier_custom

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In	Inport	Value	arg_In	none
2	Trigger	Inport	Value	arg_Trigger	none
3	Out	Outport	Pointer	arg_Out	none

Up

Down

Step function preview

Amplifier_custom (arg_In, arg_Trigger, * arg_Out)

Validation

Validate (*invokes update diagram)

Press Validate to confirm the specification is valid for this model.

OK Cancel Help Apply

Right-click building the subsystem generates the step and initialization functions according to the customizations you make.

Model Function Prototypes Example

The following procedure demonstrates how to use the **Configure Model Functions** button on the **Code Generation > Interface** pane of the Configuration Parameters dialog box to control the model function prototypes when generating code for your Simulink model.

1. Open a MATLAB session and launch the `rtwdemo_counter` demo model.

2. In the `rtwdemo_counter` Model Editor, double-click the **Generate Code Using Embedded Coder (double-click)** button to generate code for an ERT-based version of `rtwdemo_counter`. The code generation report for `rtwdemo_counter` appears.

3. In the code generation report, click the link for `rtwdemo_counter.c`.

4. In the `rtwdemo_counter.c` code display, locate and examine the generated code for the `rtwdemo_counter_step` and the `rtwdemo_counter_initialize` functions:

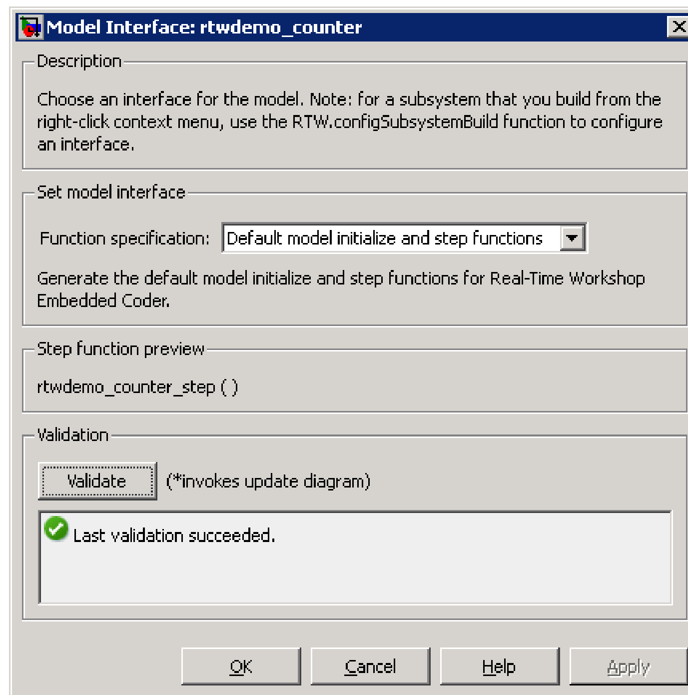
   ```
   /* Model step function */
   void rtwdemo_counter_step(void)
   {
    ...
   }

   /* Model initialize function */
   void rtwdemo_counter_initialize(void)
   {
    ...
   }
   ```

   You can close the report window after you have examined the generated code. Optionally, you can save `rtwdemo_counter.c` and any other generated files of interest to a different location for later comparison.

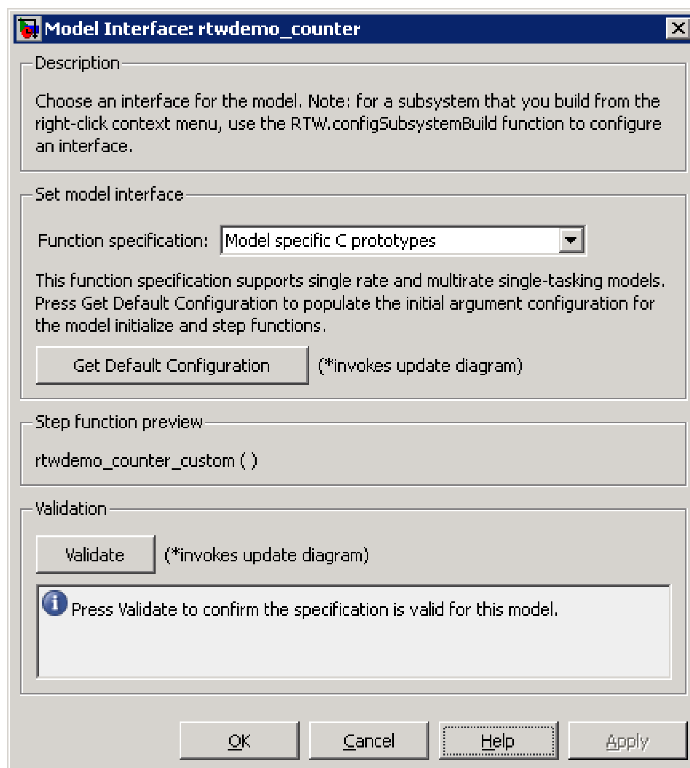
5. From the `rtwdemo_counter` model, open the Configuration Parameters dialog box.

6 Navigate to the **Code Generation > Interface** pane and click the **Configure Model Functions** button. The Model Interface dialog box appears.

7 In the initial (`Default model initialize and step funtions`) view of the Model Interface dialog box, click the **Validate** button to validate and preview the default function prototype for the `rtwdemo_counter_step` function. The function prototype arguments under **Step function preview** should correspond to the default prototype in step 4.



8 In the Model Interface dialog box, set **Function specification** field to `Model specific C prototypes`. Making this selection switches the dialog box from the `Default model initialize and step functions` view to the `Model specific C prototypes` view.



9 In the `Model specific C prototypes` view, click the **Get Default Configuration** button to activate the **Configure model initialize and step functions** subpane.

Model Interface: rtwdemo_counter

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: Model specific C prototypes

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: rtwdemo_counter_initialize

Step function name: rtwdemo_counter_custom

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	Input	Inport	Value	arg_Input	none
2	Output	Outport	Pointer	arg_Output	none

Up

Down

Step function preview

rtwdemo_counter_custom (arg_Input, * arg_Output)

Validation

Validate (*invokes update diagram)

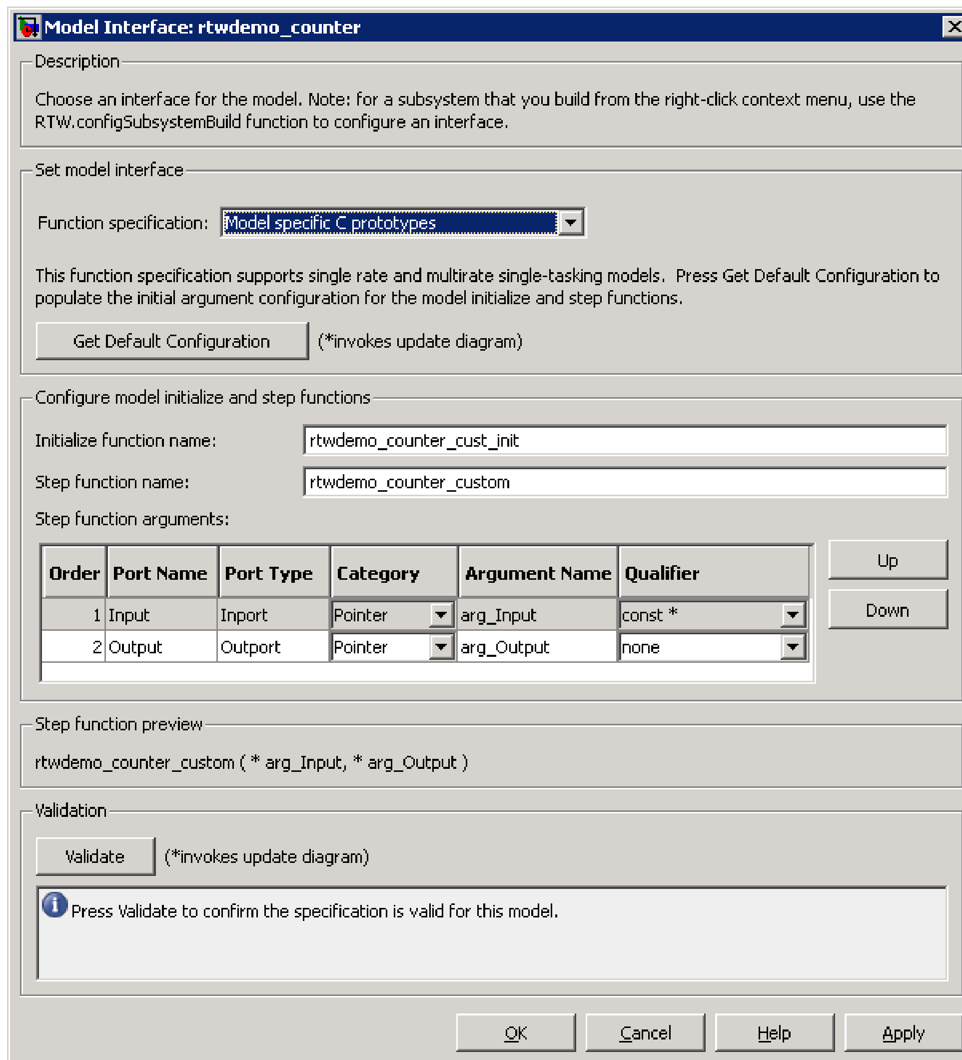
Press Validate to confirm the specification is valid for this model.

OK Cancel Help Apply

10 In the **Configure model initialize and step functions** subpane, change **Initialize function name** to `rtwdemo_counter_cust_init`.

11 In the **Configure model initialize and step functions** subpane, in the row for the `Input` argument, change the value of **Category** from `Value` to

`Pointer` and change the value of **Qualifier** from `none` to `const *`. The preview reflects your changes.



12 Click the **Validate** button to validate the modified function prototype. The **Validation** subpane displays a message that the validation succeeded.

13 Click **OK** to exit the Model Interface dialog box.

14 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears.

15 In the code generation report, click the link for `rtwdemo_counter.c`.

16 Locate and examine the generated code for the `rtwdemo_counter_custom` and `rtwdemo_counter_cust_init` functions:

```
/* Customized model step function */
void rtwdemo_counter_custom(const int32_T *arg_Input, int32_T *arg_Output)
{
 ...
}

 /* Model initialize function */
void rtwdemo_counter_cust_init(void)
{
 ...
}
```

17 Verify that the generated code is consistent with the function prototype modifications that you specified in the Model Interface dialog box.

Configuring Model Function Prototypes Programmatically

You can use the function prototype control functions (listed in Function Prototype Control Functions on page 29-20), to programmatically control model function prototypes. Typical uses of these functions include:

- **Create and validate a new function prototype**

  1. Create a model-specific C function prototype with *obj* = `RTW.ModelSpecificCPrototype`, where *obj* returns a handle to a newly created, empty function prototype.

  2. Add argument configuration information for your model ports using `RTW.ModelSpecificCPrototype.addArgConf`.

  3. Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.

  4. Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.

  5. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

- **Modify and validate an existing function prototype**

  1. Get the handle to an existing model-specific C function prototype that is attached to your loaded ERT-based Simulink model with *obj* = `RTW.getFunctionSpecification(`*modelName*`)`, where *modelName* is a string specifying the name of a loaded ERT-based Simulink model, and *obj* returns a handle to a function prototype attached to the specified model.

     You can use other function prototype control functions on the returned handle only if the test `isa(obj,'RTW.ModelSpecificCPrototype')` returns 1. If the model does not have a function prototype configuration, the function returns `[]`. If the function returns a handle to an object of type `RTW.FcnDefault`, you cannot modify the existing function prototype.

  2. Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 29-20 to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.

3. Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
4. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

- **Create and validate a new function prototype, starting with default configuration information from your Simulink model**

1. Create a model-specific C function prototype using *`obj`* `= RTW.ModelSpecificCPrototype`, where *`obj`* returns a handle to a newly created, empty function prototype.
2. Attach the function prototype to your loaded ERT-based Simulink model using `RTW.ModelSpecificCPrototype.attachToModel`.
3. Get default configuration information from your model using `RTW.ModelSpecificCPrototype.getDefaultConf`.
4. Use the `Get` and `Set` functions listed in Function Prototype Control Functions on page 29-20 to test and reset such items as the function names, argument names, argument positions, argument categories, and argument type qualifiers.
5. Validate the function prototype using `RTW.ModelSpecificCPrototype.runValidation`.
6. If validation succeeds, save your model and then generate code using the `rtwbuild` function.

Note You should not use the same model-specific C function prototype object across multiple models. If you do, changes that you make to the step and initialization function prototypes in one model are propagated to other models, which is usually not desirable.

Function Prototype Control Functions

Function	Description
RTW.ModelSpecificCPrototype.addArgConf	Add step function argument configuration information for Simulink model port to model-specific C function prototype
RTW.ModelSpecificCPrototype.attachToModel	Attach model-specific C function prototype to loaded ERT-based Simulink model
RTW.ModelSpecificCPrototype.getArgCategory	Get step function argument category for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgName	Get step function argument name for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgPosition	Get step function argument position for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getArgQualifier	Get step function argument type qualifier for Simulink model port from model-specific C function prototype
RTW.ModelSpecificCPrototype.getDefaultConf	Get default configuration information for model-specific C function prototype from Simulink model to which it is attached
RTW.ModelSpecificCPrototype.getFunctionName	Get function names from model-specific C function prototype
RTW.ModelSpecificCPrototype.getNumArgs	Get number of step function arguments from model-specific C function prototype
RTW.ModelSpecificCPrototype.getPreview	Get model-specific C function prototype code previews
RTW.configSubsystemBuild	Launch GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
RTW.getFunctionSpecification	Get handle to model-specific C function prototype object

Function Prototype Control Functions (Continued)

Function	Description
`RTW.ModelSpecificCPrototype.runValidation`	Validate model-specific C function prototype against Simulink model to which it is attached
`RTW.ModelSpecificCPrototype.setArgCategory`	Set step function argument category for Simulink model port in model-specific C function prototype
`RTW.ModelSpecificCPrototype.setArgName`	Set step function argument name for Simulink model port in model-specific C function prototype
`RTW.ModelSpecificCPrototype.setArgPosition`	Set step function argument position for Simulink model port in model-specific C function prototype
`RTW.ModelSpecificCPrototype.setArgQualifier`	Set step function argument type qualifier for Simulink model port in model-specific C function prototype
`RTW.ModelSpecificCPrototype.setFunctionName`	Set function names in model-specific C function prototype

Sample Script for Configuring Model Function Prototypes

The following sample MATLAB script configures the model function prototypes for the `rtwdemo_counter` model, using the Function Prototype Control Functions on page 29-20.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Create a model-specific C function prototype
a=RTW.ModelSpecificCPrototype

%% Add argument configuration information for Input and Output ports
addArgConf(a,'Input','Pointer','inputArg','const *')
addArgConf(a,'Output','Pointer','outputArg','none')

%% Attach the model-specific C function prototype to the model
attachToModel(a,gcs)

%% Rename the initialization function
setFunctionName(a,'InitFunction','init')

%% Rename the step function and change some argument attributes
setFunctionName(a,'StepFunction','step')
setArgPosition(a,'Output',1)
setArgCategory(a,'Input','Value')
setArgName(a,'Input','InputArg')
setArgQualifier(a,'Input','none')

%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

Verifying Generated Code for Customized Functions

You can use software-in-the-loop (SIL) testing to verify the generated code for your customized step and initialization functions. This involves creating a SIL block with your generated code, which then can be integrated into a Simulink model to verify that the generated code provides the same result as the original model or nonvirtual subsystem. For more information, see Chapter 26, “Generating S-Function Wrappers” and Chapter 39, “Verifying Generated Code With SIL and PIL Simulations”.

Model Function Prototype Control Limitations

The following limitations apply to controlling model function prototypes:

- Function prototype control supports only step and initialization functions generated from a Simulink model.
- Function prototype control supports only single-instance implementations. For standalone targets, you must clear the **Generate reusable code** check box (on the **Interface** pane of the Configuration Parameters dialog box). For model reference targets, you must select `One` for the **Total number of instances allowed per top model** parameter (on the **Model Referencing** pane of the Configuration Parameters dialog box).
- For model reference targets, the code generator ignores the **Generate reusable code** parameter (on the **Interface** pane of the Configuration Parameters dialog box).
- You must select the **Single output/update function** parameter (on the **Interface** pane of the Configuration Parameters dialog box).
- Function prototype control does not support multitasking models. Multirate models are supported, but you must configure the models for single-tasking.
- You must configure root-level inports and outports to use `Auto` storage classes.
- The generated code for a parent model does not call the function prototype control step functions generated from referenced models.
- Do not control function prototypes with the static `ert_main.c` provided by MathWorks. Specifying a function prototype control configuration other than the default creates a mismatch between the generated code and the default static `ert_main.c`.
- The code generator removes the data structure for the root inports of the model unless a subsystem implemented by a nonreusable function uses the value of one or more of the inports.
- The code generator removes the data structure for the root outports of the model except when you enable MAT-file logging, or if the sample time of one or more of the outports is not the fundamental base rate (including a constant rate).

- If you copy a subsystem block and paste it to create a new block in either a new model or the same model, the function prototype control interface information from the original subsystem block does not copy to the new subsystem block.
- For a Stateflow chart that uses a model root inport value, or that calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
  - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
  - Make the Stateflow function a nonreusable function.
  - Insert a Signal Conversion block immediately after the root inport and select the **Exclude this block from 'Block reduction' optimization** check box in the Signal Conversion block parameters.

30

Controlling Generation of Encapsulated C++ Model Interfaces

- “Overview of C++ Encapsulation” on page 30-2
- “C++ Encapsulation Quick-Start Example” on page 30-4
- “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 30-11
- “Configuring C++ Encapsulation Interfaces Programmatically” on page 30-21
- “Sample Script for Configuring the Step Method for a Model Class” on page 30-24
- “C++ Encapsulation Interface Control Limitations” on page 30-26

Overview of C++ Encapsulation

Using the **Language** option, `C++ (Encapsulated)`, on the **Code Generation** pane of the Configuration Parameters dialog box, you can generate a C++ class interface to model code. The generated interface encapsulates all required model data into C++ class attributes and all model entry point functions into C++ class methods. The benefits of encapsulation include:

- Greater control over access to model data
- Ability to multiply instantiate model classes
- Easier integration of model code into C++ programming environments

C++ encapsulation also works for right-click builds of nonvirtual subsystems. (For information on requirements that apply, see “Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems” on page 30-19.)

The general procedure for generating C++ encapsulation interfaces to model code is as follows:

1. Configure your model to use an `ert.tlc` system target file provided by MathWorks.

2. Select the language option `C++ (Encapsulated)` for your model.

3. Optionally, configure related C++ encapsulation interface settings for your model code, using either a graphical user interface (GUI) or application programming interface (API).

4. Generate model code and examine the results.

To get started with an example, see “C++ Encapsulation Quick-Start Example” on page 30-4. For more details about configuring C++ encapsulation interfaces for your model code, see “Generating and Configuring C++ Encapsulation Interfaces to Model Code” on page 30-11 and “Configuring C++ Encapsulation Interfaces Programmatically” on page 30-21. For limitations that apply, see “C++ Encapsulation Interface Control Limitations” on page 30-26.

Note For a demonstration of the C++ encapsulation capability, see the demo model `rtwdemo_cppencap`.

C++ Encapsulation Quick-Start Example

This example illustrates a simple use of the `C++ (Encapsulated)` option. It uses C++ encapsulation to generate interfaces for code from a demo model, without extensive modifications to default settings.

Note For details about setting C++ encapsulation options, see the sections that follow this example, beginning with "Generating and Configuring C++ Encapsulation Interfaces to Model Code" on page 30-11.

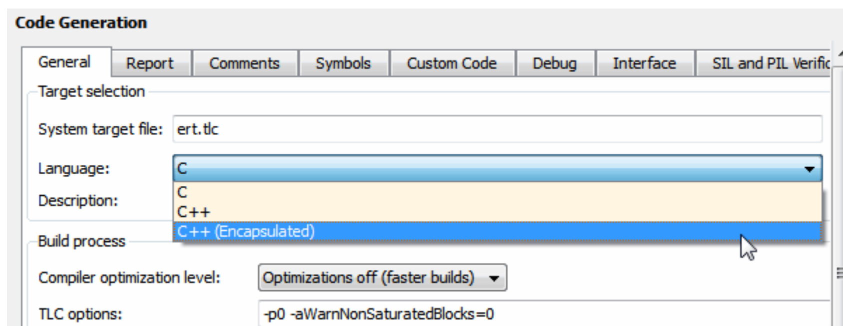
To generate C++ encapsulated interfaces for a Simulink model:

1. Open a model for which you would like to generate C++ encapsulation interfaces. This example uses the demo model `rtwdemo_counter`.

2. Configure the model to use an `ert.tlc` system target file provided by MathWorks. For example, open the Configuration Parameters dialog box, go to the **Code Generation** pane, select an appropriate target value from the **System target file** menu, and click **Apply**.

3. Optionally, as a baseline for later code comparison, generate code from the model using a different **Language** parameter setting, `C++` or `C`. (You can set up the build directory naming or location to distinguish your baseline build from later builds of the same model.)

4. On the **Code Generation** pane of the Configuration Parameters dialog box, select the `C++ (Encapsulated)` language option.

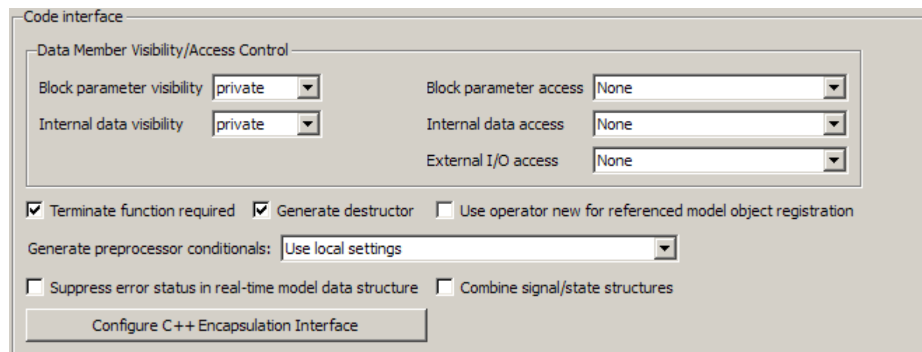


Click **Apply**.

Note To immediately generate the default style of encapsulated C++ code, without exploring the related model configuration options, skip steps 5–9 and go directly to step 10.

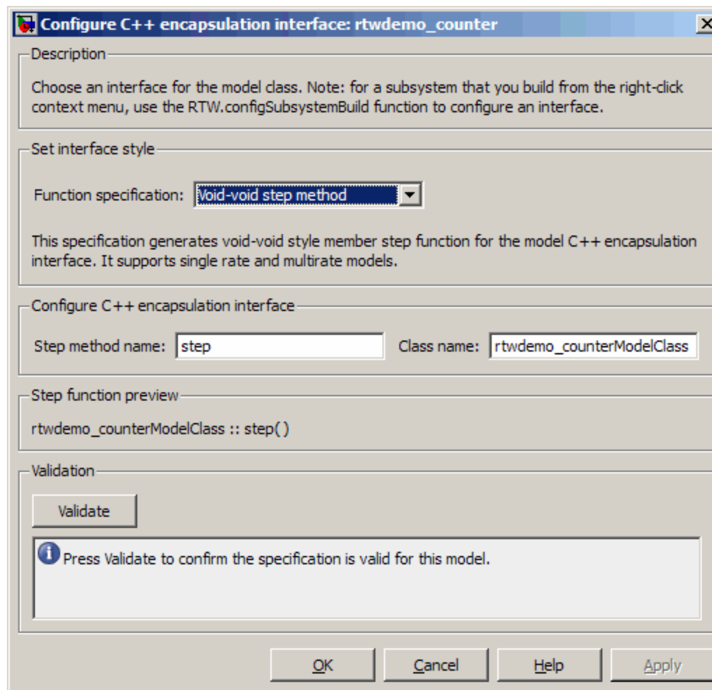
---

5 Go to the **Interface** pane of the Configuration Parameters dialog box and examine the **Code interface** subpane.



When you selected the `C++ (Encapsulated)` language option for your model, C++ encapsulation interface controls replaced the default options on the **Code interface** subpane. See "Configuring Code Interface Options" on page 30-12 for descriptions of these controls. Examine the default settings and modify as appropriate.

6 Click the **Configure C++ Encapsulation Interface** button. This action opens the Configure C++ encapsulation interface dialog box, which allows you to configure the step method for your generated model class. The dialog box initially displays a view for configuring a `void-void` style step method (passing no I/O arguments) for the model class. In this view, you can rename the model class and the step method for your model.

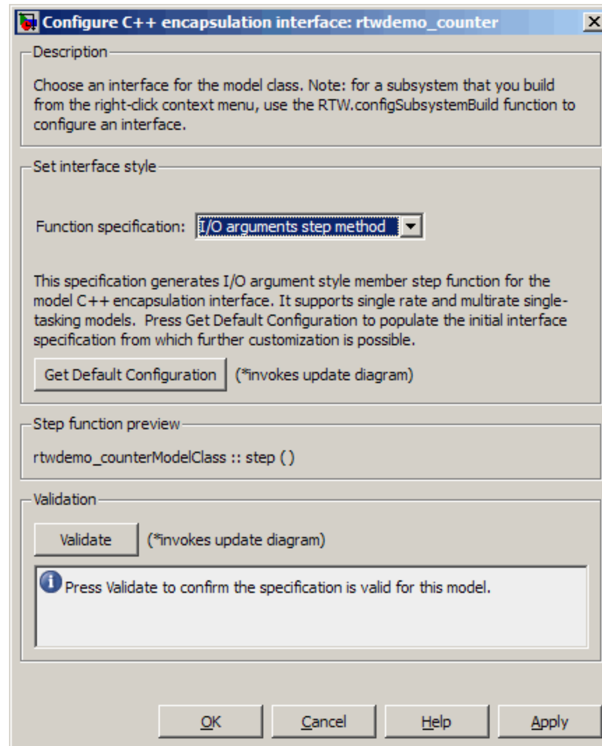


See "Configuring the Step Method for Your Model Class" on page 30-15 for descriptions of these controls.

Note If the `void-void` interface style meets your needs, you can skip steps 7–9 and go directly to step 10.

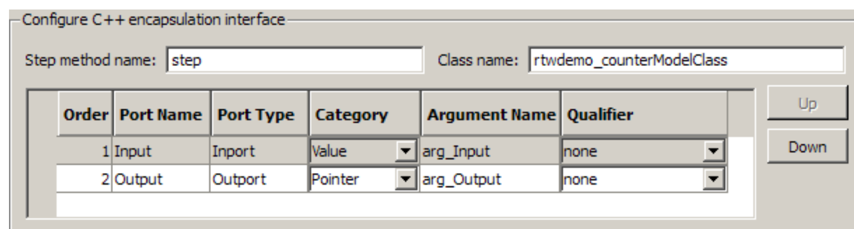
---

7 If you want root-level model input and output to be arguments on the step method, select the value `I/O arguments step method` from the **Function specification** menu. The dialog box displays a view for configuring an I/O arguments style step method for the model class.



See “Configuring the Step Method for Your Model Class” on page 30-15 for descriptions of these controls.

8 Click the **Get Default Configuration** button. This action causes a **Configure C++ encapsulation interface** subpane to appear in the dialog box. The subpane displays the initial interface configuration for your model, which provides a starting point for further customization.



See "Passing I/O Arguments" on page 30-17 for descriptions of these controls.

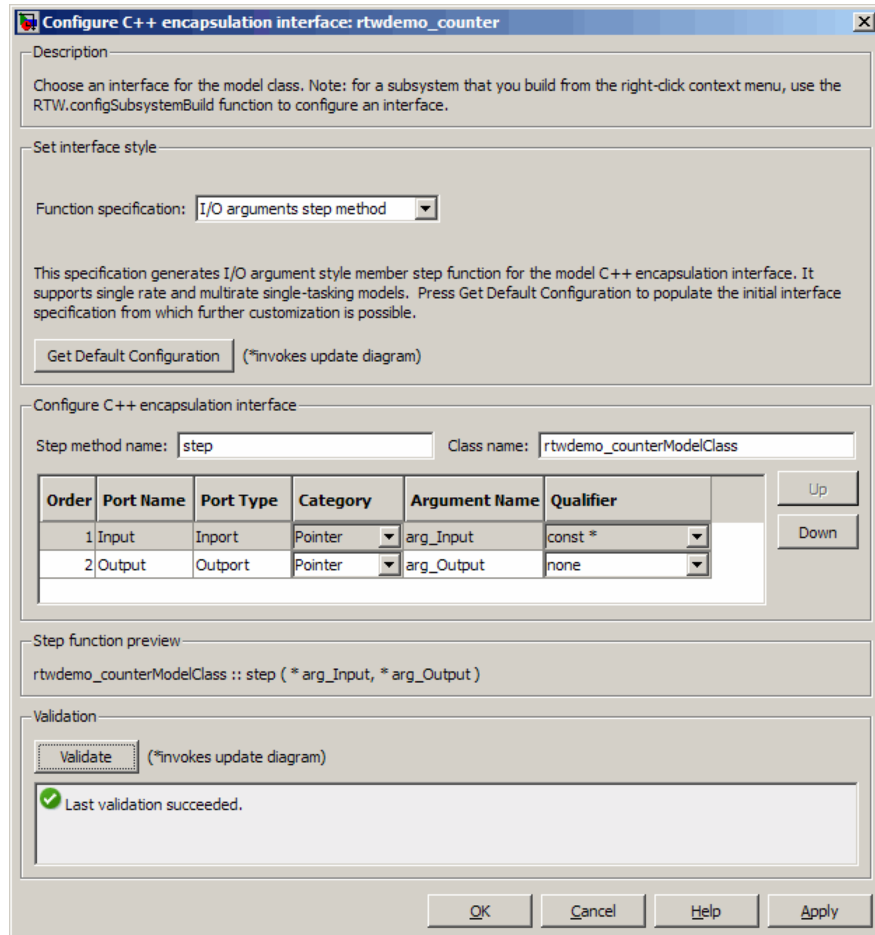
9 Perform this optional step only if you want to customize the configuration of the I/O arguments generated for your model step method.

Note If you choose to skip this step, you should click **Cancel** to exit the dialog box.

If you choose to perform this step, first you must check that the required option **Remove root level I/O zero initialization** is selected on the **Optimization** pane, and then navigate back to the `I/O arguments step method` view of the Configure C++ encapsulation interface dialog box.

Now you can use the dialog box controls to configure I/O argument attributes. For example, in the **Configure C++ encapsulation interface** subpane, in the row for the `Input` argument, you can change the value of **Category** from `Value` to `Pointer` and change the value of **Qualifier** from `none` to `const *`. The preview updates to reflect your changes. Click the **Validate** button to validate the modified interface configuration.

Continue modifying and validating until you are satisfied with the step method configuration.



Click **Apply** and **OK**.

10 Generate code for the model. When the build completes, the code generation report for `rtwdemo_counter` appears. Examine the report and observe that all required model data is encapsulated into C++ class attributes and all model entry point functions are encapsulated into C++ class methods. For example, click the link for `rtwdemo_counter.h` to see the class declaration for the model.

Back Forward

Contents

Summary

Subsystem Report

Traceability Report

Generated Source Files

ert_main.cpp

rtwdemo_counter.cpp

rtwdemo_counter.h

rtwdemo_counter_private.h

rtwdemo_counter_types.h

rtwtypes.h

```
58  /* Class declaration for model rtwdemo_counter */
59  class rtwdemo_counterModelClass {
60    /* public data and function members */
61   public:
62    /* Model entry point functions */
63
64    /* model initialize function */
65    void initialize();
66
67    /* model step function */
68    void step(const int32_T *arg_Input, int32_T *arg_Output);
69
70    /* model terminate function */
71    void terminate();
72
73    /* Constructor */
74    rtwdemo_counterModelClass();
75
76    /* Destructor */
77    ~rtwdemo_counterModelClass();
78
79    /* Real-Time Model get method */
80    RT_MODEL_rtwdemo_counter * getRTM() const;
81
82    /* private data and function members */
83   private:
84    /* Block signals */
85    BlockIO_rtwdemo_counter rtwdemo_counter_B;
86
87    /* Block states */
88    D_Work_rtwdemo_counter rtwdemo_counter_DWork;
89    PrevZCSigStates_rtwdemo_counter rtwdemo_counter_PrevZCSigState;/* Trig
90
91    /* Real-Time Model */
92    RT_MODEL_rtwdemo_counter rtwdemo_counter_M_;
93
94    /* Real-Time Model pointer */
95    RT_MODEL_rtwdemo_counter *rtwdemo_counter_M;
96  };
```

OK Help

Note

- If you configured custom I/O arguments for the model step method (optional step 9), examine the generated code for the step method in `rtwdemo_counter.h` and `rtwdemo_counter.cpp`. The arguments should reflect your changes. For example, if you performed the `Input` argument modifications in step 9, the input argument should appear as `const int32_T *arg_Input`.
- If you saved a baseline model build (optional step 3), you can traverse and compare the generated files in the corresponding build directories.

Generating and Configuring C++ Encapsulation Interfaces to Model Code

In this section...

"Selecting the C++ (Encapsulated) Option" on page 30-11

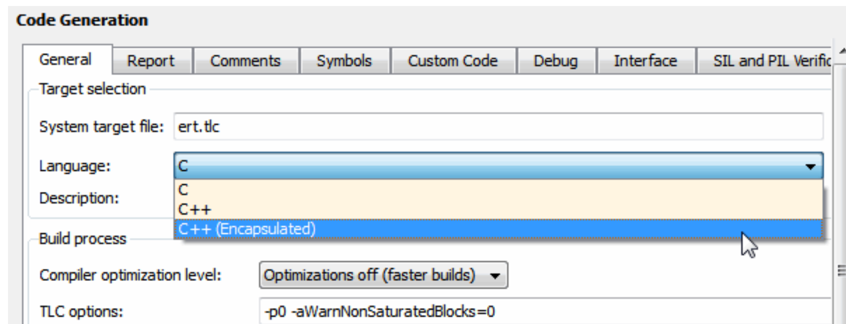
"Configuring Code Interface Options" on page 30-12

"Configuring the Step Method for Your Model Class" on page 30-15

"Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems" on page 30-19

Selecting the C++ (Encapsulated) Option

To select the `C++ (Encapsulated)` option, in the Configuration Parameters dialog box, on the **Code Generation** pane, use the **Language** menu:



When you select this option, you see the following effects on other panes in the Configuration Parameters dialog box:

- Disables model configuration options that `C++ (Encapsulated)` does not support. For details, see "C++ Encapsulation Interface Control Limitations" on page 30-26.
- Replaces the default options on the **Interface** pane, in the **Code interface** subpane, with C++ encapsulation interface controls, which are described in the next section.

Configuring Code Interface Options

When you select the `C++ (Encapsulated)` option for your model, the C++ encapsulation interface controls shown below replace the **Code interface** default options on the **Interface** pane.

- **Block parameter visibility**

  Specifies whether to generate the block parameter structure as a `public`, `private`, or `protected` data member of the C++ model class (`private` by default).

- **Internal data visibility**

  Specifies whether to generate internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, as `public`, `private`, or `protected` data members of the C++ model class (`private` by default).

- **Block parameter access**

  Specifies whether to generate access methods for block parameters for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **Internal data access**

  Specifies whether to generate access methods for internal data structures, such as Block I/O, DWork vectors, Runtime model, Zero-crossings, and continuous states, for the C++ model class (`None` by default). You can select noninlined access methods (`Method`) or inlined access methods (`Inlined method`).

- **External I/O access**

  Specifies whether to generate access methods for root-level I/O signals for the C++ model class (`None` by default). If you want to generate access methods, you have the following options:

  - Generate either noninlined or inlined access methods.
  - Generate either *per-signal* or *structure-based* access methods. That is, you can generate a series of set and get methods on a per-signal basis, or generate just one set method that takes the address of an external input structure as an argument and, for external outputs (if applicable), just one get method that returns a reference to an external output structure. The generated code for structure-based access methods has the following general form:

```
class ModelClass {
...
   /* Root inports set method*/
   void setExternalInputs(const ExternalInputs* pExternalInputs);
   /* Root outports get method*/
   const ExternalOutputs & getExternalOutputs() const;
}
```

Note This parameter affects generated code only if you are using the default (`void-void` style) step method for your model class; *not* if you are explicitly passing arguments for root-level I/O signals using an I/O arguments style step method. For more information, see “Passing No Arguments (void-void)” on page 30-15 and “Passing I/O Arguments” on page 30-17.

- **Terminate function**

  Specifies whether to generate the *`model`*`_terminate` function (on by default). This function contains all model termination code and should be called as part of system shutdown.

- **Generate destructor**

  Specifies whether to generate a destructor for the C++ model class (on by default).

- **Use operator new for referenced model object registration**

  For a model containing Model blocks, specifies whether generated code should use dynamic memory allocation, during model object registration, to instantiate objects for referenced models configured with a C++ encapsulation interface (off by default). If you select this option, during instantiation of an object for the top model in a model reference hierarchy, the generated code uses the operator `new` to instantiate objects for referenced models.

  Selecting this option frees a parent model from having to maintain information about submodels beyond its direct children. Clearing this option means that a parent model maintains information about all of its submodels, including its direct and indirect children.

  Note If you select this option, be aware that a `bad_alloc` exception might be thrown, per the C++ standard, if an out-of-memory error occurs during the use of `new`. You must provide code to catch and process the `bad_alloc` exception in case an out-of-memory error occurs for a `new` call during construction of a top model object.

- **Generate preprocessor conditionals**

  For a model containing Model blocks, specifies whether to generate preprocessor conditional directives globally for a model, locally for each variant Model block, or conditionally based on the **Generate preprocessor conditionals** setting in the Model Reference Parameter dialog for each variant Model block (`Use local settings` by default).

- **Suppress error status in real-time model data structure**

  Specifies whether to omit the error status field from the generated real-time model data structure `rtModel` (off by default). Selecting this option reduces memory usage.

  Be aware that selecting this option can cause the code generator to omit the `rtModel` data structure from generated code.

- **Combine signal/state structures**

Specifies whether to combine global block signals and global state data into one data structure in the generated code (off by default). Selecting this option reduces RAM and improves readability of the generated code.

- **Configure C++ Encapsulation Interface**

  Opens the Configure C++ encapsulation interface dialog box, which allows you to configure the step method for your model class. For more information, see "Configuring the Step Method for Your Model Class" on page 30-15.

Configuring the Step Method for Your Model Class

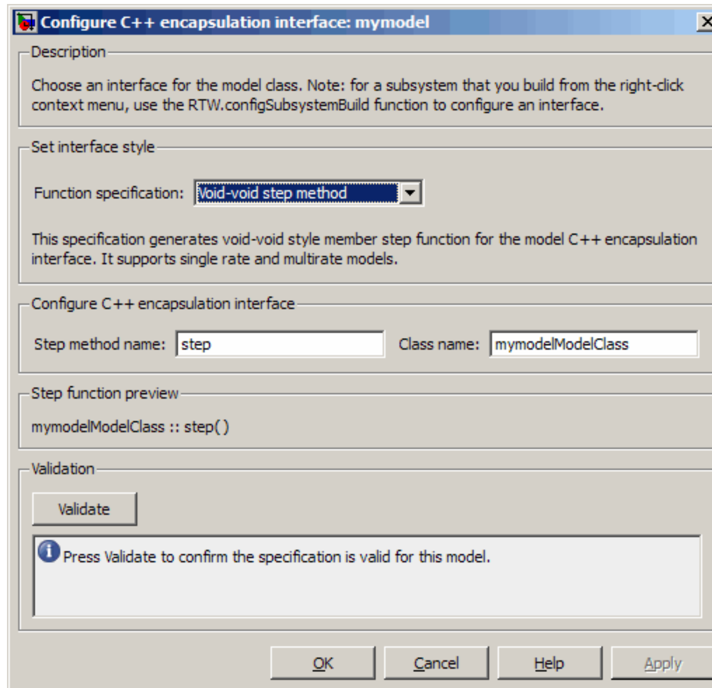
To configure the step method for your model class, on the **Interface** pane, click the **Configure C++ Encapsulation Interface** button, which is available when you select`C++ (Encapsulated)` for your model. This action opens the Configure C++ encapsulation interface dialog box, where you can configure the step method for your model class in either of two styles:

- "Passing No Arguments (void-void)" on page 30-15
- "Passing I/O Arguments" on page 30-17

Note The `void-void` style of step method specification supports single-rate models and multirate models, while the I/O arguments style supports single-rate models and multirate single-tasking models.

Passing No Arguments (void-void)

The Configure C++ encapsulation interface dialog box initially displays a view for configuring a `void-void` style step method for the model class.



- **Step method name**

  Allows you to specify a step method name other than the default, `step`.

- **Class name**

  Allows you to specify a model class name other than the default, *`model`*`ModelClass`.

- **Step function preview**

  Displays a preview of the model step function prototype as currently configured. The preview display is dynamically updated as you make configuration changes.

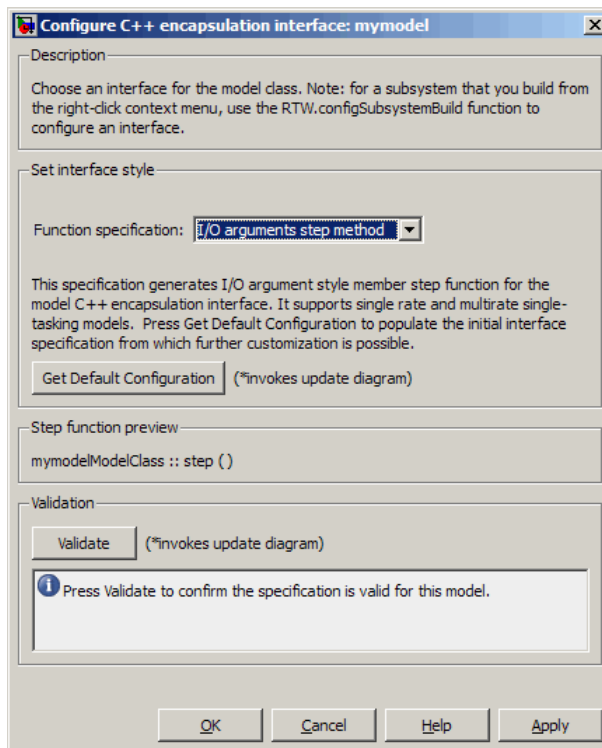
- **Validate**

  Validates your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure.

Passing I/O Arguments

If you select `I/O arguments step method` from the **Function specification** menu, the dialog box displays a view for configuring an I/O arguments style step method for the model class.

Note To use the I/O arguments style step method, you must select the option **Remove root level I/O zero initialization** on the **Optimization** pane of the Configuration Parameters dialog box.



- **Get Default Configuration**

  Click this button to get the initial interface configuration that provides a starting point for further customization.

- **Step function preview**

  Displays a preview of the model step function prototype as currently configured. The preview dynamically updates as you make configuration changes.

- **Validate**

  Validates your current model step function configuration. The **Validation** pane displays success or failure status and an explanation of any failure.

When you click **Get Default Configuration**, the **Configure C++ encapsulation interface** subpane appears in the dialog box, displaying the initial interface configuration. For example:

Configure C++ encapsulation interface

Step method name: step Class name: mymodelModelClass

Order	Port Name	Port Type	Category	Argument Name	Qualifier
1	In1	Inport	Value	arg_In1	none
2	In2	Inport	Value	arg_In2	none
3	In3	Inport	Value	arg_In3	none
4	Out1	Outport	Pointer	arg_Out1	none
5	Out2	Outport	Pointer	arg_Out2	none

Up Down

- **Step method name**

  Allows you to specify a step method name other than the default, `step`.

- **Class name**

  Allows you to specify a model class name other than the default, *`model`*`ModelClass`.

- **Order**

  Displays the numerical position of each argument. Use the **Up** and **Down** buttons to change argument order.

- **Port Name**

  Displays the port name of each argument (not configurable using this dialog box).

- **Port Type**

Displays the port type, `Inport` or `Outport`, of each argument (not configurable using this dialog box).

- **Category**

  Displays the passing mechanism for each argument. To change the passing mechanism for an argument, select `Value`, `Pointer`, or `Reference` from the argument's **Category** menu.

- **Argument Name**

  Displays the name of each argument. To change an argument name, click in the argument's **Argument name** field, position the cursor for text entry, and enter the new name.

- **Qualifier**

  Displays the `const` type qualifier for each argument. To change the qualifier for an argument, select an available value from the argument's **Qualifier** menu. The possible values are:

  - `none`
  - `const` (value)
  - `const*` (value referenced by the pointer)
  - `const*const` (value referenced by the pointer and the pointer itself)
  - `const &` (value referenced by the reference)

Tip When a model includes a referenced model, the `const` type qualifier for the root input argument of the referenced model's specified step function interface is set to `none` and the qualifier for the source signal in the referenced model's parent is set to a value other than `none`, code generation honors the referenced model's interface specification by generating a type cast that discards the `const` type qualifier from the source signal. To override this behavior, add a `const` type qualifier to the referenced model.

Configuring C++ Encapsulation Interfaces for Nonvirtual Subsystems

C++ encapsulation interfaces can be configured for right-click builds of nonvirtual subsystems in Simulink models, provided that:

- You select the system target file `ert.tlc` for the model.
- You select the **Language** parameter value `C++ (Encapsulated)` for the model.
- The subsystem is convertible to a Model block using the function `Simulink.SubSystem.convertToModelReference`. For referenced model conversion requirements, see the Simulink reference page `Simulink.SubSystem.convertToModelReference`.

To configure C++ encapsulation interfaces for a subsystem that meets the requirements:

1. Open the containing model and select the subsystem block.

2. Enter the following MATLAB command:

   ```
   RTW.configSubsystemBuild(gcb)
   ```

   where `gcb` is the Simulink function `gcb`, returning the full block path name of the current block.

   This command opens a subsystem equivalent of the Configure C++ encapsulation interface dialog sequence that is described in detail in the preceding section, "Configuring the Step Method for Your Model Class" on page 30-15. (For more information about using the MATLAB command, see `RTW.configSubsystemBuild`.)

3. Use the Configure C++ encapsulation interface dialog boxes to configure C++ encapsulation settings for the subsystem.

4. Right-click the subsystem and select **Code Generation > Build Subsystem**.

5. When the subsystem build completes, you can examine the C++ encapsulation interfaces in the generated files and the HTML code generation report.

Configuring C++ Encapsulation Interfaces Programmatically

If you select the **Language** option `C++ (Encapsulated)` for your model, you can use the C++ encapsulation interface control functions (listed in C++ Encapsulation Interface Control Functions on page 30-22) to programmatically configure the step method for your model class.

Typical uses of these functions include:

- **Create and validate a new step method interface, starting with default configuration information from your Simulink model**

  1. Create a model-specific C++ encapsulation interface with `obj = RTW.ModelCPPVoidClass` or `obj = RTW.ModelCPPArgsClass`, where `obj` returns a handle to an newly created, empty C++ encapsulation interface.
  2. Attach the C++ encapsulation interface to your loaded ERT-based Simulink model using `attachToModel`.
  3. Get default C++ encapsulation interface configuration information from your model using `getDefaultConf`.
  4. Use the `Get` and `Set` functions listed in C++ Encapsulation Interface Control Functions on page 30-22 to test or reset the model class name and model step method name. Additionally, if you are using the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.
  5. Validate the C++ encapsulation interface using `runValidation`. (If validation fails, use the error message information that`runValidation` returns to address the issues.)
  6. Save your model and then generate code using the `rtwbuild` function.

- **Modify and validate an existing step method interface for a Simulink model**

  1. Get the handle to an existing model-specific C++ encapsulation interface that is attached to your loaded ERT-based Simulink model using `obj = RTW.getEncapsulationInterfaceSpecification(modelName)`, where `modelName` is a string specifying the name of a loaded ERT-based

Simulink model, and *obj* returns a handle to a C++ encapsulation interface attached to the specified model. If the model does not have an attached C++ encapsulation interface configuration, the function returns `[]`.

2. Use the `Get` and `Set` functions listed in C++ Encapsulation Interface Control Functions on page 30-22 to test or reset the model class name and model step method name. Additionally, if the returned interface uses the I/O arguments style step method, you can test and reset argument names, argument positions, argument categories, and argument type qualifiers.

3. Validate the C++ encapsulation interface using `runValidation`. (If validation fails, use the error message information that `runValidation` returns to address the issues.)

4. Save your model and then generate code using the `rtwbuild` function.

Note You should not use the same model-specific C++ encapsulation interface control object across multiple models. If you do, changes that you make to the step method configuration in one model propagate to other models, which is usually not desirable.

C++ Encapsulation Interface Control Functions

Function	Description
`attachToModel`	Attach model-specific C++ encapsulation interface to loaded ERT-based Simulink model
`getArgCategory`	Get argument category for Simulink model port from model-specific C++ encapsulation interface
`getArgName`	Get argument name for Simulink model port from model-specific C++ encapsulation interface
`getArgPosition`	Get argument position for Simulink model port from model-specific C++ encapsulation interface
`getArgQualifier`	Get argument type qualifier for Simulink model port from model-specific C++ encapsulation interface

C++ Encapsulation Interface Control Functions (Continued)

Function	Description
getClassName	Get class name from model-specific C++ encapsulation interface
getDefaultConf	Get default configuration information for model-specific C++ encapsulation interface from Simulink model to which it is attached
getNumArgs	Get number of step method arguments from model-specific C++ encapsulation interface
getStepMethodName	Get step method name from model-specific C++ encapsulation interface
RTW.configSubsystemBuild	Open GUI to configure C function prototype or C++ encapsulation interface for right-click build of specified subsystem
RTW.getEncapsulation-InterfaceSpecification	Get handle to model-specific C++ encapsulation interface control object
runValidation	Validate model-specific C++ encapsulation interface against Simulink model to which it is attached
setArgCategory	Set argument category for Simulink model port in model-specific C++ encapsulation interface
setArgName	Set argument name for Simulink model port in model-specific C++ encapsulation interface
setArgPosition	Set argument position for Simulink model port in model-specific C++ encapsulation interface
setArgQualifier	Set argument type qualifier for Simulink model port in model-specific C++ encapsulation interface
setClassName	Set class name in model-specific C++ encapsulation interface
setStepMethodName	Set step method name in model-specific C++ encapsulation interface

Sample Script for Configuring the Step Method for a Model Class

The following sample MATLAB script configures the step method for the `rtwdemo_counter` model class, using the C++ Encapsulation Interface Control Functions on page 30-22.

```
%% Open the rtwdemo_counter model
rtwdemo_counter

%% Select ert.tlc as the System Target File for the model
set_param(gcs,'SystemTargetFile','ert.tlc')

%% Select C++ (Encapsulated) as the target language for the model
set_param(gcs,'TargetLang','C++ (Encapsulated)')

%% Set required option for I/O arguments style step method (cmd off = GUI on)
set_param(gcs,'ZeroExternalMemoryAtStartup','off')

%% Create a C++ encapsulated interface using an I/O arguments style step method
a=RTW.ModelCPPArgsClass

%% Attach the C++ encapsulated interface to the model
attachToModel(a,gcs)

%% Get the default C++ encapsulation interface configuration from the model
getDefaultConf(a)

%% Move the Output port argument from position 2 to position 1
setArgPosition(a,'Output',1)

%% Reset the model step method name from step to StepMethod
setStepMethodName(a,'StepMethod')

%% Change the Input port argument name, category, and qualifier
setArgName(a,'Input','inputArg')
setArgCategory(a,'Input','Pointer')
setArgQualifier(a,'Input','const *')
```

```
%% Validate the function prototype against the model
[status,message]=runValidation(a)

%% if validation succeeded, generate code and build
if status
    rtwbuild(gcs)
end
```

C++ Encapsulation Interface Control Limitations

- The `C++ (Encapsulated)` option does not support some Simulink model configuration options. Selecting `C++ (Encapsulated)` disables the following items in the Configuration Parameters dialog box:
  - **Identifier format control** subpane on the **Symbols** pane
  - **Templates** pane
    - The **Templates** pane parameter **File customization template** is not supported for `C++ (Encapsulated)` code generation.
    - Selecting `C++ (Encapsulated)` turns on the **Templates** pane option **Generate an example main program** but removes it from the Configuration Parameters dialog box. If desired, you can disable it using the command line parameter `GenerateSampleERTMain`.
  - **Code Placement** pane
  - **Memory Sections** pane

Note Selecting `C++ (Encapsulated)` forces on the **Code Generation** pane model option **Ignore custom storage classes**. By design, `C++ (Encapsulated)` code generation treats data objects with custom storage classes as if their storage class attribute is set to `Auto`.

- Among the data exchange interfaces available on the **Interface** pane of the Configuration Parameters dialog box, only the `C API` interface is supported for `C++ (Encapsulated)` code generation. If you select `External mode` or `ASAP2`, code generation fails with a validation error.
- The I/O arguments style of step method specification supports single-rate models and multirate single-tasking models, but not multirate multitasking models.
- The `C++ (Encapsulated)` option does not support use of the `IncludeERTFirstTime` model option to include the `firstTime` argument in the *`model`*`_initialize` function generated for an ERT-based models. (The `IncludeERTFirstTime` option is off by default except for models created with R2006a.) Also, the `C++ (Encapsulated)` option requires that the target selected for the model support `firstTime` argument control by setting

the `ERTFirstTimeCompliant` target option, which all targets provided by MathWorks do by default. In other words, the `C++ (Encapsulated)` option requires that the target option `ERTFirstTimeCompliant` is on and the model option `IncludeERTFirstTime` is off.

- The **Code Generation > Export Functions** capability does not support `C++ (Encapsulated)` as the target language.
- For a Stateflow chart that resides in a root model configured to use the `I/O arguments step method` function specification, and that uses a model root inport value or calls a subsystem that uses a model root inport value, you must do one of the following to generate code:
  - Clear the **Execute (enter) Chart At Initialization** check box in the Stateflow chart.
  - Insert a Signal Conversion block immediately after the root inport and select the **Exclude this block from 'Block reduction' optimization** check box in the Signal Conversion block parameters.
- When building a referenced model that is configured to generate a C++ encapsulation interface:
  - You must use the `I/O arguments step method` style of the C++ encapsulated interface. The `void-void step method` style is not supported for referenced models.
  - You cannot use a C++ encapsulation interface in cases when a referenced model cannot have a combined output/update function. Cases include a model that
    - Has multiple sample times
    - Has a continuous sample time
    - Saves states

31

Replacing Math Functions and Operators Using Target Function Libraries

- "Introduction to Target Function Libraries" on page 31-2
- "Creating Function Replacement Tables" on page 31-16
- "Examining and Validating Function Replacement Tables" on page 31-139
- "Registering Target Function Libraries" on page 31-148
- "Target Function Library Limitations" on page 31-156

Introduction to Target Function Libraries

In this section...

"Overview of Target Function Libraries" on page 31-2

"Target Function Libraries General Workflow" on page 31-7

"Target Function Libraries Quick-Start Example" on page 31-9

Overview of Target Function Libraries

The Embedded Coder software provides the target function library (TFL) API, which allows you to create and register function replacement tables. When selected for a model, these TFL tables provide the basis for replacing default math functions and operators in your model code with target-specific code. The ability to control function and operator replacements potentially allows you to optimize target performance (speed and memory) and better integrate model code with external and legacy code.

A *target function library* (TFL) is a set of one or more function replacement tables that define the target-specific implementations of math functions and operators to be used in generating code for your Simulink model. The code generation software provides default TFLs, described in the following table. You select these TFLs from the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

TFL	Description	Contains Tables...
C89/C90 (ANSI)	Generates calls to the ISO®/IEC 9899:1990 C standard math library for floating-point functions.	ansi_tfl_table_tmw.mat
C99 (ISO)	Generates calls to the ISO/IEC 9899:1999 C standard math library.	iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat
GNU99 (GNU)	Generates calls to the GNU®[7] gcc math library, which provides C99 extensions as defined by compiler option -std=gnu99.	gnu_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat

7. GNU® is a registered trademark of the Free Software Foundation.

TFL	Description	Contains Tables...
C++ (ISO)	Generates calls to the ISO/IEC 14882:2003 C++ standard math library.	iso_cpp_tfl_table_tmw.mat private_iso_cpp_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat

When a TFL contains multiple tables, the order in which they are listed reflects the order in which they are searched. The TFL API allows you to create your own TFLs, made up of your own function tables in combination with one of the default TFLs. For example, you could create a TFL for an embedded processor that combines some special-purpose function customizations with a processor-specific library of function and operator implementations:

MyProcessor (ANSI)	Generates calls to my custom function implementations or a processor-specific library.	tfl_table_sinfcns.m tfl_table_myprocessor.m ansi_tfl_table_tmw.mat

Each TFL function replacement table contains one or more table entries, with each table entry representing a potential replacement for a single math function or an operator. Each table entry provides a mapping between a *conceptual view* of the function or operator (similar to the Simulink block view of the function or operator) and a *target-specific implementation* of that function or operator.

The conceptual view of a function or operator is represented in a TFL table entry by the following elements, which identify the function or operator entry to the code generation process:

- A function or operator key (a function name such as `'cos'` or an operator ID string such as `'RTW_OP_ADD'`)
- A set of conceptual arguments that observe a Simulink naming scheme (`'y1'`, `'u1'`, `'u2'`, ...), along with their I/O types (output or input) and data types
- Other attributes, such as fixed-point saturation and rounding characteristics for operators, as needed to identify the function or operator

to the code generation process as exactly as you require for matching purposes

The target-specific implementation of a function or operator is represented in a TFL table entry by the following elements:

- The name of your implementation function (such as `'cos_dbl'` or `'u8_add_u8_u8'`)
- A set of implementation arguments that you define (the order of which must correspond to the conceptual arguments), along with their I/O types (output or input) and data types
- Parameters providing the build information for your implementation function, including header file and source file names and paths

Additionally, a TFL table entry includes a priority value (0–100, with 0 as the highest priority), which defines the entry's priority relative to other entries in the table.

During code generation for your model, when the code generation process encounters a call site for a math function or operator, it creates and partially populates a TFL entry object, for the purpose of querying the TFL for a replacement function. The information provided for the TFL query includes the function or operator key and the conceptual argument list. The TFL entry object is then passed to the TFL. If there is a matching table entry in the TFL, a fully-populated TFL entry, including the implementation function name, argument list, and build information, is returned to the call site and used to generate code.

Within the TFL that is selected for your model, the tables that comprise the TFL are searched in the order in which they are listed (by `RTW.viewTFL` or by the TFL's **Target function library** tool tip). Within each table, if multiple matches are found for a TFL entry object, priority level determines the match that is returned. A higher-priority (lower-numbered) entry is used over a similar entry with a lower priority (higher number).

The TFL API supports the following functions for replacement with custom library functions using TFL tables:

Math Functions			
Note For detailed support information, see "Example: Mapping Math Functions to Target-Specific Implementations" on page 31-27.			
abs	cos	log	saturate
acos	cosh	log10	sign
acosh	exactrSqrt	max	sin
asin	exp	min	sinh
asinh	fix	mod/fmod	sqrt
atan	floor	pow	tan
atan2	hypot	rem	tanh
atanh	ldexp	round	
ceil	ln	rSqrt	
Memory Utility Functions			
memcmp	memcpy	memset	memset2zero[8]
Nonfinite Support Utility Functions			
getInf	getMinusInf	getNaN	

The TFL API also supports the following operations for replacement with custom library functions using TFL tables:

8. Some target processors provide optimized `memset` functions for use when performing a memory set to zero. The TFL API supports replacing `memset` to zero functions with more efficient target-specific functions.

Note Unless otherwise stated, the listed operators are supported for the following input data types:

- `single`, `double`, and their complex equivalents
- `int8`, `int16`, `int32`, and their complex equivalents
- `uint8`, `uint16`, `uint32`, and their complex equivalents
- Fixed-point data types
- Mixed data types (different types for different inputs)

Operator	Key	Scalar Inputs	Nonscalar Inputs
Addition (+)	`RTW_OP_ADD`	Yes	Yes
Subtraction (-)	`RTW_OP_MINUS`	Yes	Yes
Multiplication (*, .*)	`RTW_OP_MUL`	Yes	Yes, including the ability to map to Basic Linear Algebra Subroutine (BLAS) multiplication functions
Division (/)	`RTW_OP_DIV`	Yes	—
Data type conversion (cast)	`RTW_OP_CAST`	Yes	—
Shift left (<<)	`RTW_OP_SL`	Yes, for integer and fixed-point data types	—

Operator	Key	Scalar Inputs	Nonscalar Inputs
Shift right (>>)	RTW_OP_SRA (arithmetic) [9] RTW_OP_SRL (logical)	Yes, for integer and fixed-point data types	—
Complex conjugation	RTW_OP_CONJUGATE	Yes	Yes
Transposition (.')	RTW_OP_TRANS	—	Yes
Hermitian (complex conjugate) transposition (')	RTW_OP_HERMITIAN	—	Yes
Multiplication with transposition	RTW_OP_TRMUL	—	Yes, including the ability to map to BLAS multiplication functions
Multiplication with Hermitian transposition	RTW_OP_HMMUL	—	Yes, including the ability to map to BLAS multiplication functions

Target Function Libraries General Workflow

The general steps for creating and using a target function library are as follows:

1 Create one or more TFL tables containing replacement entries for math operators (+, –, *, /) and functions using an API based on the MATLAB API.

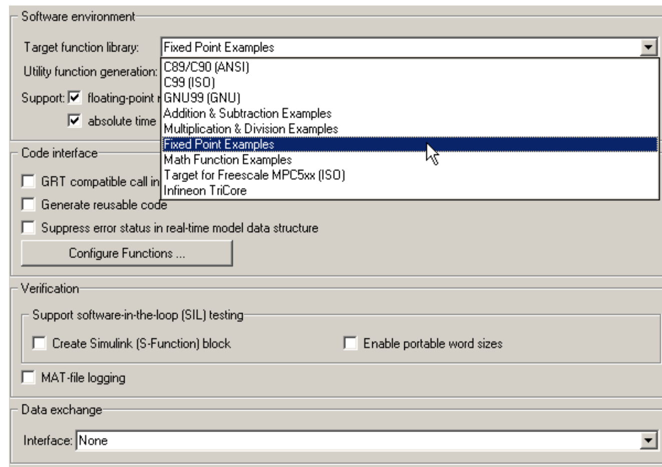
9. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

(The demo `rtwdemo_tfl_script` provides example tables that can be used as a starting point for customization.)

2. Register a target function library, consisting of one or more replacement tables, for use with Simulink or MATLAB Coder software. The MATLAB APIs `sl_customization` and `rtwTargetInfo` are provided for this purpose.

3. Open your Simulink model and select the desired target function library from the **Target function library** drop-down list located on the **Interface** pane of the Configuration Parameters dialog box. For MATLAB Coder applications, instantiate a Simulink Coder configuration object, set the Target Function Library, and provide the configuration object in a call to the codegen function, as follows:

```
cfg = coder.config('lib','ecoder',true);
cgv.TargetFunctionLibrary = 'Addition & Subtraction Examples';
codegen -config cfg addsub_tow_int16 -args {t,t};
```



4 Build your Simulink model or MATLAB Coder application.

See the demo `rtwdemo_tfl_script`, which illustrates how to use TFLs to replace operators and functions in generated code. With each example model included in this demo, a separate TFL is provided to illustrate the creation of operator and function replacements and how to register the replacements with Simulink software.

Target Function Libraries Quick-Start Example

This section steps you through a simple example of the complete TFL workflow. (The materials for this example can easily be created based on the file and model displays in this section.)

1 Create and save a TFL table definition file that instantiates and populates a TFL table entry, such as the file `tfl_table_sinfcn.m` shown below. This file creates function table entries for the `sin` function. For detailed information on creating table definition files for math functions and operators, see “Creating Function Replacement Tables” on page 31-16.

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.

hTable = RTW.TflTable;
```

```
% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                                              'double', '<sin_dbl.h>','','');

% Create entry for single data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'single', 'sin_sgl', ...
                                              'double', '<sin_sgl.h>','','');
```

Note See "Example: Mapping Math Functions to Target-Specific Implementations" on page 31-27 for another example of `sin` function replacement, in which function arguments are created individually.

---

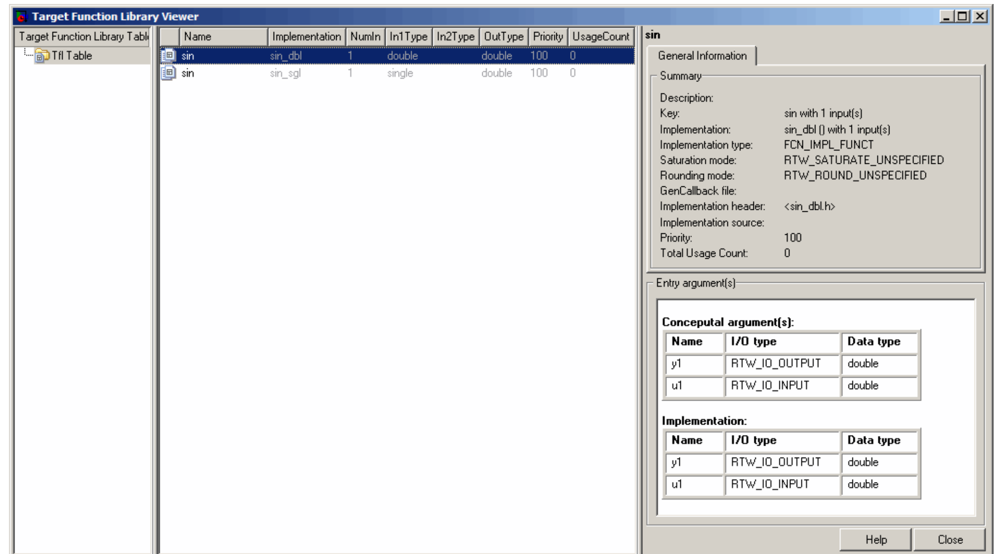
2 As a first check of the validity of your table entries, invoke the TFL table definition file as follows:

```
>> tbl = tfl_table_sinfcn

tbl =

RTW.TflTable
                Version: '1.0'
             AllEntries: [2x1 RTW.TflCFunctionEntry]
        ReservedSymbols: []
    StringResolutionMap: []
>>
```

Any errors found during the invocation are displayed.

3 As a further check of your table entries, invoke the TFL Viewer using the following MATLAB command:

```
>> RTW.viewTfl(tfl_table_sinfcn)
```



Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. (The TFL Viewer can also help you debug issues with the order of entries in a table, the order of tables in a TFL, and function signature mismatches. For more information, see "Examining and Validating Function Replacement Tables" on page 31-139.)

4 Create and save a TFL registration file that includes the `tfl_table_sinfcn` table, such as the `sl_customization.m` file shown below. The file specifies that the TFL to be registered is named `'Sine Function Example'` and consists of `tfl_table_sinfcn`, with the default ANSI[10] math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

10. ANSI® is a registered trademark of the American National Standards Institute, Inc.

```
% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

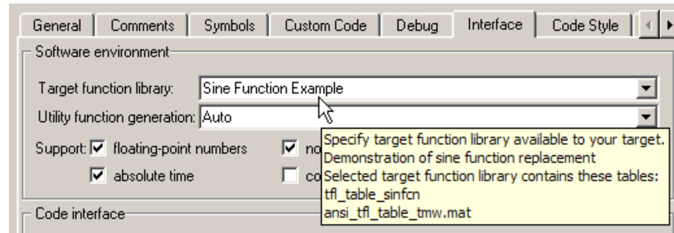
If you place this `sl_customization.m` file in the MATLAB search path or in the current working folder, the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see “Registering Target Function Libraries” on page 31-148.

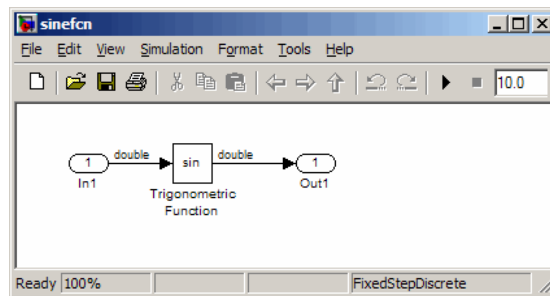
5. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip contains information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Sine Function Example`.

6 Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, `Sine Function Example`, is selected for this model.

7 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

8 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Code Generation > Navigate to Code**. This selection highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `sin_dbl` in the generated code.

```
21
22   /* Real-time model */
23   RT_MODEL_sinefcn sinefcn_M_;
24   RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;
25
26   /* Model step function */
27   void sinefcn_step(void)
28   {
29     /* Outport: '<Root>/Out1' incorporates:
30      *  Inport: '<Root>/In1'
31      *  Trigonometry: '<Root>/Trigonometric Function'
32      */
33     sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
34   }
35
36   /* Model initialize function */
37   void sinefcn_initialize(void)
38   {
39     /* Registration code */
```

9 If functions were not replaced as you intended, you can use the techniques described in “Examining and Validating Function Replacement Tables” on page 31-139 to help you determine why the code generation process was unable to match a function signature with the TFL table entry you created for it.

For example, you can view the TFL cache hits and misses logged during the most recent build. For the code generation step in this example, there was one cache hit and zero cache misses, as shown in the following `HitCache` and `MissCache` entries:

```
>> a=get_param( sinefcn , TargetFcnLibHandle )

a =

RTW.TflControl
        Version: '1.0'
       HitCache: [1x1 RTW.TflCFunctionEntry]
      MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
      TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =
```

```
RTW.TflCFunctionEntry
                          Key: 'sin'
                     Priority: 100
               ConceptualArgs: [2x1 RTW.TflArgNumeric]
               Implementation: [1x1 RTW.CImplementation]
.
.
.
>>
```

Creating Function Replacement Tables

In this section...

"Overview of Function Replacement Table Creation" on page 31-16

"Creating Table Entries" on page 31-20

"Example: Mapping Math Functions to Target-Specific Implementations" on page 31-27

"Example: Mapping the memcpy Function to a Target-Specific Implementation" on page 31-34

"Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations" on page 31-38

"Example: Mapping Scalar Operators to Target-Specific Implementations" on page 31-43

"Mapping Nonscalar Operators to Target-Specific Implementations" on page 31-49

"Mapping Fixed-Point Operators to Target-Specific Implementations" on page 31-78

"Remapping Operator Outputs to Implementation Function Input Positions" on page 31-113

"Refining TFL Matching and Replacement Using Custom TFL Table Entries" on page 31-115

"Replacing Math Functions Based on Computation Method" on page 31-132

"Specifying Build Information for Function Replacements" on page 31-134

"Adding Target Function Library Reserved Identifiers" on page 31-137

Overview of Function Replacement Table Creation

To create a TFL table containing replacement information for supported functions and operators, you perform the following steps:

1 Create a table definition file containing a function definition in the following general form:

```
function hTable = tfl_table_name()
```

```
%TFL_TABLE_NAME - Describe entries for a Target Function Library table.
.
.
.
```

For example, the following sample function definition is from the "Target Function Libraries Quick-Start Example" on page 31-9:

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.
.
.
.
```

2 Within the function body, instantiate a TFL table with a command such as the following:

```
hTable = RTW.TflTable;
```

3 Use the TFL table creation functions (listed in the table below) to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to

   a Instantiate a table entry.

   b Add conceptual arguments, implementation arguments, and other attributes to the entry.

   c Add the entry to the table.

   "Creating Table Entries" on page 31-20 describes this procedure in detail, including two methods for creating function entries. The following sample function entry is from the "Target Function Libraries Quick-Start Example" on page 31-9:

```
% Create entry for double data type sine function replacement
hTable.registerCFunctionEntry(100, 1, 'sin', 'double', 'sin_dbl', ...
                                          'double', '<sin_dbl.h>','','');
```

4 Save the table definition file using the name of the table definition function, for example, `tfl_table_sinfcn.m`.

After you have created a table definition file, you can do the following:

- Examine and validate the table, as described in "Examining and Validating Function Replacement Tables" on page 31-139.
- Register a TFL containing the table with the Simulink software, as described in "Registering Target Function Libraries" on page 31-148.

After you register a TFL with the Simulink software, it appears in the Simulink GUI and can be selected for use in building models.

The following table provides a functional grouping of the TFL table creation functions.

Function	Description
Table entry creation	
`addEntry`	Add table entry to collection of table entries registered in TFL table
`copyConceptualArgsToImplementation`	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
`createAndAddConceptualArg`	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
`createAndAddImplementationArg`	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
`createAndSetCImplementationReturn`	Create implementation return argument from specified properties and add to implementation for TFL table entry
`enableCPP`	Enable C++ support for function entry in TFL table
`setNameSpace`	Set name space for C++ function entry in TFL table
`setTflCFunctionEntryParameters`	Set specified parameters for function entry in TFL table
`setTflCOperationEntryParameters`	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	

Function	Description
`addConceptualArg`	Add conceptual argument to array of conceptual arguments for TFL table entry
`getTflArgFromString`	Create TFL argument based on specified name and built-in data type
Alternative method for function entry creation	
`registerCFunctionEntry`	Create TFL function entry based on specified parameters and register in TFL table
`registerCPPFunctionEntry`	Create TFL C++ function entry based on specified parameters and register in TFL table
`registerCPromotableMacroEntry`	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)
Build information	
`addAdditionalHeaderFile`	Add additional header file to array of additional header files for TFL table entry
`addAdditionalIncludePath`	Add additional include path to array of additional include paths for TFL table entry
`addAdditionalLinkObj`	Add additional link object to array of additional link objects for TFL table entry
`addAdditionalLinkObjPath`	Add additional link object path to array of additional link object paths for TFL table entry
`addAdditionalSourceFile`	Add additional source file to array of additional source files for TFL table entry
`addAdditionalSourcePath`	Add additional source path to array of additional source paths for TFL table entry
Reserved identifiers	
`setReservedIdentifiers`	Register specified reserved identifiers to be associated with TFL table

Creating Table Entries

- "Overview of Table Entry Creation" on page 31-20
- "General Method for Creating Function and Operator Entries" on page 31-22
- "Alternative Method for Creating Function Entries" on page 31-26

Overview of Table Entry Creation

You define TFL table entries by issuing TFL table creation function calls inside a table definition file. The function calls must follow a function declaration and a TFL table instantiation, such as the following:

```
function hTable = tfl_table_sinfcn()
%TFL_TABLE_SINFCN - Describe function entries for a Target Function Library table.

hTable = RTW.TflTable;
```

Within the function body, you use the TFL table creation functions to add table entries representing your replacements for supported functions and operators. For each individual function or operator entry, you issue one or more function calls to

1. Instantiate a table entry.
2. Add conceptual arguments, implementation arguments, and other attributes to the entry.
3. Add the entry to the table.

The general method for creating function and operator entries, described in "General Method for Creating Function and Operator Entries" on page 31-22, uses the functions shown in the following table.

Function	Description
Table entry creation	
`addEntry`	Add table entry to collection of table entries registered in TFL table

Function	Description
copyConceptualArgsToImplementation	Copy conceptual argument specifications to matching implementation arguments for TFL table entry
createAndAddConceptualArg	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
createAndAddImplementationArg	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
createAndSetCImplementationReturn	Create implementation return argument from specified properties and add to implementation for TFL table entry
enableCPP	Enable C++ support for function entry in TFL table
setNameSpace	Set name space for C++ function entry in TFL table
setTflCFunctionEntryParameters	Set specified parameters for function entry in TFL table
setTflCOperationEntryParameters	Set specified parameters for operator entry in TFL table
Alternative method for conceptual argument creation	
addConceptualArg	Add conceptual argument to array of conceptual arguments for TFL table entry
getTflArgFromString	Create TFL argument based on specified name and built-in data type

A simpler alternative creation method is available for function entries, with the constraints that input types must be uniform and implementation arguments must use default Simulink naming. The alternative method uses the following functions and is described in "Alternative Method for Creating Function Entries" on page 31-26.

Function	Description
Alternative method for function entry creation	
registerCFunctionEntry	Create TFL function entry based on specified parameters and register in TFL table

Function	Description
`registerCPPFunctionEntry`	Create TFL C++ function entry based on specified parameters and register in TFL table
`registerCPromotableMacroEntry`	Create TFL promotable macro entry based on specified parameters and register in TFL table (for `abs` function replacement only)

General Method for Creating Function and Operator Entries

The general workflow for creating TFL table entries applies equally to function and operator replacements, and involves the following steps.

Note

- You can remap operator outputs to implementation function inputs for operator replacement entries (see "Remapping Operator Outputs to Implementation Function Input Positions" on page 31-113). However, for function replacement entries, implementation argument order must match the conceptual argument order. Remapping the argument order in a function implementation is not supported.
- For function entries, if your implementations additionally meet the requirements that all input arguments are of the same type and your implementation arguments use default Simulink naming (return argument `y1` and input arguments `u`*n*), you can use a simpler alternative method for creating the entries, as described in "Alternative Method for Creating Function Entries" on page 31-26.

---

1. Within the function body of your table definition file, instantiate a TFL table entry for a function or operator, using one of the following lines of code:

`fcn_entry = RTW.TflCFunctionEntry;`	Supports function replacement
`fcn_entry = MyCustomFunctionEntry;` (where *`MyCustomFunctionEntry`* is a class derived from `RTW.TflCFunctionEntry`)	Supports function replacement using custom TFL table entries, described in "Refining TFL Matching and Replacement Using Custom TFL Table Entries" on page 31-115
`op_entry = RTW.TflCOperationEntry;`	Supports operator replacement
`op_entry = RTW.TflCOperationEntry-Generator;`	Provides relative scaling factor (RSF) fixed-point parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations" on page 31-78, that are not available in `RTW.TflCOperationEntry`
`op_entry = RTW.TflCOperationEntry-Generator_NetSlope;`	Provides net slope parameters, described in "Mapping Fixed-Point Operators to Target-Specific Implementations" on page 31-78, that are not available in `RTW.TflCOperationEntry`
`op_entry = RTW.TflBlasEntry-Generator;`	Supports replacement of nonscalar operators with MathWorks BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations" on page 31-49

`op_entry = RTW.TflCBlasEntry-Generator;`	Supports replacement of nonscalar operators with ANSI/ISO C BLAS functions, described in "Mapping Nonscalar Operators to Target-Specific Implementations" on page 31-49
`op_entry =` *`MyCustomOperationEntry`*`;` (where *`MyCustomOperationEntry`* is a class derived from `RTW.TflCOperationEntry`)	Supports operator replacement using custom TFL table entries, described in "Refining TFL Matching and Replacement Using Custom TFL Table Entries" on page 31-115

2 Set the table entry parameters, which are passed in parameter/value pairs to one of the following functions:

- `setTflCFunctionEntryParameters`
- `setTflCOperationEntryParameters`

For example:

```
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                'sin', ...
                               'Priority',           30, ...
                               'ImplementationName',  'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');
```

For detailed descriptions of the settable function and operator attributes, see the `setTflCFunctionEntryParameters` and `setTflCOperationEntryParameters` reference pages.

3 Create and add conceptual arguments to the function or operator entry. Output arguments must precede input arguments, and the function signature (including argument naming, order, and attributes) must fulfill the signature match sought by function or operator callers. Conceptual argument names follow the default Simulink naming convention:

- For return argument, `y1`
- For input argument names, `u1`, `u2`, ..., `u`*`n`*

You can create and add conceptual arguments in either of two ways:

- Call the `createAndAddConceptualArg` function to create the argument and add it to the table entry. For example:

```
createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1',...
                          'IOType',       'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');
```

- Call the `getTflArgFromString` function to create an argument based on a built-in data type, and then call the `addConceptualArg` function to add the argument to the table entry.

Note If you use `getTflArgFromString`, the `IOType` property of the created argument defaults to `'RTW_IO_INPUT'`, indicating an input argument. For an output argument, you must change the `IOType` value to `'RTW_IO_OUTPUT'` by directly assigning the argument property, as shown in the following example.

```
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

4 Create and add implementation arguments, representing the signature of your implementation function, to the function or operator entry. The implementation argument order must match the conceptual argument order. You can create and add implementation arguments in either of two ways:

- Call the `copyConceptualArgsToImplementation` function to populate all of the implementation arguments as copies of the previously created conceptual arguments. For example:

```
copyConceptualArgsToImplementation(fcn_entry);
```

- Call the `createAndSetCImplementationReturn` function to create the implementation return argument and add it to the table entry, and then call the `createAndAddImplementationArg` function to individually create and add each of your implementation arguments. This method

allows you to vary argument attributes, including argument naming, as long as conceptual argument order is maintained. For example:

```
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                  'Name',       'y1', ...
                                  'IOType',     'RTW_IO_OUTPUT', ...
                                  'IsSigned',   true, ...
                                  'WordLength', 32, ...
                                  'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',       'u1', ...
                              'IOType',     'RTW_IO_INPUT',...
                              'IsSigned',   true,...
                              'WordLength', 32, ...
                              'FractionLength', 0 );

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',       'u2', ...
                              'IOType',     'RTW_IO_INPUT',...
                              'IsSigned',   true,...
                              'WordLength', 32, ...
                              'FractionLength', 0 );
```

5. Add the function or operator entry to the TFL table using the `addEntry` function. For example:

```
addEntry(hTable, fcn_entry);
```

For complete examples of function entries and operator entries created using the general method, see “Example: Mapping Math Functions to Target-Specific Implementations” on page 31-27 and “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 31-43. For syntax examples, see the examples in the TFL table creation function reference pages.

Alternative Method for Creating Function Entries

You can use a simpler alternative method for creating TFL function entries if your function implementation meets the following criteria:

- The implementation argument order matches the conceptual argument order.
- All input arguments are of the same type.
- The return argument name and all input argument names follow the default Simulink naming convention:
  - For the return argument, `y1`
  - For input argument names, `u1`, `u2`, ..., `u`*n*

The alternative method for creating function entries involves a single step. Call one of the following functions to create and add conceptual and implementation arguments and register the function entry:

- `registerCFunctionEntry`
- `registerCPPFunctionEntry`
- `registerCPromotableMacroEntry` (use only for the `abs` function)

For example:

```
hTable = RTW.TflTable;

registerCFunctionEntry(hTable, 100, 1, 'sqrt', 'double', ...
                       'sqrt', 'double', '<math.h>', '', '');
```

For detailed descriptions of the function arguments, see the `registerCFunctionEntry`, `registerCPPFunctionEntry`, and `registerCPromotableMacroEntry` reference pages.

Example: Mapping Math Functions to Target-Specific Implementations

The Embedded Coder software supports the following math functions for replacement with custom library functions using target function library (TFL) tables.

Math Function	Simulink Support	Stateflow Support	MATLAB functions and MATLAB Coder Support
abs	• Floating-point • Integer	• Floating-point • Integer	Floating-point
acos	Floating-point	Floating-point	Floating-point
acosh	Floating-point	Not available (NA)	Not replaceable (NR)
asin	Floating-point	Floating-point	Floating-point
asinh	Floating-point	NA	NR
atan	Floating-point	Floating-point	Floating-point
atan2	Floating-point	Floating-point	Floating-point
atanh	Floating-point	NA	NR
ceil	Floating-point	Floating-point	Floating-point
cos	Floating-point	Floating-point	Floating-point
cosh	Floating-point	Floating-point	Floating-point
exactrSqrt	• Floating-point • Integer	NA	NA
exp	Floating-point	Floating-point	Floating-point
fix	Floating-point	NA	NR
floor	Floating-point	Floating-point	Floating-point
hypot	Floating-point	NA	NR
ldexp	Floating-point	Floating-point	Floating-point
ln	Floating-point	NA	NA
log	Floating-point	Floating-point	Floating-point
log10	Floating-point	Floating-point	Floating-point
max	• Floating-point • Integer	• Floating-point • Integer	• Floating-point • Integer

Math Function	Simulink Support	Stateflow Support	MATLAB functions and MATLAB Coder Support
`min`	• Floating-point • Integer	• Floating-point • Integer	• Floating-point • Integer
`mod`/`fmod`	• Floating-point (`mod`) • Integer (`mod`)	Floating-point (`fmod`)	NR
`pow`	Floating-point	Floating-point	Floating-point
`rem`	Floating-point	NA	Floating-point
`round`	Floating-point	NA	NR
`rSqrt`	• Floating-point • Integer	NA	NA
`saturate`	• Floating-point • Integer	NA	NA
`sign`	• Floating-point • Integer	NA	NR
`sin`	Floating-point	Floating-point	Floating-point
`sinh`	Floating-point	Floating-point	Floating-point
`sqrt`	Floating-point	Floating-point	Floating-point
`tan`	Floating-point	Floating-point	Floating-point
`tanh`	Floating-point	Floating-point	Floating-point

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for the `sin` function.

Note See "Target Function Libraries Quick-Start Example" on page 31-9 for another example of `sin` function replacement, in which function arguments are created using the simpler method described in "Alternative Method for Creating Function Entries" on page 31-26.

1 Create and save the following TFL table definition file, `tfl_table_sinfcn2.m`. This file defines a TFL table containing a function replacement entry for the `sin` function.

   The function body sets selected sine function entry parameters, creates the `y1` and `u1` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

```
function hTable = tfl_table_sinfcn2()
%TFL_TABLE_SINFCN2 - Describe function entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for sine function replacement
fcn_entry = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
                               'Key',                  'sin', ...
                               'Priority',             30, ...
                               'ImplementationName',  'mySin', ...
                               'ImplementationHeaderFile', 'basicMath.h',...
                               'ImplementationSourceFile', 'basicMath.c');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1',...
                          'IOType',       'RTW_IO_OUTPUT',...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'u1', ...
                          'IOType',       'RTW_IO_INPUT',...
                          'DataTypeMode', 'double');

copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);
```

2 Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_sinfcn2`) and by viewing it in the TFL Viewer

(>> `RTW.viewTfl(tfl_table_sinfcn2)`). For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3 Create and save the following TFL registration file, which references the `tfl_table_sinfcn2` table.

The file specifies that the TFL to be registered is named `'Sine Function Example 2'` and consists of `tfl_table_sinfcn2`, with the default ANSI[11] math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_sinfcn2
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example 2';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn2'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

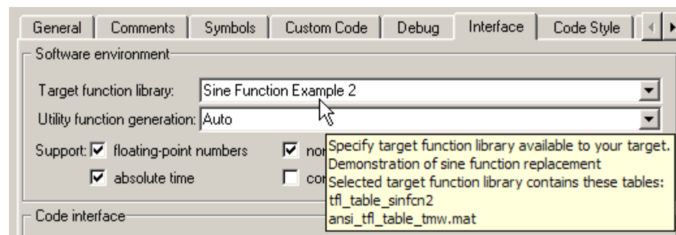
11. ANSI® is a registered trademark of the American National Standards Institute, Inc.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

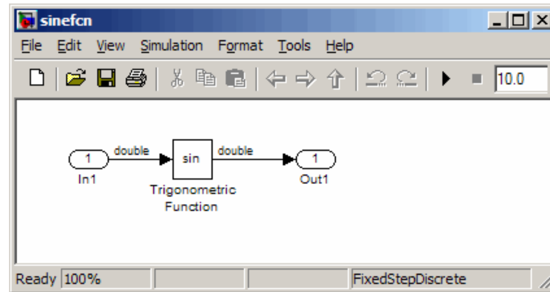
4. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Sine Function Example 2`.

5. Create an ERT-based model with a Trigonometric Function block set to the sine function, such as the following:



Make sure that the TFL you registered, `Sine Function Example 2`, is selected for this model.

6. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7. Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Trigonometric Function block and select **Code Generation > Navigate to Code**. This selection highlights the `sin` function code within the model step function in `sinefcn.c`. In this case, `sin` has been replaced with `mySin` in the generated code.

```
21
22   /* Real-time model */
23   RT_MODEL_sinefcn sinefcn_M_;
24   RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;
25
26   /* Model step function */
27   void sinefcn_step(void)
28   {
29     /* Outport: '<Root>/Out1' incorporates:
30      *  Inport: '<Root>/In1'
31      *  Trigonometry: '<Root>/Trigonometric Function'
32      */
33     sinefcn_Y.Out1 = mySin(sinefcn_U.In1);
34   }
35
36   /* Model initialize function */
37   void sinefcn_initialize(void)
38   {
39     /* Registration code */
```

Example: Mapping the memcpy Function to a Target-Specific Implementation

The Embedded Coder software supports the following memory utility functions for replacement with custom library functions using target function library (TFL) tables.

```
memcmp
memcpy
memset
memset2zero
```

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for the `memcpy` function.

1. Create and save the following TFL table definition file, `tfl_table_memcpy.m`. This file defines a TFL table containing a function replacement entry for the `memcpy` function.

   The function body sets selected `memcpy` function entry parameters, creates the `y1`, `u1`, `u2`, and `u3` conceptual arguments individually, adds each argument to the conceptual arguments array for the function, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

   ```
   function hTable = tfl_table_memcpy()
   %TFL_TABLE_MEMCPY - Describe memcpy function entry for a TFL table.

   hTable = RTW.TflTable;

   % Create function replacement entry for void* memcpy(void*, void*, size_t)
   fcn_entry = RTW.TflCFunctionEntry;
   setTflCFunctionEntryParameters(fcn_entry, ...
                                  'Key',                  'memcpy', ...
                                  'Priority',             90, ...
                                  'ImplementationName',   'memcpy_int', ...
                                  'ImplementationHeaderFile', 'memcpy_int.h',...
                                  'SideEffects',          true);
   % Set SideEffects to 'true' for function returning void to prevent it being
   % optimized away
   ```

```
arg = getTflArgFromString(hTable, 'y1', 'void*');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(fcn_entry, arg);

arg = getTflArgFromString(hTable, 'u1', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTflArgFromString(hTable, 'u2', 'void*');
addConceptualArg(fcn_entry, arg);

arg = getTflArgFromString(hTable, 'u3', 'size_t');
addConceptualArg(fcn_entry, arg);

copyConceptualArgsToImplementation(fcn_entry);
addEntry(hTable, fcn_entry);
```

2 Optionally, perform a quick check of the validity of the `memcpy` entry by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_memcpy`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_memcpy)`). For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3 Create and save the following TFL registration file, which references the `tfl_table_memcpy` table.

The file specifies that the TFL to be registered is named `'Memcpy Function Example'` and consists of `tfl_table_memcpy`, with the default ANSI[12] math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

12. ANSI® is a registered trademark of the American National Standards Institute, Inc.

```
% Local function to define a TFL containing tfl_table_memcpy
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Memcpy Function Example';
  thisTfl.Description = 'Demonstration of memcpy function replacement';
  thisTfl.TableList = {'tfl_table_memcpy'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

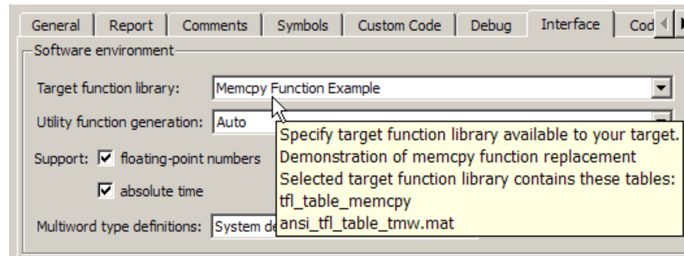
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

4. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

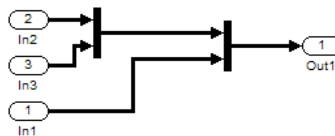
Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



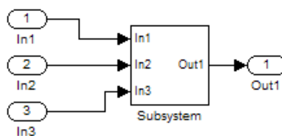
Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Memcpy Function Example`.

5 Create an ERT-based model that uses `memcpy` for vector assignments. For example,

a Use In, Out, and Mux blocks to create the following model. (Alternatively, you can open `rtwdemo_tflmath/Subsystem1` and copy the subsystem contents to a new model.)



b Select the diagram and use **Edit > Subsystem** to make it a subsystem.



c Select an ERT-based system target file on the **Code Generation** pane of the Configuration Parameters dialog box, and select the TFL you registered, `Memcpy Function Example`, on the **Interface** pane. You

should also select a fixed-step solver on the **Solver** pane. Leave the `memcpy` options on the **Optimization > Signals and Parameters** pane at their default settings, that is, **Use memcpy for vector assignment** is selected, and **Memcpy threshold (bytes)** at `64`. Apply the changes.

d Open Model Explorer and configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For each, set **Port dimensions** to `[1,100]`, and set **Data type** to `int32`. Apply the changes. Save the model. In this example, the model is saved to the name `memcpyfcn.mdl`.

6 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the **Create code generation report**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model. When code generation completes, the HTML code generation report is displayed.

7 In the HTML code generation report, click on the *`model`*`.c` section (for example, `memcpyfcn.c`) and inspect the model step function to confirm that `memcpy` has been replaced with `memcpy_int` in the generated code.

```
31  /* Model step function */
32  void memcpyfcn_step(void)
33  {
34    /* Outport: '<Root>/Out1' incorporates:
35     *  Inport: '<Root>/In1'
36     *  Inport: '<Root>/In2'
37     *  Inport: '<Root>/In3'
38     */
39    memcpy_int((void *)(&memcpyfcn_Y.Out1[0]), (void *)(&memcpyfcn_U.In2[0]), 100U
40               * sizeof(int32_T));
41    memcpy_int((void *)(&memcpyfcn_Y.Out1[100]), (void *)(&memcpyfcn_U.In3[0]),
42               100U * sizeof(int32_T));
43    memcpy_int((void *)(&memcpyfcn_Y.Out1[200]), (void *)(&memcpyfcn_U.In1[0]),
44               100U * sizeof(int32_T));
45  }
46
47  /* Model initialize function */
48  void memcpyfcn_initialize(void)
49  {
```

Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations

The Embedded Coder software supports the following nonfinite support utility functions for replacement with custom library functions using target function library (TFL) tables.

```
GetInf
GetMinusInf
```

```
GetNaN
```

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create TFL table entries for the nonfinite functions.

1. Create and save the following TFL table definition file, `tfl_table_nonfinite.m`. This file defines a TFL table containing function replacement entries for the nonfinite functions.

   For each nonfinite function, the function body uses the local function `locAddFcnEnt` to create entries for `single` and `double` replacement. For each entry, the local function sets selected function entry parameters, creates the `y1` and `u1` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally the function entry is added to the table.

   ```
   function hTable = tfl_table_nonfinite()
   %TFL_TABLE_NONFINITE - Describe function entries for a TFL table.

   hTable = RTW.TflTable;

   %% Create entries for nonfinite support utility functions
   %locAddFcnEnt(hTable, key,          implName,        out,      in1,    hdr )
   locAddFcnEnt(hTable, 'getNaN',      'getNaN',       'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getNaN',      'getNaNF',      'single', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getInf',      'getInf',       'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getInf',      'getInfF',      'single', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInf',  'double', 'void', 'nonfin.h');
   locAddFcnEnt(hTable, 'getMinusInf', 'getMinusInfF', 'single', 'void', 'nonfin.h');

   %% Local Function
   function locAddFcnEnt(hTable, key, implName, out, in1, hdr)
     if isempty(hTable)
       return;
     end

     fcn_entry = RTW.TflCFunctionEntry;
     setTflCFunctionEntryParameters(fcn_entry, ...
                                    'Key', key, ...
   ```

```
                                              'Priority', 90, ...
                                              'ImplementationName', implName, ...
                                              'ImplementationHeaderFile', hdr);

    arg = getTflArgFromString(hTable, 'y1', out);
    arg.IOType = 'RTW_IO_OUTPUT';
    addConceptualArg(fcn_entry, arg);

    arg = getTflArgFromString(hTable, 'u1', in1);
    addConceptualArg(fcn_entry, arg);

    copyConceptualArgsToImplementation(fcn_entry);

    addEntry(hTable, fcn_entry);

    %EOF
```

2. Optionally, perform a quick check of the validity of the nonfinite function entries by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_nonfinite`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_nonfinite)`). For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3. Create and save the following TFL registration file, which references the `tfl_table_nonfinite` table.

   The file specifies that the TFL to be registered is named `'Nonfinite Functions Example'` and consists of `tfl_table_nonfinite`, with the default ANSI[13] math library as the base TFL table.

   ```
   function sl_customization(cm)
   % sl_customization function to register a target function library (TFL)

     % Register the TFL defined in local function locTflRegFcn
     cm.registerTargetInfo(@locTflRegFcn);

   end % End of SL_CUSTOMIZATION
   ```

13. ANSI® is a registered trademark of the American National Standards Institute, Inc.

```
% Local function to define a TFL containing tfl_table_nonfinite
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Nonfinite Functions Example';
  thisTfl.Description = 'Demonstration of nonfinite functions replacement';
  thisTfl.TableList = {'tfl_table_nonfinite'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

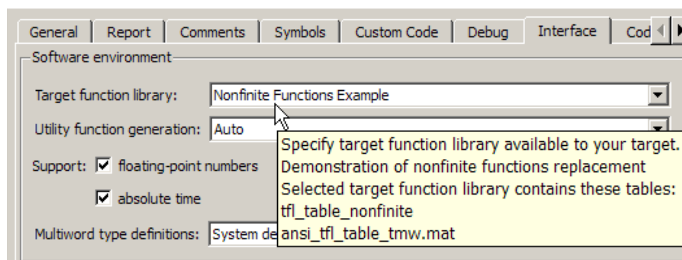
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

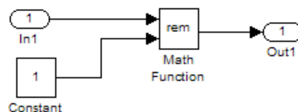
4. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Nonfinite Functions Example`.

5 Create an ERT-based model with a Math Function block set to the `rem` function, such as the following:



Open Model Explorer. Select the **Support: non-finite numbers** parameter on the **Code Generation > Interface** pane of the Configuration Parameters dialog box and configure the **Signal Attributes** for the `In1` and `Constant` source blocks. For each source block, set **Data type** to `double`. Apply the changes. Save the model. In this example, the model is saved to the name `nonfinitefcns.mdl`.

Make sure that the TFL you registered, `Nonfinite Functions Example`, is selected for this model.

6 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the option **Create code generation report**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7 In the HTML code generation report, click on the `rtnonfinite.c` link and inspect the `rt_InitInfAndNaN` function to confirm that your replacements for nonfinite support functions are present in the generated code.

```
13  /*
14   * Abstract:
15   *      Real-Time Workshop function to intialize non-finites,
16   *      (Inf, NaN and -Inf).
17   */
18  #include "rt_nonfinite.h"
19  #include "nonfin.h"
20  #include "nonfin.h"
21  #define NumBitsPerChar                 8
22
23  real_T rtInf;
24  real_T rtMinusInf;
25  real_T rtNaN;
26  real32_T rtInfF;
27  real32_T rtMinusInfF;
28  real32_T rtNaNF;
29
30  /* Function: rt_InitInfAndNaN ============================================
31   * Abstract:
32   *      Initialize the rtInf, rtMinusInf, and rtNaN needed by the
33   *      generated code. NaN is initialized as non-signaling. Assumes IEEE.
34   */
35  void rt_InitInfAndNaN(size_t realSize)
36  {
37    (void) (realSize);
38    rtNaN = getNaN();
39    rtNaNF = getNaNF();
40    rtInf = getInf();
41    rtInfF = getInfF();
42    rtMinusInf = getMinusInf();
43    rtMinusInfF = getMinusInfF();
44  }
```

Example: Mapping Scalar Operators to Target-Specific Implementations

The Embedded Coder software supports the following scalar operators for replacement with custom library functions using target function library (TFL) tables:

Operator	Key
Addition (+)	`RTW_OP_ADD`
Subtraction (-)	`RTW_OP_MINUS`
Multiplication (*)	`RTW_OP_MUL`
Division (/)	`RTW_OP_DIV`

Operator	Key
Data type conversion (cast)	`RTW_OP_CAST`
Shift left (<<) [integer and fixed-point data types]	`RTW_OP_SL`
Shift right (>>) [integer and fixed-point data types]	`RTW_OP_SRA` (arithmetic) [14] `RTW_OP_SRL` (logical)
Complex conjugation	`RTW_OP_CONJUGATE`

Unless otherwise stated, the listed operators are supported for the following input data types:

- `single`, `double`, and their complex equivalents
- `int8`, `int16`, `int32`, and their complex equivalents
- `uint8`, `uint16`, `uint32`, and their complex equivalents
- Fixed-point data types
- Mixed data types (different types for different inputs)

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for the + (addition) operator.

1. Create and save the following TFL table definition file, `tfl_table_add_uint8.m`. This file defines a TFL table containing an operator replacement entry for the + (addition) operator.

   The function body sets selected addition operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then copies the conceptual arguments to the implementation arguments. Finally, the operator entry is added to the table.

14. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

```
function hTable = tfl_table_add_uint8
%TFL_TABLE_ADD_UINT8 - Describe operator entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
% Saturation on, Rounding no preference
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c' );

arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);

arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

copyConceptualArgsToImplementation(op_entry);

addEntry(hTable, op_entry);
```

2 Optionally, perform a quick check of the validity of the operator entry by invoking the table definition file at the MATLAB command line (>> `tbl = tfl_table_add_uint8`) and by viewing it in the TFL Viewer (>> `RTW.viewTfl(tfl_table_add_uint8)`).

For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3 Create and save the following TFL registration file, which references the `tfl_table_add_uint8` table.

The file specifies that the TFL to be registered is named `'Addition Operator Example'` and consists of `tfl_table_add_uint8`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_add_uint8
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Addition Operator Example';
  thisTfl.Description = 'Demonstration of addition operator replacement';
  thisTfl.TableList = {'tfl_table_add_uint8'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

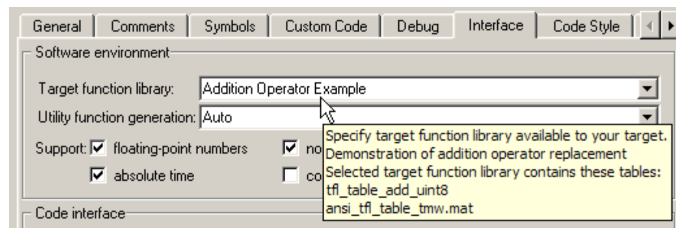
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see “Registering Target Function Libraries” on page 31-148.

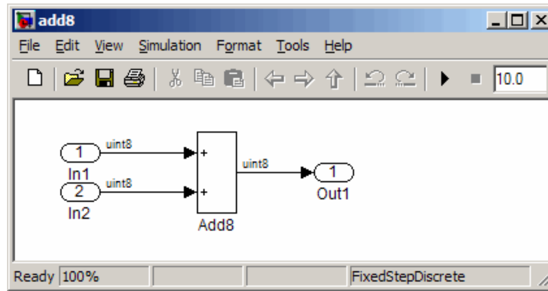
4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Addition Operator Example`.

5 Create an ERT-based model with an Add block, such as the following:



Make sure that the TFL you registered, `Addition Operator Example`, is selected for this model.

6 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Add block and select **Code Generation > Navigate to Code**. This selection highlights the Sum block code within the model step function in `add8.c`. In this case, code containing the + operator has been replaced with `u8_add_u8_u8` in the generated code.

```
21
22   /* Real-time model */
23   RT_MODEL_add8 add8_M_;
24   RT_MODEL_add8 *add8_M = &add8_M_;
25
26   /* Model step function */
27   void add8_step(void)
28   {
29     /* Outport: '<Root>/Out1' incorporates:
30      *  Inport: '<Root>/In1'
31      *  Inport: '<Root>/In2'
32      *  Sum: '<Root>/Add8'
33      */
34     add8_Y.Out1 = u8_add_u8_u8(add8_U.In1, add8_U.In2);
35   }
36
37   /* Model initialize function */
38   void add8_initialize(void)
39   {
```

Mapping Nonscalar Operators to Target-Specific Implementations

- "Example: Mapping Small Matrix Operations to Processor-Specific Intrinsic Functions" on page 31-50
- "Example: Mapping Matrix Multiplication to MathWorks BLAS Functions" on page 31-57
- "Example: Mapping Matrix Multiplication to ANSI/ISO C BLAS Functions" on page 31-67

The Embedded Coder software supports the following nonscalar operators for replacement with custom library functions using target function library (TFL) tables:

Operator	Key
Addition (+)	`RTW_OP_ADD`
Subtraction (-)	`RTW_OP_MINUS`
Multiplication (`*`, `.*`)	`RTW_OP_MUL`
Complex conjugation	`RTW_OP_CONJUGATE`
Transposition (`.'`)	`RTW_OP_TRANS`
Hermitian (complex conjugate) transposition (`'`)	`RTW_OP_HERMITIAN`
Multiplication with transposition	`RTW_OP_TRMUL`
Multiplication with Hermitian transposition	`RTW_OP_HMMUL`

These operators are supported for the following input data types:

- `single`, `double`, and their complex equivalents
- `int8`, `int16`, `int32`, and their complex equivalents
- `uint8`, `uint16`, `uint32`, and their complex equivalents
- Fixed-point data types

- Mixed data types (different types for different inputs)

Note Saturation and rounding modes are ignored for floating-point nonscalar addition and subtraction. In TFL table entries for nonscalar addition and subtraction, if the argument data types are all floating-point, the `setTflCOperationEntryParameters` function call should register `'RTW_SATURATE_UNSPECIFIED'` for the `SaturationMode` parameter and `'RTW_ROUND_UNSPECIFIED'` for the `RoundingMode` parameter.

Example: Mapping Small Matrix Operations to Processor-Specific Intrinsic Functions

You can efficiently implement small matrix operations by invoking processor-specific intrinsic functions. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry mapping small matrix sum operations to implementation functions that could invoke processor-specific intrinsic functions.

Note For examples of replacing other matrix operations and handling other data types, see the Matrix Operator Replacement section of the TFL demos page `rtwdemo_tfl_script`, including the demo model `rtwdemo_tflmatops` and its associated files.

---

1. Create and save the following TFL table definition file, `tfl_table_matrix_add_double.m`. This file defines a TFL table containing two matrix operator replacement entries for the `+` (addition) operator and the `double` data type.

   The function body sets selected addition operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures the implementation arguments. Finally, the operator entry is added to the table.

   To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.TflArgMatrix` and specify the base type and the

dimensions for which the argument is valid. In this example, the first table entry specifies [2 2] and the second table entry specifies [3 3].

```
function hTable = tfl_table_matrix_add_double
%TFL_TABLE_MATRIX_ADD_DOUBLE - Describe two matrix operator entries for a TFL table.

hTable = RTW.TflTable;

LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'tfl_demo');

% Create table entry for matrix_sum_2x2_double
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                        'RTW_OP_ADD', ...
    'Priority',                   30, ...
    'SaturationMode',             'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName',         'matrix_sum_2x2_double', ...
    'ImplementationHeaderFile',   'MatrixMath.h', ...
    'ImplementationSourceFile',   'MatrixMath.c', ...
    'ImplementationHeaderPath',   LibPath, ...
    'ImplementationSourcePath',   LibPath, ...
    'AdditionalIncludePaths',     {LibPath}, ...
    'GenCallback',                'RTW.copyFileToBuildDir', ...
    'SideEffects',                true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',       'y1', ...
                          'IOType',     'RTW_IO_OUTPUT', ...
                          'BaseType',   'double', ...
                          'DimRange',   [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',       'u1', ...
                          'BaseType',   'double', ...
                          'DimRange',   [2 2]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',       'u2', ...
                          'BaseType',   'double', ...
                          'DimRange',   [2 2]);
```

```
% Specify replacement function signature
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for matrix_sum_3x3_double
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                       'RTW_OP_ADD', ...
    'Priority',                  30, ...
    'SaturationMode',            'RTW_WRAP_ON_OVERFLOW', ...
    'ImplementationName',        'matrix_sum_3x3_double', ...
    'ImplementationHeaderFile', 'MatrixMath.h', ...
    'ImplementationSourceFile', 'MatrixMath.c', ...
    'ImplementationHeaderPath', LibPath, ...
    'ImplementationSourcePath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',               'RTW.copyFileToBuildDir', ...
    'SideEffects',               true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [3 3]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
```

```
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [3 3]);

% Specify replacement function signature
arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);
arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

2 Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (>> `tbl = tfl_table_matrix_add_double`) and by viewing it in the TFL Viewer (>> `RTW.viewTfl(tfl_table_matrix_add_double)`).

For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3 Create and save the following TFL registration file, which references the `tfl_table_matrix_add_double` table.

The file specifies that the TFL to be registered is named `'Matrix Addition Operator Example'` and consists of `tfl_table_matrix_add_double`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

```
% Local function to define a TFL containing tfl_table_matrix_add_double
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Matrix Addition Operator Example';
  thisTfl.Description = 'Demonstration of matrix addition operator replacement';
  thisTfl.TableList = {'tfl_table_matrix_add_double'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

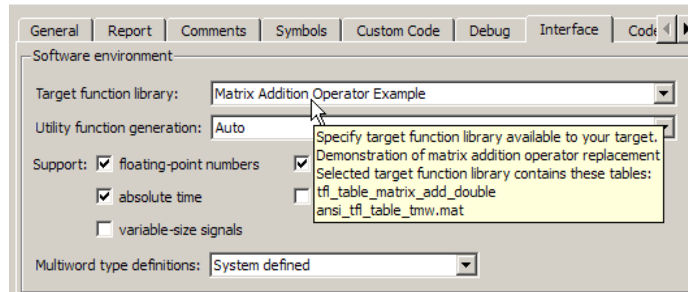
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see “Registering Target Function Libraries” on page 31-148.

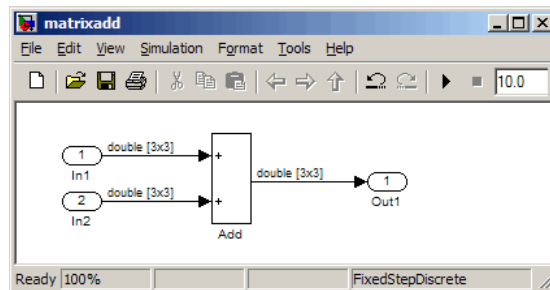
4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Matrix Addition Operator Example`.

5 Create an ERT-based model with an Add block, such as the following:



Configure the **Signal Attributes** for the `In1` and `In2` source blocks. For each source block, set **Port dimensions** to `[3 3]` and set the **Data type** to `double`. Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as `0.1`. Apply the changes. Save the model. In this example, the model is saved to the name `matrixadd.mdl`.

Make sure that the TFL you registered, `Matrix Addition Operator Example`, is selected for this model.

6 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the Add block and select **Code Generation > Navigate to Code**. This selection highlights the Sum block code within the model step function in `matrixadd.c`. In this case, code containing the + operator has been replaced with `matrix_sum_3x3_double` in the generated code.

```
26
27  /* Real-time model */
28  RT_MODEL_matrixadd matrixadd_M_;
29  RT_MODEL_matrixadd *matrixadd_M = &matrixadd_M_;
30
31  /* Model step function */
32  void matrixadd_step(void)
33  {
34    /* Outport: '<Root>/Out1' incorporates:
35     *  Inport: '<Root>/In1'
36     *  Inport: '<Root>/In2'
37     *  Sum: '<Root>/Add'
38     */
39    matrix_sum_3x3_double(&matrixadd_U.In1[0], &matrixadd_U.In2[0],
40                          &matrixadd_Y.Out1[0]);
41  }
42
43  /* Model initialize function */
44  void matrixadd_initialize(void)
45  {
46    /* Registration code */
47
48    /* initialize error status */
49    rtmSetErrorStatus(matrixadd_M, (NULL));
50
```

Note Optionally, you can reconfigure the In1 and In2 block **Port dimensions** to `[2 2]`, regenerate code, and observe that code containing the + operator is replaced with `matrix_sum_2x2_double`.

Example: Mapping Matrix Multiplication to MathWorks BLAS Functions

You can use TFL tables to map nonscalar multiplication operations to the Basic Linear Algebra Subroutine (BLAS) multiplication functions `xgemm` and `xgemv`. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry mapping floating-point matrix/matrix and matrix/vector multiplication operations to MathWorks BLAS library multiplication functions.

Note For examples of handling other data types, see the BLAS Support section of the TFL demos page `rtwdemo_tfl_script`, including the demo model `rtwdemo_tflblas` and its associated files.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$, where $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, TFLs support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, TFLs support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create and save the following TFL table definition file, `tfl_table_tmwblas_mmult_double.m`. This file defines a TFL table containing `dgemm` and `dgemv` replacement entries for the matrix multiplication operator and the `double` data type.

   For each entry, the function body sets selected matrix multiplication operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures special implementation arguments that are required for `dgemm` and `dgemv` replacements. Finally, each operator entry is added to the table.

   To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min`

`... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means any two-dimensional matrix of size 2x2 or larger. In this example, the conceptual output argument for the `dgemm32` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `dgemv32` entry for matrix/vector multiplication replacement specifies dimensions `[2 1; inf 1]`.

```
function hTable = tfl_table_tmwblas_mmult_double
%TFL_TABLE_TMWBLAS_MMULT_DOUBLE - Describe two mmult operator entries for TFL table.

hTable = RTW.TflTable;

% Define library path for Windows or UNIX
arch = computer('arch');
if ~ispc
    LibPath = fullfile('$(MATLAB_ROOT)', 'bin', arch);
else
    % Use Stateflow to get the compiler info
    compilerInfo = sf('Private','compilerman','get_compiler_info');
    compilerName = compilerInfo.compilerName;
    if strcmp(compilerName, 'msvc90') || ...
            strcmp(compilerName, 'msvc80') || ...
            strcmp(compilerName, 'msvc71') || ...
            strcmp(compilerName, 'msvc60'), ...
            compilerName = 'microsoft';
    end
    LibPath = fullfile('$(MATLAB_ROOT)', 'extern', 'lib', arch, compilerName);
end

% Create table entry for dgemm32
op_entry = RTW.TflBlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
```

```
    'Key',                        'RTW_OP_MUL', ...
    'Priority',                   100, ...
    'ImplementationName',         'dgemm32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',         {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath}, ...
    'SideEffects',                true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf inf]);

% Using RTW.TflBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
% void f(char* TRANSA, char* TRANSB, int* M, int* N, int* K,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* LDB,
%        type* BETA, type* y, int* LDC)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code. TRANSA and TRANSB both will be set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
```

```
arg = RTW.TflArgCharConstant('TRANSA');
% Possible values for PassByType property are
%  RTW_PASSBY_AUTO, RTW_PASSBY_POINTER,
%  RTW_PASSBY_VOID_POINTER, RTW_PASSBY_BASE_POINTER
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = RTW.TflArgCharConstant('TRANSB');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
```

```
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for dgemv32
op_entry = RTW.TflBlasEntryGenerator;
if ispc
    libExt = 'lib';
elseif ismac
    libExt = 'dylib';
else
    libExt = 'so';
end
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_MUL', ...
```

```
    'Priority',                 100, ...
    'ImplementationName',       'dgemv32', ...
    'ImplementationHeaderFile', 'blascompat32.h', ...
    'ImplementationHeaderPath', fullfile('$(MATLAB_ROOT)','extern','include'), ...
    'AdditionalLinkObjs',       {['libmwblascompat32.' libExt]}, ...
    'AdditionalLinkObjsPaths',  {LibPath},...
    'SideEffects',              true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf 1]);

% Using RTW.TflBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(char* TRANS, int* M, int* N,
%        type* ALPHA, type* u1, int* LDA, type* u2, int* INCX,
%        type* BETA, type* y, int* INCY)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, LDA, INCX, and INCY, and insert them into the
% generated code. TRANS will be set to 'N'.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);
```

```
arg = RTW.TflArgCharConstant('TRANS');
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX','integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
```

```
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
arg.PassByType = 'RTW_PASSBY_POINTER';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
arg.PassByType = 'RTW_PASSBY_POINTER';
arg.Type.ReadOnly = true;
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

2. Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (`>> tbl = tfl_table_tmwblas_mmult_double`) and by viewing it in the TFL Viewer (`>> RTW.viewTfl(tfl_table_tmwblas_mmult_double)`).

   For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3. Create and save the following TFL registration file, which references the `tfl_table_tmwblas_mmult_double` table.

   The file specifies that the TFL to be registered is named `'MathWorks BLAS Matrix Multiplication Operator Example'` and consists of `tfl_table_tmwblas_mmult_double`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

```
% Local function to define a TFL containing tfl_table_tmwblas_mmult_double
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'MathWorks BLAS Matrix Multiplication Operator Example';
  thisTfl.Description = 'Demonstration of MathWorks BLAS mmult operator replacement';
  thisTfl.TableList = {'tfl_table_tmwblas_mmult_double'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

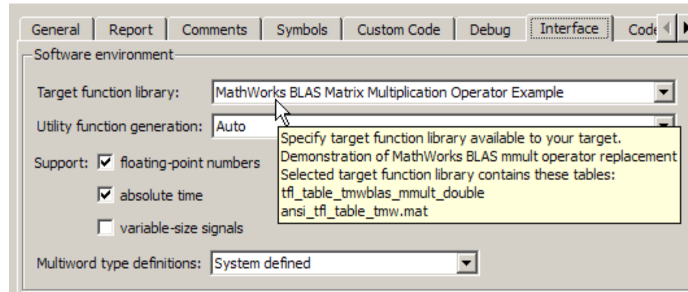
Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

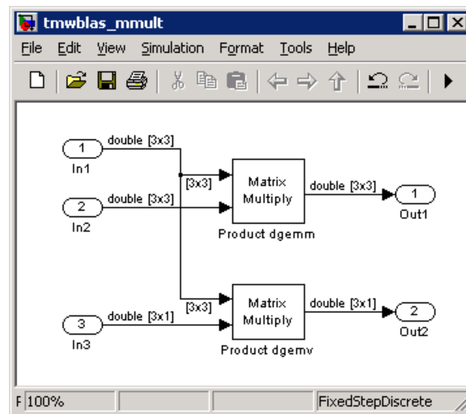
4. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `MathWorks BLAS Matrix Multiplication Operator Example`.

5 Create an ERT-based model with two Product blocks, such as the following:



a For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

b Configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For `In1` and `In2`, set **Port dimensions** to `[3 3]` and set the **Data type** to `double`. For `In3`, set **Port dimensions** to `[3 1]` and set the **Data type** to `double`.

c Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as `0.1`. Apply the changes.

d Save the model. In this example, the model is saved to the name `tmwblas_mmult.mdl`.

e Make sure that the TFL you registered, `MathWorks BLAS Matrix Multiplication Operator Example`, is selected for this model.

6 Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7 Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the top Product block and select **Code Generation > Navigate to Code**. This selection highlights the Product block code within the model step function in `tmwblas_mmult.c`. In this case, code containing the matrix multiplication operator has been replaced with a call to `dgemm32` in the generated code.

```
45    /* Outport: '<Root>/Out1' incorporates:
46     *  Inport: '<Root>/In1'
47     *  Inport: '<Root>/In2'
48     *  Product: '<Root>/Product dgemm'
49     */
50    TRANSA = 'N';
51    TRANSB = 'N';
52    M = 3;
53    N = 3;
54    K = 3;
55    ALPHA = 1.0;
56    LDA = 3;
57    LDB = 3;
58    BETA = 0.0;
59    LDC = 3;
60    dgemm32(&TRANSA, &TRANSB, &M, &N, &K, &ALPHA, &tmwblas_mmult_U.In1[0], &LDA,
61            &tmwblas_mmult_U.In2[0], &LDB, &BETA, &tmwblas_mmult_Y.Out1[0], &LDC);
62
63    /* Outport: '<Root>/Out2' incorporates:
64     *  Inport: '<Root>/In1'
65     *  Inport: '<Root>/In3'
66     *  Product: '<Root>/Product dgemv'
```

Example: Mapping Matrix Multiplication to ANSI/ISO C BLAS Functions

You can use TFL tables to map nonscalar multiplication operations to the ANSI/ISO C BLAS multiplication functions `xgemm` and `xgemv`. The following

example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry mapping floating-point matrix/matrix and matrix/vector multiplication operations to ANSI/ISO C BLAS library multiplication functions.

Note For examples of handling other data types, see the BLAS Support section of the TFL demos page `rtwdemo_tfl_script`, including the demo model `rtwdemo_tflblas` and its associated files.

BLAS libraries support matrix/matrix multiplication in the form of $C = a(op(A) * op(B)) + bC$, where $op(X)$ means X, transposition of X, or Hermitian transposition of X. However, TFLs support only the limited case of $C = op(A) * op(B)\ (a = 1.0, b = 0.0)$. Correspondingly, although BLAS libraries support matrix/vector multiplication in the form of $y = a(op(A) * x) + by$, TFLs support only the limited case of $y = op(A) * x\ (a = 1.0, b = 0.0)$.

1. Create and save the following TFL table definition file, `tfl_table_cblas_mmult_double.m`. This file defines a TFL table containing `dgemm` and `dgemv` replacement entries for the matrix multiplication operator and the `double` data type.

   For each entry, the function body sets selected matrix multiplication operator entry parameters, creates the `y1`, `u1`, and `u2` conceptual arguments individually, and then configures special implementation arguments that are required for `dgemm` and `dgemv` replacements. Finally, each operator entry is added to the table.

   To specify a matrix argument to `createAndAddConceptualArg`, use the TFL argument class `RTW.TflArgMatrix` and specify the base type and the dimensions for which the argument is valid. This type of table entry supports a range of dimensions specified in the format `[Dim1Min Dim2Min ... DimNMin; Dim1Max Dim2Max ... DimNMax]`. For example, `[2 2; inf inf]` means any two-dimensional matrix of size 2x2 or larger. In this example, the conceptual output argument for the `cblas_dgemm` entry for matrix/matrix multiplication replacement specifies dimensions `[2 2; inf inf]`, while the conceptual output argument for the `cblas_dgemv` entry

for matrix/vector multiplication replacement specifies dimensions [2 1; inf 1].

```
function hTable = tfl_table_cblas_mmult_double
%TFL_TABLE_CBLAS_MMULT_DOUBLE - Describe two mmult operator entries for TFL table.

hTable = RTW.TflTable;

LibPath = fullfile(matlabroot, 'toolbox', 'rtw', 'rtwdemos', 'tfl_demo');

% Create table entry for cblas_dgemm
op_entry = RTW.TflCBlasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                       'RTW_OP_MUL', ...
    'Priority',                  100, ...
    'ImplementationName',        'cblas_dgemm', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',   {LibPath}, ...
    'GenCallback',               'RTW.copyFileToBuildDir', ...
    'SideEffects',               true);

% Specify operands and result
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u1', ...
                          'BaseType',     'double', ...
                          'DimRange',     [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',         'u2', ...
                          'BaseType',     'double', ...
                          'DimRange',     [1 1; inf inf]);

% Using RTW.TflCBlasEntryGenerator for xgemm requires the following
% implementation signature:
%
```

```
% void f(enum ORDER, enum TRANSA, enum TRANSB, int M, int N, int K,
%        type ALPHA, type* u1, int LDA, type* u2, int LDB,
%        type BETA, type* y, int LDC)
%
% Since TFLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% appropriate enumeration type.)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, K, LDA, LDB, and LDC and insert them into the
% generated code.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'TRANSB', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'K', 'integer', 0);
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDB', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDC', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

% Create table entry for cblas_dgemv
op_entry = RTW.TflCBlasEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                        'RTW_OP_MUL', ...
    'Priority',                   100, ...
    'ImplementationName',         'cblas_dgemv', ...
    'ImplementationHeaderFile', 'cblas.h', ...
    'ImplementationHeaderPath', LibPath, ...
    'AdditionalIncludePaths',     {LibPath}, ...
    'GenCallback',                'RTW.copyFileToBuildDir', ...
    'SideEffects',                true);

% Specify operands and result
```

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',        'y1', ...
                          'IOType',      'RTW_IO_OUTPUT', ...
                          'BaseType',    'double', ...
                          'DimRange',    [2 1; inf 1]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix', ...
                          'Name',        'u1', ...
                          'BaseType',    'double', ...
                          'DimRange',    [2 2; inf inf]);
createAndAddConceptualArg(op_entry, 'RTW.TflArgMatrix',...
                          'Name',        'u2', ...
                          'BaseType',    'double', ...
                          'DimRange',    [1 1; inf 1]);

% Using RTW.TflCBlasEntryGenerator for xgemv requires the following
% implementation signature:
%
% void f(enum ORDER, enum TRANSA, int M, int N,
%        type ALPHA, type* u1, int LDA, type* u2, int INCX,
%        type BETA, type* y, int INCY)
%
% Since TFLs do not have the ability to specify enums, you must
% use integer.  (This will cause problems with C++ code generation,
% so for C++, use a wrapper function to cast each int to the
% appropriate enumeration type.)
%
% Upon a successful match, the TFL entry will compute the correct
% values for M, N, LDA, INCX, and INCY and insert them into the
% generated code.

% Specify replacement function signature

arg = getTflArgFromString(hTable, 'y2', 'void');
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'ORDER', 'integer', 102);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);
```

```
arg = getTflArgFromString(hTable, 'TRANSA', 'integer', 111);
%arg.Type.ReadOnly=true;
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'M','integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'N', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'ALPHA', 'double', 1);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u1', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'LDA', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2', ['double' '*']);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCX', 'integer', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'BETA', 'double', 0);
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'y1', ['double' '*']);
arg.IOType = 'RTW_IO_OUTPUT';
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'INCY', 'integer', 0);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

2 Optionally, perform a quick check of the validity of the operator entries by invoking the table definition file at the MATLAB command line (>> `tbl =`

tfl_table_cblas_mmult_double) and by viewing it in the TFL Viewer (>> RTW.viewTFL(tfl_table_cblas_mmult_double)).

For more information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

3 Create and save the following TFL registration file, which references the tfl_table_cblas_mmult_double table.

The file specifies that the TFL to be registered is named 'ANSI/ISO C BLAS Matrix Multiplication Operator Example' and consists of tfl_table_cblas_mmult_double, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_cblas_mmult_double
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'ANSI/ISO C BLAS Matrix Multiplication Operator Example';
  thisTfl.Description = 'Demonstration of C BLAS mmult operator replacement';
  thisTfl.TableList = {'tfl_table_cblas_mmult_double'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

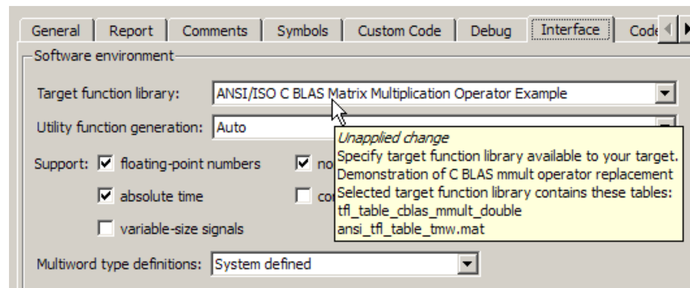
Place this sl_customization.m file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

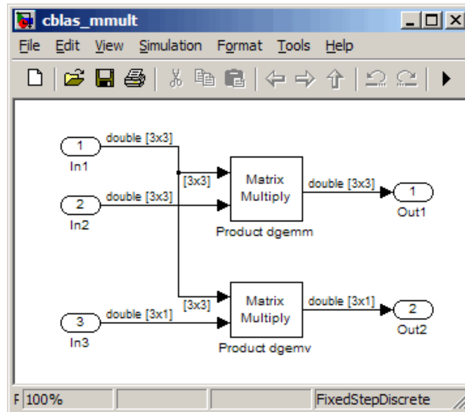
4 With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open an ERT-based Simulink model and navigate to the **Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

Note If you hover over the selected library with the cursor, a tool tip appears. This tip provides information derived from your TFL registration file, such as the TFL description and the list of tables it contains.



Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `ANSI/ISO C BLAS Matrix Multiplication Operator Example`.

5 Create an ERT-based model with two Product blocks, such as the following:



a. For each Product block, set the block parameter **Multiplication** to the value `Matrix(*)`.

b. Configure the **Signal Attributes** for the `In1`, `In2`, and `In3` source blocks. For `In1` and `In2`, set **Port dimensions** to `[3 3]` and set the **Data type** to `double`. For `In3`, set **Port dimensions** to `[3 1]` and set the **Data type** to `double`.

c. Also, go to the **Solver** pane of the Configuration Parameters dialog box and select a fixed-step, discrete solver with a fixed-step size such as `0.1`. Apply the changes.

d. Save the model. In this example, the model is saved to the name `cblas_mmult.mdl`.

e. Make sure that the TFL you registered, `ANSI/ISO C BLAS Matrix Multiplication Operator Example`, is selected for this model.

6. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

7. Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the top Product block and select **Code Generation > Navigate to Code**. This selection highlights the Product block code within the model step function in `cblas_mmult.c`. In this case, code containing the matrix multiplication

operator has been replaced with a call to `cblas_dgemm` in the generated code.

```
31  /* Model step function */
32  void cblas_mmult_step(void)
33  {
34    /* Outport: '<Root>/Out1' incorporates:
35     *  Inport: '<Root>/In1'
36     *  Inport: '<Root>/In2'
37     *  Product: '<Root>/Product dgemm'
38     */
39    cblas_dgemm(102, 111, 111, 3, 3, 3, 1.0, &cblas_mmult_U.In1[0], 3,
40                &cblas_mmult_U.In2[0], 3, 0.0, &cblas_mmult_Y.Out1[0], 3);
41
42    /* Outport: '<Root>/Out2' incorporates:
43     *  Inport: '<Root>/In1'
44     *  Inport: '<Root>/In3'
45     *  Product: '<Root>/Product dgemv'
46     */
47    cblas_dgemv(102, 111, 3, 3, 1.0, &cblas_mmult_U.In1[0], 3, &cblas_mmult_U.In3
48                [0], 1, 0.0, &cblas_mmult_Y.Out2[0], 1);
49  }
50
51  /* Model initialize function */
52  void cblas_mmult_initialize(void)
53  {
54    /* Registration code */
```

Mapping Fixed-Point Operators to Target-Specific Implementations

- "Overview of Fixed-Point Operator Replacement" on page 31-78
- "Fixed-Point Numbers and Arithmetic" on page 31-80
- "Creating Fixed-Point Operator Entries" on page 31-86
- "Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling" on page 31-89
- "Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling" on page 31-92
- "Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)" on page 31-95
- "Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)" on page 31-98
- "Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)" on page 31-102
- "Mapping Data Type Conversion (Cast) Operations to Target-Specific Implementations" on page 31-105
- "Mapping Fixed-Point Shift Left Operations to Target-Specific Implementations" on page 31-109

Overview of Fixed-Point Operator Replacement

The Embedded Coder software supports TFL-based function replacement for the following scalar operations on fixed-point data types:

Operator	Key
Addition (+)	RTW_OP_ADD
Subtraction (-)	RTW_OP_MINUS
Multiplication (*)	RTW_OP_MUL
Division (/)	RTW_OP_DIV
Data type conversion (cast)	RTW_OP_CAST

Operator	Key
Shift left (<<)	`RTW_OP_SL`
Shift right (>>)	`RTW_OP_SRA` (arithmetic) [15] `RTW_OP_SRL` (logical)

Fixed-point operator table entries can be defined as matching:

- A specific binary-point-only scaling combination on the operator inputs and output.
- A specific [slope bias] scaling combination on the operator inputs and output.
- Relative scaling or net slope between multiplication or division operator inputs and output.

  Use these methods to map a range of slope and bias values to a replacement function for multiplication or division.
- Equal slope and zero net bias across addition or subtraction operator inputs and output.

  Use this method to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

15. TFLs that provide arithmetic shift right implementations should also provide logical shift right implementations, because some arithmetic shift rights are converted to logical shift rights during code generation.

Note

- The demo `rtwdemo_tflfixpt` demonstrates these replacements and provides example tables that can be used as a starting point for customization.
- Using fixed-point data types in a model requires a Simulink Fixed Point license.
- The fixed-point terminology used in this section is defined and explained in the *Simulink Fixed Point User's Guide.* See especially "Fixed-Point Numbers" and "Arithmetic Operations".

Fixed-Point Numbers and Arithmetic

Fixed-point numbers use integers and integer arithmetic to represent real numbers and arithmetic with the following encoding scheme:

$$V = \tilde{V} = SQ + B$$

where

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value that results from fixed-point representation.
- Q is an integer that encodes $\tilde{V}$, referred to as the *quantized integer*.
- S is a coefficient of Q, referred to as the *slope*.
- B is an additive correction, referred to as the *bias*.

The general equation for an operation between fixed-point operands is as follows:

$$\left(S_O Q_O + B_O\right) = \left(S_1 Q_1 + B_1\right) < op > (S_2 Q_2 + B_2)$$

The objective of TFL fixed-point operator replacement is to replace an operator that accepts and returns fixed-point or integer inputs and output with a function that accepts and returns built-in C numeric data types (not fixed-point data types). The following sections provide additional programming information for each supported operator.

Addition

The operation $V_0 = V_1 + V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 + \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

If an addition replacement function is defined such that the scaling on the operands and sum are equal and the net bias

$$\left(\frac{B_1 + B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_add_s8_s8` that adds two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Subtraction

The operation $V_0 = V_1 - V_2$ implies that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1 - \left(\frac{S_2}{S_0}\right)Q_2 + \left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

If a subtraction replacement function is defined such that the scaling on the operands and difference are equal and the net bias

$$\left(\frac{B_1 - B_2 - B_0}{S_0}\right)$$

is zero (for example, a function `s8_sub_s8_s8` that subtracts two signed 8-bit values and produces a signed 8-bit result), then the TFL operator entry must set the operator entry parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias` to `true`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Multiplication

There are different ways to specify multiplication replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this, use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different TFL entry. For this, use a relative scaling factor (RSF) entry or a net slope entry:

- **Relative scaling factor (RSF) entry:**

  The operation $V_0 = V_1 * V_2$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = (S_1 Q_1)(S_2 Q_2)$$

$$Q_0 = \left(\frac{S_1 S_2}{S_0}\right) Q_1 Q_1$$

$$Q_0 = S_n Q_1 Q$$

  where S_n is the net slope.

  Multiplication replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s8_mul_s8_u8_rsf0p125` can multiply a signed 8-bit value by a factor of `[0 ... 0.1245]` and produce a signed 8-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

To match a multiplication operation to the `s8_mul_s8_u8_rsf0p125` replacement function, 0 <= $S_n Q_2$ <= 2^{-3}. Substituting the maximum integer value for Q_2 results in the following match criteria: When $S_n 2^8 = 2^{-3}$, or $S_n = 2^{-11}$, TFL replacement processing maps the multiplication operation to the `s8_mul_s8_u8_rsf0p125` function.

To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for F and E are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s8_mul_s8_u8_rsf0p125` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -3.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

- **Net slope entry:**

  Net slope entries are similar to the relative scaling factor entry described above. The difference is the match criteria. For a net slope entry, the net slope of the call-site operation, S_n, must match the specified net slope, $S_n = F2^E$, without regard to the maximum integer value. Specify the desired net slope F and E values using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Division

There are different ways to specify division replacements. The most direct way is to specify an exact match of the input and output types. This is feasible if a model contains only a few (known) slope and bias combinations. For this,

use the `TflCOperationEntry` class and specify the exact values of slope and bias on each argument. For scenarios where there are numerous slope/bias combinations, it is not feasible to specify each value with a different TFL entry. For this, use a relative scaling factor (RSF) entry or a net slope entry:

- **Relative scaling factor (RSF) entry:**

  The operation $V_0 = (V_1 \ / \ V_2)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{S_2 Q_2} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{Q_2} \right)$$

  where S_n is the net slope.

  As with multiplication, division replacement functions may be defined such that all scaling is contained by a single operand. For example, a replacement function `s16_rsf0p5_div_s16_s16` can divide a signed 16<<16 value by a signed 16-bit value and produce a signed 16-bit result. The following discussion describes how to convert the slope on each operand into a net factor.

  To match a division operation to the `s16_rsf0p5_div_s16_s16` replacement function, $0 <= S_n Q_1 <= 2^{-1}$. Substituting the maximum integer value for Q_1 results in the following match criteria: When $S_n 2^{15} = 2^{-1}$, or $S_n = 2^{-16}$, TFL replacement processing maps the division operation to the `s8_mul_s8_u8_rsf0p125` function.

  To accomplish this mapping, the TFL operator entry must define a *relative scaling factor*, $F2^E$, where the values for F and E are provided using operator entry parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.) For the `s16_rsf0p5_div_s16_s16` function, the `RelativeScalingFactorF` would be set to 1 and the `RelativeScalingFactorE` would be set to -1.

Note When an operator entry specifies `RelativeScalingFactorF` and `RelativeScalingFactorE`, zero bias is implied for the inputs and output.

- **Net slope entry:**

  Net slope entries are similar to the relative scaling factor entry described above. The difference is the match criteria. For a net slope entry, the net slope of the call-site operation, S_n, must match the specified net slope, $S_n = F2^E$, without regard to the maximum integer value. Specify the desired net slope F and E values using operator entry parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`. (For parameter descriptions, see the reference page for the function `setTflCOperationEntryParameters`.)

Note When an operator entry specifies `NetSlopeAdjustmentFactor` and `NetFixedExponent`, matching entries must have arguments with zero bias.

Data Type Conversion (Cast)

The data type conversion operation $V_0 = V_1$ implies, for binary-point-only scaling, that

$$Q_0 = \left(\frac{S_1}{S_0}\right)Q_1$$

$$Q_0 = S_n Q_1$$

where S_n is the net slope.

Shift

The shift left or shift right operation $V_0 = (V_1 \ / \ 2^n)$ implies, for binary-point-only scaling, that

$$S_0 Q_0 = \left(\frac{S_1 Q_1}{2^n} \right)$$

$$Q_0 = \left(\frac{S_1}{S_0} \right) + \left(\frac{Q_1}{2^n} \right)$$

$$Q_0 = S_n \left(\frac{Q_1}{2^n} \right)$$

where S_n is the net slope.

Creating Fixed-Point Operator Entries

To create TFL table entries for fixed-point operators, you use the "General Method for Creating Function and Operator Entries" on page 31-22 and specify fixed-point parameter/value pairs to the functions shown in the following table.

Function	Description
`createAndAddConceptualArg`	Create conceptual argument from specified properties and add to conceptual arguments for TFL table entry
`createAndAddImplementationArg`	Create implementation argument from specified properties and add to implementation arguments for TFL table entry
`createAndSetCImplementationReturn`	Create implementation return argument from specified properties and add to implementation for TFL table entry
`setTflCOperationEntryParameters`	Set specified parameters for operator entry in TFL table

The following table maps some common methods of matching TFL fixed-point operator table entries to the associated fixed-point parameters that you need to specify in your TFL table definition file.

To match...	Instantiate class...	Minimally specify parameters...
A specific binary-point-only scaling combination on the operator inputs and output See “Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling” on page 31-89.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point binary-point-only scaling. • `FractionLength`: Specify a fraction length (for example, 3).
A specific [slope bias] scaling combination on the operator inputs and output See “Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling” on page 31-92.	`RTW.TflCOperationEntry`	`createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `true`. • `CheckBias`: Specify the value `true`. • `DataTypeMode` (or `DataType/Scaling` equivalent): Specify fixed-point [slope bias] scaling. • `Slope` (or `SlopeAdjustmentFactor/-FixedExponent` equivalent): Specify a slope value (for example, 15). • `Bias`: Specify a bias value (for example, 2).

<table>
<tr><th>To match...</th><th>Instantiate class...</th><th>Minimally specify parameters...</th></tr>
<tr><td>Relative scaling between operator inputs and output (multiplication and division)

See “Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)” on page 31-95.</td><td>RTW.TflCOperationEntry-Generator</td><td>setTflCOperationEntryParameters function:

• RelativeScalingFactorF: Specify the slope adjustment factor (F) part of the relative scaling factor, $F2^E$ (for example, 1.0).
• RelativeScalingFactorE: Specify the fixed exponent (E) part of the relative scaling factor, $F2^E$ (for example, -3.0).

createAndAddConceptualArg function:

• CheckSlope: Specify the value false.
• CheckBias: Specify the value false.
• DataType: Specify the value 'Fixed'.</td></tr>
<tr><td>Net slope between operator inputs and output (multiplication and division)

See “Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)” on page 31-98.</td><td>RTW.TflCOperationEntry-Generator_NetSlope</td><td>setTflCOperationEntryParameters function:

• NetSlopeAdjustmentFactor: Specify the slope adjustment factor (F) part of the net slope, $F2^E$ (for example, 1.0).
• NetFixedExponent: Specify the fixed exponent (E) part of the net slope, $F2^E$ (for example, -3.0).

createAndAddConceptualArg function:

• CheckSlope: Specify the value false.
• CheckBias: Specify the value false.
• DataType: Specify the value 'Fixed'.</td></tr>
</table>

To match...	Instantiate class...	Minimally specify parameters...
Equal slope and zero net bias across operator inputs and output (addition and subtraction) See "Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)" on page 31-102.	`RTW.TflCOperationEntry-Generator`	`setTflCOperationEntryParameters` function: • `SlopesMustBeTheSame`: Specify the value `true`. • `MustHaveZeroNetBias`: Specify the value `true`. `createAndAddConceptualArg` function: • `CheckSlope`: Specify the value `false`. • `CheckBias`: Specify the value `false`.

Example: Creating Fixed-Point Operator Entries for Binary-Point-Only Scaling

TFL table entries for operations on fixed-point data types can be defined as matching a specific binary-point-only scaling combination on the operator inputs and output. These binary-point-only scaling entries can map the specified binary-point-scaling combination to a replacement function for addition, subtraction, multiplication, or division.

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for multiplication of fixed-point data types where arguments are specified with binary-point-only scaling. In this example:

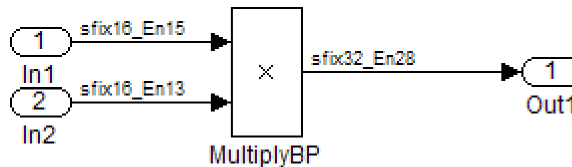
- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (multiplication), the saturation mode (saturate on overflow), the rounding mode (unspecified), and the name of the replacement function (`s32_mul_s16_s16_binarypoint`).

- The function createAndAddConceptualArg is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is binary-point-only scaling, and its derived slope and bias values must exactly match the call-site slope and bias values. The output argument is 32 bits, signed, with a fraction length of 28, while the input arguments are 16 bits, signed, with fraction lengths of 15 and 13.
- The functions createAndSetCImplementationReturn and createAndAddImplementationArg are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output argument is 32 bits and signed (int32) and the input arguments are 16 bits and signed (int16).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                        'RTW_OP_MUL', ...
                    'Priority',                   90, ...
                    'SaturationMode',             'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',               'RTW_ROUND_UNSPECIFIED', ...
                    'ImplementationName',         's32_mul_s16_s16_binarypoint', ...
                    'ImplementationHeaderFile', 's32_mul_s16_s16_binarypoint.h', ...
                    'ImplementationSourceFile', 's32_mul_s16_s16_binarypoint.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric',...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: binary point scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     32, ...
                          'FractionLength', 28);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
```

```
                             'CheckSlope',     true, ...
                             'CheckBias',      true, ...
                             'DataTypeMode',   'Fixed-point: binary point scaling', ...
                             'IsSigned',       true, ...
                             'WordLength',     16, ...
                             'FractionLength', 15);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                             'Name',           'u2', ...
                             'IOType',         'RTW_IO_INPUT', ...
                             'CheckSlope',     true, ...
                             'CheckBias',      true, ...
                             'DataTypeMode',   'Fixed-point: binary point scaling', ...
                             'IsSigned',       true, ...
                             'WordLength',     16, ...
                             'FractionLength', 13);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                 'Name',           'y1', ...
                                 'IOType',         'RTW_IO_OUTPUT', ...
                                 'IsSigned',       true, ...
                                 'WordLength',     32, ...
                                 'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                 'Name',           'u1', ...
                                 'IOType',         'RTW_IO_INPUT', ...
                                 'IsSigned',       true, ...
                                 'WordLength',     16, ...
                                 'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                                 'Name',           'u2', ...
                                 'IOType',         'RTW_IO_INPUT', ...
                                 'IsSigned',       true, ...
                                 'WordLength',     16, ...
                                 'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in "Example: Mapping Scalar Operators to Target-Specific Implementations" on page 31-43, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13)`
- In the Product block:
  - Set **Output data type** to `fixdt(1,32,28)`
  - Select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for [Slope Bias] Scaling

TFL table entries for operations on fixed-point data types can be defined as matching a specific [slope bias] scaling combination on the operator inputs and output. These [slope bias] scaling entries can map the specified [slope bias] combination to a replacement function for addition, subtraction, multiplication, or division.

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for division of fixed-point data types where arguments are specified using [slope bias] scaling. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation

(division), the saturation mode (saturate on overflow), the rounding mode (round to ceiling), and the name of the replacement function (s16_div_s16_s16_slopebias).

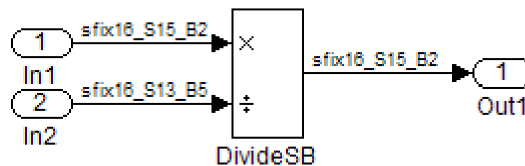
- The function createAndAddConceptualArg is called to create and add conceptual output and input arguments to the operator entry. Each argument specifies that the data type is fixed-point, the mode is [slope bias] scaling, and its specified slope and bias values must exactly match the call-site slope and bias values. The output argument and input arguments are 16 bits, signed, each with specific [slope bias] specifications.
- The functions createAndSetCImplementationReturn and createAndAddImplementationArg are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (int16).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                      'RTW_OP_DIV', ...
                    'Priority',                 90, ...
                    'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                    'RoundingMode',             'RTW_ROUND_CEILING', ...
                    'ImplementationName',       's16_div_s16_s16_slopebias', ...
                    'ImplementationHeaderFile', 's16_div_s16_s16_slopebias.h', ...
                    'ImplementationSourceFile', 's16_div_s16_s16_slopebias.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          15, ...
                          'Bias',           2);
```

```
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          15, ...
                          'Bias',           2);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'CheckSlope',     true, ...
                          'CheckBias',      true, ...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling', ...
                          'IsSigned',       true, ...
                          'WordLength',     16, ...
                          'Slope',          13, ...
                          'Bias',           5);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',           'y1', ...
                              'IOType',         'RTW_IO_OUTPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u1', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u2', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
```

```
                                      'WordLength',     16, ...
                                      'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in “Example: Mapping Scalar Operators to Target-Specific Implementations” on page 31-43, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(1,16,15,2)`
- Set the Inport 2 **Data type** to `fixdt(1,16,13,5)`
- In the Divide block:
  - Set **Output data type** to `Inherit:  Inherit via back propagation`
  - Set **Integer rounding mode** to `Ceiling`
  - Select the option **Saturate on integer overflow**

Example: Creating Fixed-Point Operator Entries for Relative Scaling (Multiplication and Division)

TFL table entries for multiplication or division of fixed-point data types can be defined as matching relative scaling between operator inputs and output. These relative scaling entries can map a range of slope and bias values to a replacement function for multiplication or division.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 31-22 to create a TFL table entry for division of fixed-point data types using a relative scaling factor. In this example:

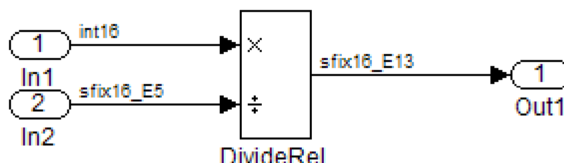
- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator` class, which provides access to the fixed-point parameters `RelativeScalingFactorF` and `RelativeScalingFactorE`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (division), the saturation mode (saturation off), the rounding mode (round to ceiling), and the name of the replacement function (`s16_div_s16_s16_rsf0p125`). Additionally, `RelativeScalingFactorF` and `RelativeScalingFactorE` are used to specify the `F` and `E` parts of the relative scaling factor $F2^E$.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as fixed-point, 16 bits, and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and signed (`int16`).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                        'RTW_OP_DIV', ...
                    'Priority',                   90, ...
                    'SaturationMode',             'RTW_WRAP_ON_OVERFLOW', ...
                    'RoundingMode',               'RTW_ROUND_CEILING', ...
                    'RelativeScalingFactorF',     1.0, ...
                    'RelativeScalingFactorE',     -3.0, ...
                    'ImplementationName',         's16_div_s16_s16_rsf0p125', ...
                    'ImplementationHeaderFile', 's16_div_s16_s16_rsf0p125.h', ...
                    'ImplementationSourceFile', 's16_div_s16_s16_rsf0p125.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
```

```
                          'Name',          'y1', ...
                          'IOType',        'RTW_IO_OUTPUT', ...
                          'CheckSlope',    false, ...
                          'CheckBias',     false, ...
                          'DataType',      'Fixed', ...
                          'IsSigned',      true, ...
                          'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',          'u1', ...
                          'IOType',        'RTW_IO_INPUT', ...
                          'CheckSlope',    false, ...
                          'CheckBias',     false, ...
                          'DataType',      'Fixed', ...
                          'IsSigned',      true, ...
                          'WordLength',    16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',          'u2', ...
                          'IOType',        'RTW_IO_INPUT', ...
                          'CheckSlope',    false, ...
                          'CheckBias',     false, ...
                          'DataType',      'Fixed', ...
                          'IsSigned',      true, ...
                          'WordLength',    16);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',          'y1', ...
                              'IOType',        'RTW_IO_OUTPUT', ...
                              'IsSigned',      true, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',          'u1', ...
                              'IOType',        'RTW_IO_INPUT', ...
                              'IsSigned',      true, ...
                              'WordLength',    16, ...
                              'FractionLength', 0);
```

```
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u2', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       true, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in "Example: Mapping Scalar Operators to Target-Specific Implementations" on page 31-43, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `int16`
- Set the Inport 2 **Data type** to `fixdt(1,16,-5)`
- In the Divide block:
  - Set **Output data type** to `fixdt(1,16,-13)`
  - Set **Integer rounding mode** to `Ceiling`

Example: Creating Fixed-Point Operator Entries for Net Slope (Multiplication and Division)

TFL table entries for multiplication or division of fixed-point data types can be defined as matching net slope between operator inputs and output. These net slope entries can map a range of slope and bias values to a replacement function for multiplication or division.

The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry for division of fixed-point data types using a net slope. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator_NetSlope` class, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (division), the saturation mode (wrap on overflow), the rounding mode (unspecified), and the name of the replacement function (`user_div_*`). Additionally, `NetSlopeAdjustmentFactor` and `NetFixedExponent` are used to specify the `F` and `E` parts of the net slope $F2^E$.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The function `getTflArgFromString` is called to create implementation output and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
hTable = RTW.TflTable;

wv = [16,32];
for iy = 1:2
  for inum = 1:2
    for iden = 1:2
      hTable = getDivOpEntry(hTable, ...
                       fixdt(1,wv(iy)),fixdt(1,wv(inum)),fixdt(1,wv(iden)));
    end
  end
end


%----------------------------------------------------------
```

```
function hTable = getDivOpEntry(hTable,dty,dtnum,dtden)
%----------------------------------------------------------
% Create an entry for division of fixed-point data types where
% arguments are specified using Slope and Bias scaling
% Saturation on, Rounding unspecified

funcStr = sprintf('user_div_%s_%s_%s',...
        typeStrFunc(dty),...
        typeStrFunc(dtnum),...
        typeStrFunc(dtden));

op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
setTflCOperationEntryParameters(op_entry, ...
                                'Key',                        'RTW_OP_DIV', ...
                                'Priority',                   90, ...
                                'SaturationMode',             'RTW_WRAP_ON_OVERFLOW',...
                                'RoundingMode',               'RTW_ROUND_UNSPECIFIED',...
                                'NetSlopeAdjustmentFactor', 1.0, ...
                                'NetFixedExponent',           0.0, ...
                                'ImplementationName',         funcStr, ...
                                'ImplementationHeaderFile', [funcStr,'.h'], ...
                                'ImplementationSourceFile', [funcStr,'.c'] );

createAndAddConceptualArg(op_entry, ...
                          'RTW.TflArgNumeric', ...
                          'Name',           'y1',...
                          'IOType',         'RTW_IO_OUTPUT',...
                          'CheckSlope',     false,...
                          'CheckBias',      false,...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling',...
                          'IsSigned',       dty.Signed,...
                          'WordLength',     dty.WordLength,...
                          'Bias',           0);

createAndAddConceptualArg(op_entry, ...
                          'RTW.TflArgNumeric',...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT',...
                          'CheckSlope',     false,...
                          'CheckBias',      false,...
```

```
                          'DataTypeMode',   'Fixed-point: slope and bias scaling',...
                          'IsSigned',       dtnum.Signed,...
                          'WordLength',     dtnum.WordLength,...
                          'Bias',           0);

createAndAddConceptualArg(op_entry, ...
                          'RTW.TflArgNumeric', ...
                          'Name',           'u2', ...
                          'IOType',         'RTW_IO_INPUT',...
                          'CheckSlope',     false,...
                          'CheckBias',      false,...
                          'DataTypeMode',   'Fixed-point: slope and bias scaling',...
                          'IsSigned',       dtden.Signed,...
                          'WordLength',     dtden.WordLength,...
                          'Bias',           0);

arg = getTflArgFromString(hTable, 'y1', typeStrBase(dty));
op_entry.Implementation.setReturn(arg);

arg = getTflArgFromString(hTable, 'u1', typeStrBase(dtnum));
op_entry.Implementation.addArgument(arg);

arg = getTflArgFromString(hTable, 'u2',typeStrBase(dtden));
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);

%-------------------------------------------------------------
function str = typeStrFunc(dt)
%-------------------------------------------------------------

if dt.Signed
    sstr = 's';
else
    sstr = 'u';
end
str = sprintf('%s%d',sstr,dt.WordLength);

%-------------------------------------------------------------
function str = typeStrBase(dt)
```

```
%------------------------------------------------------------

if dt.Signed
    sstr = ;
else
    sstr = 'u';
end
str = sprintf('%sint%d',sstr,dt.WordLength);
```

Example: Creating Fixed-Point Operator Entries for Equal Slope and Zero Net Bias (Addition and Subtraction)

TFL table entries for addition or subtraction of fixed-point data types can be defined as matching relative slope and bias values (equal slope and zero net bias) across operator inputs and output. These entries allow you to disregard specific slope and bias values and map relative slope and bias values to a replacement function for addition or subtraction.

The following example uses the method described in “General Method for Creating Function and Operator Entries” on page 31-22 to create a TFL table entry for addition of fixed-point data types where slopes must be equal and net bias must be zero across the operator inputs and output. In this example:

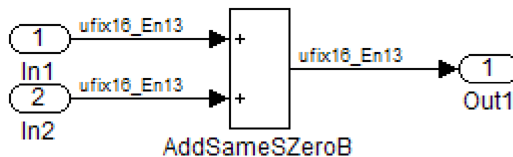
- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator` class, which provides access to the fixed-point parameters `SlopesMustBeTheSame` and `MustHaveZeroNetBias`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (addition), the saturation mode (saturation off), the rounding mode (unspecified), and the name of the replacement function (`u16_add_SameSlopeZeroBias`). Additionally, `SlopesMustBeTheSame` and `MustHaveZeroNetBias` are set to `true` to indicate that slopes must be equal and net bias must be zero across the addition inputs and output.
- The function `createAndAddConceptualArg` is called to create and add conceptual output and input arguments to the operator entry. Each argument is specified as 16 bits and unsigned. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.

- The functions createAndSetCImplementationReturn and createAndAddImplementationArg are called to create and add implementation output and input arguments to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types). In this case, the output and input arguments are 16 bits and unsigned (uint16).

```
hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntryGenerator;
setTflCOperationEntryParameters(op_entry, ...
                      'Key',                        'RTW_OP_ADD', ...
                      'Priority',                    90, ...
                      'SaturationMode',              'RTW_WRAP_ON_OVERFLOW', ...
                      'RoundingMode',                'RTW_ROUND_UNSPECIFIED', ...
                      'SlopesMustBeTheSame',         true, ...
                      'MustHaveZeroNetBias',         true, ...
                      'ImplementationName',          'u16_add_SameSlopeZeroBias', ...
                      'ImplementationHeaderFile', 'u16_add_SameSlopeZeroBias.h', ...
                      'ImplementationSourceFile', 'u16_add_SameSlopeZeroBias.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                            'Name',           'y1', ...
                            'IOType',         'RTW_IO_OUTPUT', ...
                            'CheckSlope',     false, ...
                            'CheckBias',      false, ...
                            'IsSigned',       false, ...
                            'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                            'Name',           'u1', ...
                            'IOType',         'RTW_IO_INPUT', ...
                            'CheckSlope',     false, ...
                            'CheckBias',      false, ...
                            'IsSigned',       false, ...
                            'WordLength',     16);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                            'Name',           'u2', ...
                            'IOType',         'RTW_IO_INPUT', ...
```

```
                              'CheckSlope',     false, ...
                              'CheckBias',      false, ...
                              'IsSigned',       false, ...
                              'WordLength',     16);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',           'y1', ...
                              'IOType',         'RTW_IO_OUTPUT', ...
                              'IsSigned',       false, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u1', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       false, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                              'Name',           'u2', ...
                              'IOType',         'RTW_IO_INPUT', ...
                              'IsSigned',       false, ...
                              'WordLength',     16, ...
                              'FractionLength', 0);

addEntry(hTable, op_entry);
```

To generate code using this table entry, you can follow the general procedure in "Example: Mapping Scalar Operators to Target-Specific Implementations" on page 31-43, substituting in the code above and an ERT-based model such as the following:



For this model,

- Set the Inport 1 **Data type** to `fixdt(0,16,13)`
- Set the Inport 2 **Data type** to `fixdt(0,16,13)`
- In the Add block:
  - Verify that **Output data type** is set to its default, `Inherit via internal rule`
  - Set **Integer rounding mode** to `Zero`

Mapping Data Type Conversion (Cast) Operations to Target-Specific Implementations

- "Example: Creating a TFL Entry to Replace Casts From `int32` To `int16`" on page 31-105
- "Example: Creating a TFL Entry to Replace Fixed-Point Casts Using Net Slope" on page 31-106

You can use TFL table entries to replace the default generated code for data type conversion (cast) operations with calls to optimized functions.

For details of the arithmetic supported for replacement of data type conversion, see the data type conversion (cast) subsection of "Fixed-Point Numbers and Arithmetic" on page 31-80.

Example: Creating a TFL Entry to Replace Casts From `int32` To `int16`. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry to replace `int32` to `int16` data type conversion (cast) operations. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (cast), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (`my_sat_cast`).

- The function getTflArgFromString is called to create an int16 output argument, which is then added to the operator entry both as the first conceptual argument and the implementation return argument.
- The function getTflArgFromString is called to create an int32 input argument, which is then added to the operator entry both as the second conceptual argument and the sole implementation input argument.

```
hTable = RTW.TflTable;

% Create an int16 to int32 cast replacement
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                        'RTW_OP_CAST', ...
    'Priority',                   50, ...
    'ImplementationName',         'my_sat_cast', ...
    'SaturationMode',             'RTW_SATURATE_ON_OVERFLOW', ...
    'RoundingMode',               'RTW_ROUND_FLOOR', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

% Create int16 arg as conceptual arg 1 and implementation return
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);

% Create int32 arg as conceptual arg 2 and implementation input arg 1
arg = getTflArgFromString(hTable, 'u1', 'int32');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

Example: Creating a TFL Entry to Replace Fixed-Point Casts Using Net Slope. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry to replace data type conversions (casts) of fixed-point data types using a net slope. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntryGenerator_NetSlope` class, which provides access to the fixed-point parameters `NetSlopeAdjustmentFactor` and `NetFixedExponent`.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (cast), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (`my_fxp_cast`). Additionally, `NetSlopeAdjustmentFactor` and `NetFixedExponent` are used to specify the `F` and `E` parts of the net slope $F2^{E}$.
- The function `createAndAddConceptualArg` is called to create conceptual output and input arguments that are added to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions `createAndSetCImplementationReturn` and `createAndAddImplementationArg` are called to create implementation return and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).

```
hTable = RTW.TflTable;

% Create a fixed-point cast replacement using a NetSlope entry
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
   'Key',                       'RTW_OP_CAST', ...
   'Priority',                  50, ...
   'SaturationMode',            'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingMode',              'RTW_ROUND_FLOOR', ...
   'NetSlopeAdjustmentFactor',  1.0, ...
   'NetFixedExponent',          (OutFL - InFL), ...
```

```
   'SaturationMode',            'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingMode',              'RTW_ROUND_FLOOR', ...
   'ImplementationName',        'my_fxp_cast', ...
   'ImplementationHeaderFile',  'some_hdr.h', ...
   'ImplementationSourceFile',  'some_hdr.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'CheckSlope',   false, ...
                          'CheckBias',    false, ...
                          'DataTypeMode', 'Fixed-point: binary point scaling', ...
                          'IsSigned',     OutSgn, ...
                          'WordLength',   OutWL, ...
                          'FractionLength',OutFL);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'u1', ...
                          'IOType',       'RTW_IO_INPUT', ...
                          'CheckSlope',   false, ...
                          'CheckBias',    false, ...
                          'DataTypeMode', 'Fixed-point: binary point scaling', ...
                          'IsSigned',     InSgn, ...
                          'WordLength',   InWL, ...
                          'FractionLength',InFL);

createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'IsSigned',       OutSgn, ...
                          'WordLength',     OutWL, ...
                          'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric',...
                          'Name',           'u1', ...
                          'IOType',         'RTW_IO_INPUT', ...
                          'IsSigned',       InSgn, ...
                          'WordLength',     InWL, ...
                          'FractionLength', 0);
```

```
addEntry(hTable, op_entry);
```

Mapping Fixed-Point Shift Left Operations to Target-Specific Implementations

- "Example: Creating a TFL Entry to Replace Shift Lefts for `int16` Data" on page 31-109
- "Example: Creating a TFL Entry to Replace Fixed-Point Shift Lefts Using Net Slope" on page 31-110

You can use TFL table entries to replace the default generated code for `<<` (shift left) operations with calls to optimized functions.

For details of the arithmetic supported for replacement of shift-left operations, see the shift left subsection of "Fixed-Point Numbers and Arithmetic" on page 31-80.

Example: Creating a TFL Entry to Replace Shift Lefts for `int16` Data. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry to replace `<<` (shift left) operations for `int16` data. In this example:

- The TFL operator entry is instantiated using the `RTW.TflCOperationEntry` class.
- The function `setTflCOperationEntryParameters` is called to set operator entry parameters. These parameters include the type of operation (shift left) and the name of the replacement function (`my_shift_left`).
- The function `getTflArgFromString` is called to create an `int16` output argument, which is then added to the operator entry both as the first conceptual argument and the implementation return argument.
- The function `getTflArgFromString` is called to create an `int16` input argument, which is then added to the operator entry both as the second conceptual argument and the first implementation input argument.
- The function `getTflArgFromString` is called to create an `int8` input argument, which is then added to the operator entry both as the third conceptual argument and the second implementation input argument. This argument specifies the number of bits to shift the previous input argument.

Since the argument type is not relevant, type checking is disabled by setting the `CheckType` property to `false`.

```
hTable = RTW.TflTable;

% Create a shift left replacement for int16 data
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
    'Key',                      'RTW_OP_SL', ...
    'Priority',                 50, ...
    'ImplementationName',       'my_shift_left', ...
    'ImplementationHeaderFile', 'some_hdr.h', ...
    'ImplementationSourceFile', 'some_hdr.c');

% Create int16 arg as conceptual arg 1 and implementation return
arg = getTflArgFromString(hTable, 'y1', 'int16');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
op_entry.Implementation.setReturn(arg);

% Create int16 arg as conceptual arg 2 and implementation input arg 1
arg = getTflArgFromString(hTable, 'u1', 'int16');
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

% Create int8 arg as conceptual arg 3 and implementation input arg 2
% Turn off type checking for number of bits to shift argument
arg = getTflArgFromString(hTable, 'u2', 'int8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

Example: Creating a TFL Entry to Replace Fixed-Point Shift Lefts Using Net Slope. The following example uses the method described in "General Method for Creating Function and Operator Entries" on page 31-22 to create a TFL table entry to replace << (shift left) operations for fixed-point data using a net slope. In this example:

- The TFL operator entry is instantiated using the RTW.TflCOperationEntryGenerator_NetSlope class, which provides access to the fixed-point parameters NetSlopeAdjustmentFactor and NetFixedExponent.
- The function setTflCOperationEntryParameters is called to set operator entry parameters. These parameters include the type of operation (shift left), the saturation mode (saturate on overflow), the rounding mode (toward negative infinity), and the name of the replacement function (my_fxp_shift_left). Additionally, NetSlopeAdjustmentFactor and NetFixedExponent are used to specify the F and E parts of the net slope $F2^E$.
- The function createAndAddConceptualArg is called to create conceptual output and input arguments that are added to the operator entry. Each argument is specified as fixed-point and signed. Also, each argument specifies that TFL replacement request processing should *not* check for an exact match to the call-site slope and bias values.
- The functions createAndSetCImplementationReturn and createAndAddImplementationArg are called to create implementation return and input arguments that are added to the operator entry. Implementation arguments must describe fundamental numeric data types (not fixed-point data types).
- The function getTflArgFromString is called to create a uint8 input argument, which is then added to the operator entry both as the third conceptual argument and the second implementation input argument. This argument specifies the number of bits to shift the previous input argument. Since the argument type is not relevant, type checking is disabled by setting the CheckType property to false.

```
hTable = RTW.TflTable;

% Create a fixed-point shift left replacement using a NetSlope entry
op_entry = RTW.TflCOperationEntryGenerator_NetSlope;
InFL = 2;
InWL = 16;
InSgn = true;
OutFL = 4;
OutWL = 32;
OutSgn = true;
setTflCOperationEntryParameters(op_entry, ...
```

```
   'Key',                        'RTW_OP_SL', ...
   'Priority',                   50, ...
   'SaturationMode',             'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingMode',               'RTW_ROUND_FLOOR', ...
   'NetSlopeAdjustmentFactor',  1.0, ...
   'NetFixedExponent',           (OutFL - InFL),...
   'SaturationMode',             'RTW_SATURATE_ON_OVERFLOW', ...
   'RoundingMode',               'RTW_ROUND_FLOOR', ...
   'ImplementationName',         'my_fxp_shift_left', ...
   'ImplementationHeaderFile',  'some_hdr.h', ...
   'ImplementationSourceFile',  'some_hdr.c');

% Create fixed-point arg as conceptual arg 1
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',          'y1', ...
                          'IOType',        'RTW_IO_OUTPUT', ...
                          'CheckSlope',   false, ...
                          'CheckBias',    false, ...
                          'DataTypeMode', 'Fixed-point: binary point scaling', ...
                          'IsSigned',     OutSgn, ...
                          'WordLength',   OutWL, ...
                          'FractionLength',OutFL);

% Create fixed-point arg as conceptual arg 2
createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',          'u1', ...
                          'IOType',        'RTW_IO_INPUT', ...
                          'CheckSlope',   false, ...
                          'CheckBias',    false, ...
                          'DataTypeMode', 'Fixed-point: binary point scaling', ...
                          'IsSigned',     InSgn, ...
                          'WordLength',   InWL, ...
                          'FractionLength',InFL);

% Create implementation return arg
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',           'y1', ...
                          'IOType',         'RTW_IO_OUTPUT', ...
                          'IsSigned',       OutSgn, ...
                          'WordLength',     OutWL, ...
```

```
                        'FractionLength', 0);

% Create implementation input arg 1
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
                        'Name',           'u1', ...
                        'IOType',         'RTW_IO_INPUT', ...
                        'IsSigned',       InSgn, ...
                        'WordLength',     InWL, ...
                        'FractionLength', 0);

% Create uint8 arg as conceptual arg 3 and implementation input arg 2
% Turn off type checking for number of bits to shift argument
arg = getTflArgFromString(hTable, 'u2', 'uint8');
arg.CheckType = false;
addConceptualArg(op_entry, arg);
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

Remapping Operator Outputs to Implementation Function Input Positions

If you need your generated code to meet a specific coding pattern or you want more flexibility, for example, to further improve performance, you have the option of remapping operator outputs to input positions in an implementation function argument list.

Note Remapping outputs to implementation function inputs is supported only for operator replacement.

For example, for a sum operation, the build process might generate code similar to the following:

```
rtY.Out1 = u8_add_u8_u8(rtU.In1, rtU.In2);
```

If you remap the output to the first input, the build process generates code similar to the following:

```
uint8_T rtb_Add8;
```

```
u8_add_u8_u8(&rtb_Add8, rtU.In1, rtU.In2);
rtY.Out1 = rtb_Add8;
```

To remap an operator output to an implementation function input for an existing TFL operator replacement entry, you modify the TFL table definition file as follows:

1. In the `setTflCOperationEntryParameters` function call for the operator replacement, specify the `SideEffects` parameter as `true`.

2. When defining the implementation function return, create a new `void` output argument, for example, `y2`.

3. When defining the implementation function arguments, set the operator output argument (for example, `y1`) as an additional input argument, marking its `IOType` as output, and make its type a pointer type.

For example, the following TFL table definition file for a sum operation has been modified to remap operator output `y1` as the first function input argument. The modified lines of code are shown in **bold** type. (This definition file generated the example remap code shown above.)

```
function hTable = tfl_table_add_uint8
%TFL_TABLE_ADD_UINT8 - Describe operator entry for a Target Function Library table.

hTable = RTW.TflTable;

% Create entry for addition of built-in uint8 data type
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                    'Key',                      'RTW_OP_ADD', ...
                    'Priority',                 90, ...
                    'ImplementationName',       'u8_add_u8_u8', ...
                    'ImplementationHeaderFile', 'u8_add_u8_u8.h', ...
                    'ImplementationSourceFile', 'u8_add_u8_u8.c', ...
                     SideEffects ,              true );

arg = getTflArgFromString(hTable, 'y1', 'uint8');
arg.IOType = 'RTW_IO_OUTPUT';
addConceptualArg(op_entry, arg);
```

```
arg = getTflArgFromString(hTable, 'u1', 'uint8');
addConceptualArg(op_entry, arg );

arg = getTflArgFromString(hTable, 'u2', 'uint8');
addConceptualArg(op_entry, arg );

% Create new void output y2
arg = getTflArgFromString(hTable,  y2 ,  void );
arg.IOType =  RTW_IO_OUTPUT ;
op_entry.Implementation.setReturn(arg);

% Set y1 as first input arg, mark IOType as output, and use pointer type
arg=getTflArgFromString(hTable,  y1 ,  uint8* );
arg.IOType =  RTW_IO_OUTPUT ;
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u1', 'uint8');
op_entry.Implementation.addArgument(arg);

arg=getTflArgFromString(hTable, 'u2', 'uint8');
op_entry.Implementation.addArgument(arg);

addEntry(hTable, op_entry);
```

Refining TFL Matching and Replacement Using Custom TFL Table Entries

- “Example: Customizing TFL Matching and Replacement for Operators” on page 31-117
- “Example: Customizing TFL Matching and Replacement for Functions” on page 31-126

During code generation for your model, the TFL replacement capability uses

- Preset match criteria to identify math functions and operators for which target-specific implementations should replace default implementations
- Preset replacement function signatures

However, preset match criteria and preset replacement function signatures might not be sufficient for all function and operator replacement needs. For example,

- You might want to replace an operator with a particular fixed-point implementation function only when fraction lengths are within a particular range.
- When a match is made, you might want to modify your replacement function signature based on compile-time information, such as passing fraction-length values into the function.

When you need to add extra logic into the TFL matching and replacement process, you can create custom TFL table entries. Custom entries allow you to specify additional match criteria and/or modify the replacement function signature to meet your application needs.

To create a custom TFL table entry, you perform the following steps:

1. Create a custom TFL entry class, derived from either `RTW.TflCOperationEntryML` (for operator replacement) or `RTW.TflCFunctionEntryML` (for function replacement).

2. In your derived class, implement a `do_match` method with a fixed preset signature as a MATLAB function. In your `do_match` method, you can provide either or both of the following customizations for use by TFL table entries that instantiate the class:

   a. Add any additional match criteria not provided by the base class. The base class provides a match based on argument number, argument name, signedness, word size, slope (if not wildcarded), bias (if not wildcarded), math modes such as saturation and rounding, and operator or function key. For example, you can accept a match only when additional size or range conditions are met.

   b. Modify the implementation signature by adding additional arguments or setting constant input argument values. For example, you can inject a constant value, such as an input's scaling value, as an additional argument to the replacement function.

3. Create TFL table entries that instantiate your custom TFL entry class.

4 Register a TFL containing the TFL table entries. The registered TFL is then available for selection in the **Interface** pane of the Simulink Configuration Parameters dialog box.

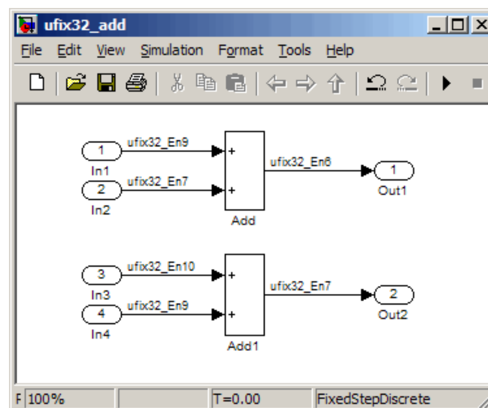
During code generation, the TFL matching process first tries to match function or operator call sites with the base class of your derived entry class. If a match is found, the software calls your `do_match` method to execute your additional match logic (if any) and your replacement function customizations (if any).

The following sections provide examples of creating custom TFL table entries to refine matching and replacement for operators and functions. For more examples, see the TFL demos page, including the demo model `rtwdemo_tflcustomentry`.

Example: Customizing TFL Matching and Replacement for Operators

This example demonstrates how to use custom TFL table entries to refine the matching and replacement logic for operators. In this example, a fixed-point addition replacement needs to be modified such that the implementation function passes in the fraction lengths of the input and output data types as arguments.

1 To exercise the custom TFL table entries created in this example, create an ERT-based model with one or more unsigned 32-bit fixed-point addition operations, such as the following:



For the purposes of this example, in the block parameters for both Add blocks, set **Integer rounding mode** to `Floor` and select the option **Saturate on integer overflow**.

2 Create a class folder using the name of your derived class, such as `@TflCustomOperationEntry`. Make sure the class folder is in the MATLAB search path or in the current working folder.

3 In the class folder, create and save the following class definition file, `TflCustomOperationEntry.m`. This file defines the class `TflCustomOperationEntry`, which is derived from the base class `RTW.TflCOperationEntryML`

The derived class defines a `do_match` method . In the `do_match` method signature,

- `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `TflCOperationEntry` handle.
- `hThis` is the handle to this object.
- `hCSO` is a handle to an object created by the code generator for the purpose of querying the TFL for a replacement.
- The remaining arguments are the number of bits for various data types of the current target.

The purpose of the `do_match` method is to add any additional match criteria not provided by the base class and make any desired modifications to the implementation signature. In this case, the `do_match` method can rely on the base class for checking word size and signedness, and additionally only needs to match the number of conceptual arguments to the value 3 (two inputs and one output) and the bias for each argument to the value 0. If a match is made, the method sets the return handle, removes slope and bias wildcarding from the conceptual arguments (since the match is for specific slope and bias values), and writes fraction-length values for the inputs and output into replacement function arguments 3, 4, and 5.

Note The three additional implementation function arguments for passing fraction lengths can be created and added either here in the class definition or in each TFL table entry definition that instantiates this class. In this example, the arguments are created and added in a TFL table definition file and set to specific values in the class definition code. For an example of creating and adding additional implementation function arguments in a class definition, see "Example: Customizing TFL Matching and Replacement for Functions" on page 31-126.

```
classdef TflCustomOperationEntry < RTW.TflCOperationEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong) %#ok
      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This will check additional data and perhaps modify
      % the implementation function.

      % The base class checks word size and signedness. Slopes and biases
      % have been wildcarded, so the only additional checking needed is
      % to make sure the biases are zero and that there are only three
      % conceptual arguments (one output, two inputs)

      ent = []; % default the return to empty, indicating the match failed.

      if length(hCSO.ConceptualArgs) == 3 && ...
          hCSO.ConceptualArgs(1).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(2).Type.Bias == 0 && ...
          hCSO.ConceptualArgs(3).Type.Bias == 0

        % Need to modify the default implementation. Since this is a
        % generator entry, a concrete entry is created using this entry
        % as a template. The type of entry being created is a standard
        % TflCOperationEntry. Using the standard operation entry is
```

```
              % sufficient, since it provides the necessary information, and
              % a custom match function will no longer be needed.
              ent = RTW.TflCOperationEntry(hThis);

              % Since this entry is modifying the implementation for specific
              % fraction-length values (arguments 3, 4, and 5), the conceptual argument
              % wildcards must be removed (the wildcards were inherited from the
              % generator when it was used as a template for the concrete entry).
              % This concrete entry is now for a specific slope and bias
              % (not for any slope and bias). The hCSO holds the correct
              % slope and bias values (created by the code generator).
              for idx=1:3
                ent.ConceptualArgs(idx).CheckSlope = true;
                ent.ConceptualArgs(idx).CheckBias = true;

                % Set the specific Slope and Biases
                ent.ConceptualArgs(idx).Type.Slope = hCSO.ConceptualArgs(idx).Type.Slope;
                ent.ConceptualArgs(idx).Type.Bias = 0;
              end

              % Set the fraction-length values in the implementation function.
              ent.Implementation.Arguments(3).Value = ...
                 -1.0*hCSO.ConceptualArgs(2).Type.FixedExponent;
              ent.Implementation.Arguments(4).Value = ...
                 -1.0*hCSO.ConceptualArgs(3).Type.FixedExponent;
              ent.Implementation.Arguments(5).Value = ...
                 -1.0*hCSO.ConceptualArgs(1).Type.FixedExponent;
            end
          end
        end
      end
```

Exit the class folder and return to the previous working folder.

4. Create and save the following TFL table definition file, `tfl_table_custom_add_ufix32.m`. This file defines a TFL table containing a single operator entry, a TFL entry generator for unsigned 32-bit fixed-point addition operations, with arbitrary fraction-length values on the inputs and the output. This entry instantiates the derived class from the previous step, `TflCustomOperationEntry`.

Note

- If you want to replace all word sizes and signedness attributes (not just 32-bit and unsigned), you can use the same derived class, but not the same TFL entry, because the `WordLength` and `IsSigned` arguments cannot be wildcarded. For example, to support `uint8`, `int8`, `uint16`, `int16`, and `int32`, you would need to add five other distinct TFL entries. Similarly, if you wanted to use different implementation functions for saturation and rounding modes other than overflow and round to floor, you would need to add TFL entries for those match permutations.
- This table entry creates and adds three implementation arguments to hold the fraction-length values for the inputs and output. Alternatively, this table entry could omit those argument definitions and instead the `do_match` method of the derived class `TflCustomOperationEntry` could create and add the three implementation arguments. In particular, you should use the alternative approach when the number of additional implementation arguments required might vary based on compile-time information.

```
function hTable = tfl_table_custom_add_ufix32

hTable = RTW.TflTable;

%% Add TflCustomOperationEntry
op_entry = TflCustomOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                      'RTW_OP_ADD', ...
                'Priority',                 30, ...
                'SaturationMode',           'RTW_SATURATE_ON_OVERFLOW', ...
                'RoundingMode',             'RTW_ROUND_FLOOR', ...
                'ImplementationName',       'myFixptAdd', ...
                'ImplementationHeaderFile', 'myFixptAdd.h', ...
                'ImplementationSourceFile', 'myFixptAdd.c');

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                          'Name',       'y1', ...
                          'IOType',     'RTW_IO_OUTPUT', ...
                          'CheckSlope', false, ...
```

```
                                  'CheckBias',  false, ...
                                  'DataType',   'Fixed', ...
                                  'Scaling',    'BinaryPoint', ...
                                  'IsSigned',   false, ...
                                  'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                                  'Name',       'u1', ...
                                  'IOType',     'RTW_IO_INPUT', ...
                                  'CheckSlope', false, ...
                                  'CheckBias',  false, ...
                                  'DataType',   'Fixed', ...
                                  'Scaling',    'BinaryPoint', ...
                                  'IsSigned',   false, ...
                                  'WordLength', 32);

createAndAddConceptualArg(op_entry, 'RTW.TflArgNumeric', ...
                                  'Name',       'u2', ...
                                  'IOType',     'RTW_IO_INPUT', ...
                                  'CheckSlope', false, ...
                                  'CheckBias',  false, ...
                                  'DataType',   'Fixed', ...
                                  'Scaling',    'BinaryPoint', ...
                                  'IsSigned',   false, ...
                                  'WordLength', 32);

% Specify replacement function signature
createAndSetCImplementationReturn(op_entry, 'RTW.TflArgNumeric', ...
                                       'Name',       'y1', ...
                                       'IOType',     'RTW_IO_OUTPUT', ...
                                       'IsSigned',   false, ...
                                       'WordLength', 32, ...
                                       'FractionLength', 0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
                                       'Name',       'u1', ...
                                       'IOType',     'RTW_IO_INPUT', ...
                                       'IsSigned',   false, ...
                                       'WordLength', 32, ...
                                       'FractionLength', 0);
```

```
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumeric', ...
                              'Name',       'u2', ...
                              'IOType',     'RTW_IO_INPUT', ...
                              'IsSigned',   false, ...
                              'WordLength', 32, ...
                              'FractionLength', 0);

% Add 3 fraction-length args. Actual values will be set during code generation.
createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
                              'Name',       'fl_in1', ...
                              'IOType',     'RTW_IO_INPUT', ...
                              'IsSigned',   false, ...
                              'WordLength', 32, ...
                              'FractionLength', 0, ...
                              'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
                              'Name',       'fl_in2', ...
                              'IOType',     'RTW_IO_INPUT', ...
                              'IsSigned',   false, ...
                              'WordLength', 32, ...
                              'FractionLength', 0, ...
                              'Value',       0);

createAndAddImplementationArg(op_entry, 'RTW.TflArgNumericConstant', ...
                              'Name',       'fl_out', ...
                              'IOType',     'RTW_IO_INPUT', ...
                              'IsSigned',   false, ...
                              'WordLength', 32, ...
                              'FractionLength', 0, ...
                              'Value',       0);

addEntry(hTable, op_entry);
```

5 Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (>> `tbl = tfl_table_custom_add_ufix32`) and by viewing it in the TFL Viewer (>> `RTW.viewTfl(tfl_table_custom_add_ufix32)`). For more information

about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

6 Create and save the following TFL registration file, which references the `tfl_table_custom_add_ufix32` table.

The file specifies that the TFL to be registered is named `'Custom TFL Operator Entry Example'` and consists of `tfl_table_custom_add_ufix32`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_custom_add_ufix32
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Custom TFL Operator Entry Example';
  thisTfl.Description = 'Demonstration of custom match for operator replacement';
  thisTfl.TableList = {'tfl_table_custom_add_ufix32'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see “Registering Target Function Libraries” on page 31-148.

7. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open the model you created in step 1 and navigate to the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

   Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Custom TFL Operator Entry Example`.

8. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

9. Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click either Add block and select **Code Generation > Navigate to Code**. This selection highlights the Sum block code within the model step function in the *`model`*`.c` file. As shown below, the default implementation code for the unsigned 32-bit fixed-point addition operation has been replaced with `myFixptAdd`, and the three additional fraction-length arguments are present.

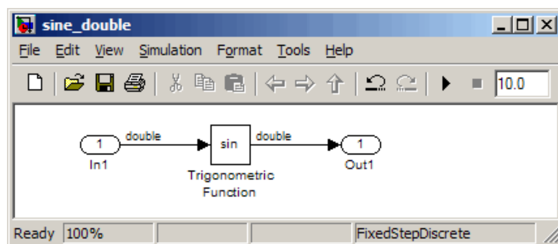
```
/* Model step function */
void ufix32_add_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Inport: '<Root>/In2'
   *  Sum: '<Root>/Add'
```

```
     */
    ufix32_add_Y.Out1 = myFixptAdd(ufix32_add_U.In1, ufix32_add_U.In2, 9U, 7U, 6U);

    /* Outport: '<Root>/Out2' incorporates:
     *  Inport: '<Root>/In3'
     *  Inport: '<Root>/In4'
     *  Sum: '<Root>/Add1'
     */
    ufix32_add_Y.Out2 = myFixptAdd(ufix32_add_U.In3, ufix32_add_U.In4, 10U, 9U, 7U);
  }
```

Example: Customizing TFL Matching and Replacement for Functions

This example demonstrates how to use custom TFL table entries to refine the matching and replacement logic for functions. In this example, a sine function replacement needs to be modified, only if the integer size on the current target platform is 32 bits, such that the implementation function passes in a degrees-versus-radians flag as an input argument.

1 To exercise the custom TFL table entries created in this example, create an ERT-based model with a sine function block, such as the following:



For the purposes of this example, in the import block parameters, set the signal **Data type** to `double`. Also, if the target platform selected for your model on the **Hardware Implementation** pane of the Configuration Parameters dialog box supports an integer size other than 32, you should either temporarily select a target platform with a 32-bit integer size, or modify the code in this example to match the integer size of your target platform.

2 Create a class folder using the name of your derived class, such as `@TflCustomFunctionEntry`. Make sure the class folder is in the MATLAB search path or in the current working folder.

3 In the class folder, create and save the following class definition file, `TflCustomFunctionEntry.m`. This file defines the class `TflCustomFunctionEntry`, which is derived from the base class `RTW.TflCFunctionEntryML`

   The derived class defines a `do_match` method . In the `do_match` method signature,

   - `ent` is the return handle, which is returned either as empty (indicating that the match failed) or as a `TflCFunctionEntry` handle.
   - `hThis` is the handle to this object.
   - `hCSO` is a handle to an object created by the code generator for the purpose of querying the TFL for a replacement.
   - The remaining arguments are the number of bits for various data types of the current target.

   The purpose of the `do_match` method is to add any additional match criteria not provided by the base class and make any desired modifications to the implementation signature. In this case, the `do_match` method only needs to match `targetBitPerInt`, representing the number of bits in the C `int` data type for the current target, to the value 32. If a match is made, the method sets the return handle and creates and adds an input argument, representing whether units are expressed as degrees or radians, to the replacement function signature.

   Note Alternatively, the additional implementation function argument for passing a units flag could be created and added in each TFL table definition file that instantiates this class. In that case, this class definition code would not create the argument and would only set its value. For an example of creating and adding additional implementation function arguments in a table definition file, see "Example: Customizing TFL Matching and Replacement for Operators" on page 31-117.

```
classdef TflCustomFunctionEntry < RTW.TflCFunctionEntryML
  methods
    function ent = do_match(hThis, ...
        hCSO, ... %#ok
        targetBitPerChar, ... %#ok
        targetBitPerShort, ... %#ok
        targetBitPerInt, ... %#ok
        targetBitPerLong) %#ok
      % DO_MATCH - Create a custom match function. The base class
      % checks the types of the arguments prior to calling this
      % method. This will check additional data and perhaps modify
      % the implementation function.

      ent = []; % default the return to empty, indicating the match failed.

      % Match sine function only if the target int size is 32 bits
      if targetBitPerInt == 32
        % Need to modify the default implementation, starting from a copy
        % of the standard TflCFunctionEntry.
        ent = RTW.TflCFunctionEntry(hThis);

        % If the target int size is 32 bits, the implementation function
        % takes an additional input flag argument indicating degress vs.
        % radians. The additional argument can be created and added either
        % in the TFL table definition file that instantiates this class, or
        % here in the class definition, as follows:
        createAndAddImplementationArg(ent, 'RTW.TflArgNumericConstant', ...
                                      'Name',           'u2', ...
                                      'IsSigned',       true, ...
                                      'WordLength',     32, ...
                                      'FractionLength', 0, ...
                                      'Value',          1);
      end
    end
  end
end
```

Exit the class folder and return to the previous working folder.

4 Create and save the following TFL table definition file, `tfl_table_custom_sinfcn_double.m`. This file defines a TFL table containing a function table entry for sine with `double` input and output. This entry instantiates the derived class from the previous step, `TflCustomFunctionEntry`.

```
function hTable = tfl_table_custom_sinfcn_double

hTable = RTW.TflTable;

%% Add TflCustomFunctionEntry
fcn_entry = TflCustomFunctionEntry;
setTflCFunctionEntryParameters(fcn_entry, ...
                'Key',                      'sin', ...
                'Priority',                 30, ...
                'ImplementationName',       'mySin', ...
                'ImplementationHeaderFile', 'mySin.h', ...
                'ImplementationSourceFile', 'mySin.c');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'y1', ...
                          'IOType',       'RTW_IO_OUTPUT', ...
                          'DataTypeMode', 'double');

createAndAddConceptualArg(fcn_entry, 'RTW.TflArgNumeric', ...
                          'Name',         'u1', ...
                          'IOType',       'RTW_IO_INPUT', ...
                          'DataTypeMode', 'double');

% TflCustomFunctionEntry class do_match method will create and add
% an implementation function argument during code generation if
% the supported integer size on the current target is 32 bits.
copyConceptualArgsToImplementation(fcn_entry);

addEntry(hTable, fcn_entry);
```

5 Optionally, perform a quick check of the validity of the function entry by invoking the table definition file at the MATLAB command line (>> `tbl = tfl_table_custom_sinfcn_double`) and by viewing it in the TFL Viewer (>> `RTW.viewTfl(tfl_table_custom_sinfcn_double)`). For more

information about validating TFL tables, see "Examining and Validating Function Replacement Tables" on page 31-139.

6 Create and save the following TFL registration file, which references the `tfl_table_custom_sinfcn_double` table.

The file specifies that the TFL to be registered is named `'Custom TFL Function Entry Example'` and consists of `tfl_table_custom_sinfcn_double`, with the default ANSI math library as the base TFL table.

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_custom_sinfcn_double
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Custom TFL Function Entry Example';
  thisTfl.Description = 'Demonstration of custom match for function replacement';
  thisTfl.TableList = {'tfl_table_custom_sinfcn_double'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Place this `sl_customization.m` file in the MATLAB search path or in the current working folder, so that the TFL is registered at each Simulink startup.

Tip To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`. (To refresh MATLAB Coder TFL registration information within a MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.)

For more information about registering TFLs with Simulink or MATLAB Coder software, see "Registering Target Function Libraries" on page 31-148.

7. With your `sl_customization.m` file in the MATLAB search path or in the current working folder, open the model you created in step 1 and navigate to the **Code Generation > Interface** pane of the Configuration Parameters dialog box. Verify that the **Target function library** option lists the TFL name you specified and select it.

   Optionally, you can relaunch the TFL Viewer, using the MATLAB command `RTW.viewTFL` with no argument, to examine all registered TFLs, including `Custom TFL Function Entry Example`.

8. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**. Then go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

9. Go to the model window and use model-to-code highlighting to trace the code generated using your TFL entry. For example, right-click the sine block and select **Code Generation > Navigate to Code**. This selection highlights the sine block code within the model step function in the *`model`*`.c` file. As shown below, the default implementation code for the sine function has been replaced with `mySin`, and the additional units flag input argument is present.

```
/* Model step function */
void sine_double_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Trigonometry: '<Root>/Trigonometric Function'
   */
  sine_double_Y.Out1 = mySin(sine_double_U.In1, 1);
```

```
}
```

Note Optionally, you can change the current target for the model such that the supported integer size is not 32 bits, and then regenerate code. In that case, the custom match is not made, and the additional input argument does not appear in the generated code for the sine block.

Replacing Math Functions Based on Computation Method

Certain math function blocks are configured with computation or approximation methods that you can use as distinguishing attributes to control TFL-based function replacement. For example,

- You can configure the rSqrt block to use either of two computation methods, `Newton-Raphson` or `Exact`.
- You can configure the Trigonometric Function block, with **Function** set to `sin` or `cos`, to use either of two approximation methods, `CORDIC` or `None`.

You can define TFL table entries to replace these functions for one or all of the available computation methods. For example, you could provide a table entry to replace only `Newton-Raphson` instances of the `rSqrt` function.

To distinguish between computation methods for a given function, use the `EntryInfoAlgorithm` property of TFL function entries in a call to the `setTflCFunctionEntryParameters` function. The arguments for specifying the computation method to match during code generation are:

- For rSqrt:
  - `'RTW_DEFAULT'` (match the default computation method, `Exact`)
  - `'RTW_NEWTON_RAPHSON'`
  - `'RTW_UNSPECIFIED'` (match any computation method)
- For Sine or Cosine:
  - `'RTW_CORDIC'`
  - `'RTW_DEFAULT'` (match the default approximation method, `None`)

- 'RTW_UNSPECIFIED' (match any approximation method)

For example, to replace only `Newton-Raphson` instances of the `rSqrt` function, you can create a table entry similar to the following:

```
hLib = RTW.TflTable;

%
% real_T rsqrt(real_T)
%

e = RTW.TflCFunctionEntry;
setTflCFunctionEntryParameters(e, ...
  'Key', 'rSqrt', ...
  'Priority', 80, ...
  'ImplementationName', 'rsqrt_newton', ...
  'ImplementationHeaderFile', 'rsqrt.h', ...
   EntryInfoAlgorithm ,  RTW_NEWTON_RAPHSON );
createAndAddConceptualArg(e, 'RTW.TflArgNumeric', ...
  'Name', 'y1', ...
  'IOType', 'RTW_IO_OUTPUT', ...
  'DataTypeMode', 'double');
createAndAddConceptualArg(e, 'RTW.TflArgNumeric', ...
  'Name', 'u1', ...
  'DataTypeMode', 'double');
copyConceptualArgsToImplementation(e);
addEntry(hLib, e);
```

The generated code for a `Newton-Raphson` instance of the `rSqrt` function resembles the following:

```
/* Model step function */
void mrsqrt_step(void)
{
  /* Outport: '<Root>/Out1' incorporates:
   *  Inport: '<Root>/In1'
   *  Sqrt: '<Root>/rSqrtBlk'
   */
  mrsqrt_Y.Out1 = rsqrt_newton(mrsqrt_U.In1);
}
```

Specifying Build Information for Function Replacements

- "Functions for Specifying Table Entry Build Information" on page 31-134
- "Using RTW.copyFileToBuildDir to Copy Files to the Build Folder" on page 31-135
- "RTW.copyFileToBuildDir Examples" on page 31-135

Functions for Specifying Table Entry Build Information

As you create TFL table entries for function or operator replacement, you specify the header and source file information for each function implementation using one of the following:

- The arguments `ImplementationHeaderFile`, `ImplementationHeaderPath`, `ImplementationSourceFile`, and `ImplementationSourcePath` to `setTflCFunctionEntryParameters` or `setTflCOperationEntryParameters`
- The `headerFile` argument to `registerCFunctionEntry`, `registerCPPFunctionEntry`, or `registerCPromotableMacroEntry`

Also, each table entry can specify additional header files, source files, and object files to be included in model builds whenever the TFL table entry is matched and used to replace a function or operator in generated code. To add an additional header file, source file, or object file, use the following TFL table creation functions.

Function	Description
`addAdditionalHeaderFile`	Add additional header file to array of additional header files for TFL table entry
`addAdditionalIncludePath`	Add additional include path to array of additional include paths for TFL table entry
`addAdditionalLinkObj`	Add additional link object to array of additional link objects for TFL table entry

Function	Description
addAdditionalLinkObjPath	Add additional link object path to array of additional link object paths for TFL table entry
addAdditionalSourceFile	Add additional source file to array of additional source files for TFL table entry
addAdditionalSourcePath	Add additional source path to array of additional source paths for TFL table entry

Using RTW.copyFileToBuildDir to Copy Files to the Build Folder

If a TFL table entry uses header, source, or object files that reside in external directories, and if the table entry is matched and used to replace a function or operator in generated code, the external files will need to be copied to the build folder before the generated code is built. The `RTW.copyFileToBuildDir` function can be invoked after code generation to copy the table entry's specified header file, source file, additional header files, additional source files, and additional link objects to the build folder. The copied files are then available for use in the build process.

To direct that a table entry's external files should be copied to the build folder after code generation, specify the argument `'RTW.copyFileToBuildDir'` to the `genCallback` parameter of the TFL function that you use to set the table entry parameters, among the following:

- `registerCFunctionEntry`
- `registerCPPFunctionEntry`
- `registerCPromotableMacroEntry`
- `setTflCFunctionEntryParameters`
- `setTflCOperationEntryParameters`

RTW.copyFileToBuildDir Examples

The following example defines a table entry for an optimized multiplication function that takes signed 32-bit integers and returns a signed 32-bit integer, taking saturation into account. Multiplications in the generated code will

be replaced with calls to your optimized function. Your optimized function resides in an external folder and must be copied into the build folder to be compiled and linked into the application.

The multiplication table entry specifies the source and header file names as well as their full paths. To request the copy to be performed, the table entry specifies the argument `'RTW.copyFileToBuildDir'` to the `genCallback` parameter of the `setTflCOperationEntryParameters` function. In this example, the header file `s32_mul.h` contains an inlined function that invokes assembly functions contained in `s32_mul.s`. If the table entry is matched and used to generate code, the `RTW.copyFileToBuildDir` function will copy the specified source and header files into the build folder.

```
function hTable = make_my_tfl_table

hTable = RTW.TflTable;

op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
                'Key',                        'RTW_OP_MUL', ...
                'Priority',                   100, ...
                'SaturationMode',             'RTW_SATURATE_ON_OVERFLOW', ...
                'RoundingMode',               'RTW_ROUND_UNSPECIFIED', ...
                'ImplementationName',         's32_mul_s32_s32_sat', ...
                'ImplementationHeaderFile', 's32_mul.h', ...
                'ImplementationSourceFile', 's32_mul.s', ...
                'ImplementationHeaderPath', {fullfile('$(MATLAB_ROOT)','tfl')}, ...
                'ImplementationSourcePath', {fullfile('$(MATLAB_ROOT)','tfl')}, ...
                'GenCallback',                'RTW.copyFileToBuildDir');
.
.
.
addEntry(hTable, op_entry);
```

The following example shows the use of the `addAdditional*` functions along with `RTW.copyFileToBuildDir`.

```
hTable = RTW.TflTable;

% Path to external source, header, and object files
libdir = fullfile('$(MATLAB_ROOT)','..', '..', 'lib');
```

```
op_entry = RTW.TflCOperationEntry;
setTflCOperationEntryParameters(op_entry, ...
               'Key',                      'RTW_OP_ADD', ...
               'Priority',                 90, ...
               'SaturationMode',           'RTW_SATURATE_UNSPECIFIED', ...
               'RoundingMode',             'RTW_ROUND_UNSPECIFIED', ...
               'ImplementationName',       's32_add_s32_s32', ...
               'ImplementationHeaderFile', 's32_add_s32_s32.h', ...
               'ImplementationSourceFile', 's32_add_s32_s32.c'...
               'GenCallback',              'RTW.copyFileToBuildDir');

addAdditionalHeaderFile(op_entry, 'all_additions.h');
addAdditionalIncludePath(op_entry, fullfile(libdir, 'include'));
addAdditionalSourceFile(op_entry, 'all_additions.c');
addAdditionalSourcePath(op_entry, fullfile(libdir, 'src'));
addAdditionalLinkObj(op_entry, 'addition.o');
addAdditionalLinkObjPath(op_entry, fullfile(libdir, 'bin'));
.
.
.
addEntry(hTable, op_entry);
```

Adding Target Function Library Reserved Identifiers

The Simulink Coder software reserves certain words for its own use as keywords of the generated code language. Reserved keywords for code generation are for use internal to the Simulink Coder software or C programming and should not be used in Simulink models as identifiers or function names. Reserved keywords for code generation include many TFL identifiers, the majority of which are function names, such as `acos`. To view a list of reserved identifiers for the TFL that you are using to generate code, call the MATLAB function `RTW.TargetRegistry.getInstance.getTflReservedIdentifiers`, passing the name of the TFL as displayed in the **Target function library** menu on the **Interface** pane of the Configuration Parameters dialog box. For example,

```
tfl_ids = RTW.TargetRegistry.getInstance.getTflReservedIdentifiers('GNU99 (GNU)')
```

For more information, see “Simulink Coder Target Function Library Keywords” in the Simulink Coder documentation.

In a TFL table, each function implementation name defined by a table entry is registered as a reserved identifier. You can register additional reserved identifiers for the table on a per-header-file basis. Providing additional reserved identifiers can help prevent duplicate symbols and other identifier-related compile and link issues.

To register additional TFL reserved identifiers, use the following function.

Function	Description
`setReservedIdentifiers`	Register specified reserved identifiers to be associated with TFL table

You can register up to four reserved identifier structures in a TFL table. One set of reserved identifiers can be associated with an arbitrary TFL, while the other three (if present) must be associated with ANSI, ISO[16], or GNU[17] libraries. The following example shows a reserved identifier structure that specifies two identifiers and the associated header file.

```
d{1}.LibraryName = 'ANSI';
d{1}.HeaderInfos{1}.HeaderName = 'math.h';
d{1}.HeaderInfos{1}.ReservedIds = {'y0', 'y1'};
```

The specified identifiers are added to the reserved identifiers collection and honored during the build procedure. For more information and examples, see `setReservedIdentifiers`.

16. ISO® is a registered trademark of the International Organization for Standardization.

17. GNU® is a registered trademark of the Free Software Foundation.

Examining and Validating Function Replacement Tables

In this section...

"Overview of Function Replacement Table Validation" on page 31-139

"Invoking the Table Definition File" on page 31-139

"Using the Target Function Library Viewer to Examine Your Table" on page 31-140

"Using the Target Function Library Viewer to Examine Registered TFLs" on page 31-141

"Tracing Code Generated Using Your Target Function Library" on page 31-143

"Examining TFL Cache Hits and Misses" on page 31-144

Overview of Function Replacement Table Validation

After you create a target function library (TFL) table containing your function replacement entries, but before you deploy production TFLs containing your table for general use in building models, you can use various techniques to examine and validate the TFL table entries. These include:

- Invoking the table definition file
- Using the TFL Viewer at various stages of TFL development to examine TFLs, tables, and entries
- Tracing code generated from models for which your TFL is selected
- Examining TFL cache hits and misses logged during code generation

Invoking the Table Definition File

Immediately after creating or modifying a table definition file (as described in "Creating Function Replacement Tables" on page 31-16), you should invoke it at the MATLAB command line. This invocation serves as a check of the validity of your table entries. For example,

```
>> tbl = tfl_table_sinfcn

tbl =
```

```
RTW.TflTable
                Version: '1.0'
             AllEntries: [2x1 RTW.TflCFunctionEntry]
        ReservedSymbols: []
    StringResolutionMap: []
>>
```

Any errors found during the invocation are displayed. In the following example, a typo in a data type name is detected and displayed.

```
>> tbl = tfl_table_sinfcn
??? RTW_CORE:tfl:TflTable: Unsupported data type, 'dooble'.

Error in ==> tfl_table_sinfcn at 7
hTable.registerCFunctionEntry(100, 1, 'sin', 'dooble', 'sin_dbl', ...

>>
```

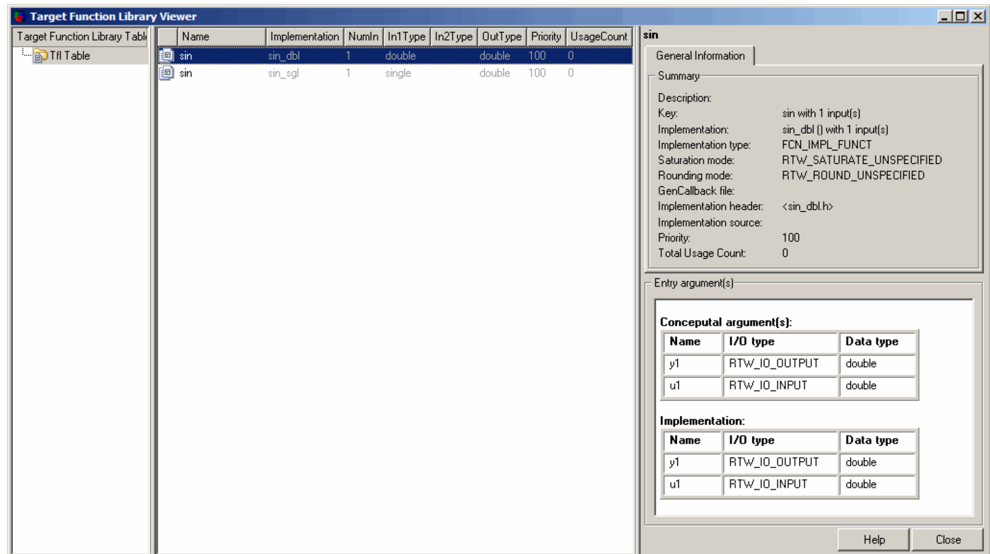
Using the Target Function Library Viewer to Examine Your Table

After creating or modifying a table definition file, as a further check of your table entries, you should use the TFL Viewer to display and examine your table. Invoke the TFL Viewer using the following form of the MATLAB command `RTW.viewTfl`:

```
RTW.viewTfl(table-name)
```

For example,

```
>> RTW.viewTfl(tfl_table_sinfcn)
```



Select entries in your table and verify that the graphical display of the contents of your table meets your expectations. Common problems that can be detected at this stage include:

- Incorrect argument order
- Conceptual argument naming that does not match the naming convention used by the code generation process
- Incorrect relative priority of entries within the table (highest priority is 0, and lowest priority is 100).

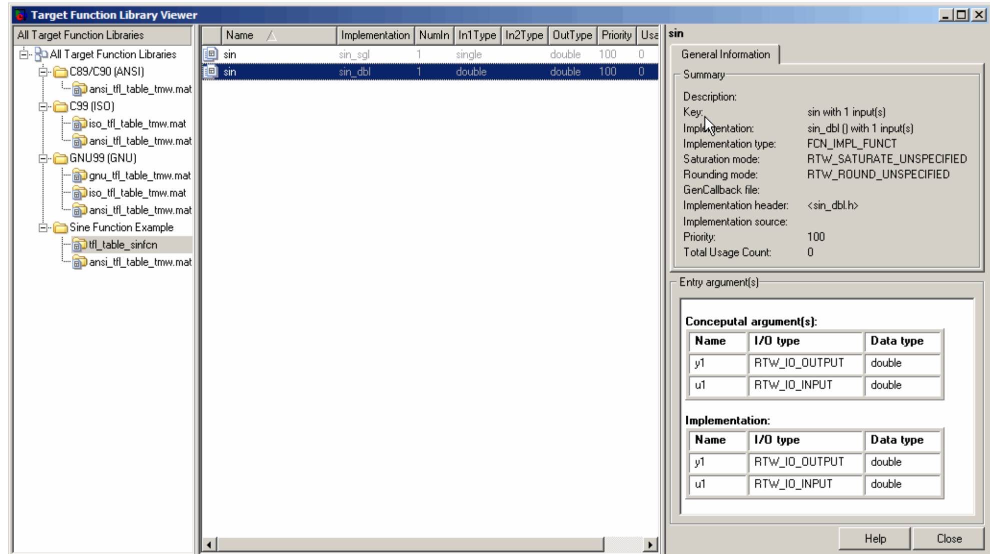
For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Simulink Coder documentation.

Using the Target Function Library Viewer to Examine Registered TFLs

After you register a TFL that includes your function replacement table (as described in “Registering Target Function Libraries” on page 31-148), you should use the TFL Viewer to verify that your TFL was properly registered and to examine the TFL and the tables it contains. Invoke the TFL Viewer

using the MATLAB command `RTW.viewTfl` with no arguments. This command displays all TFLs registered in the current Simulink session. For example:

```
>> RTW.viewTfl
```



If your TFL is not displayed,

- There may be an error in your TFL registration file.
- You may need to refresh the TFL registration information by issuing the MATLAB command `sl_refresh_customizations` or, for a MATLAB Coder TFL registration, using the command `RTW.TargetRegistry.getInstance('reset')`.

If your TFL is displayed, select the TFL and examine and compare its tables, including their relative order. Common problems that can be detected at this stage include

- Incorrect relative order of tables in the library (tables are displayed in search order)
- Table entry problems as listed in the previous section

For more information about the TFL Viewer, see “Using the Target Function Library Viewer” in the Simulink Coder documentation.

Tracing Code Generated Using Your Target Function Library

After you register a TFL that includes your function replacement tables, you should use the TFL to generate code and verify that you are obtaining the function or operator replacement that you expect. For example, the following approach uses model-to-code highlighting to trace a specific expected replacement.

1. Open a ERT-based model for which you anticipate that a function or operator replacement should occur.

2. Select your TFL in the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

3. Go to the **Code Generation > Report** pane of the Configuration Parameters dialog box and select the options **Create code generation report** and **Model-to-code**.

4. Go to the **Code Generation** pane, select the **Generate code only** option, and generate code for the model.

5. Go to the model window and use model-to-code highlighting to trace the code generated using your TFL. For example, right-click a block that you expect to have generated a function or operator replacement and select **Code Generation > Navigate to Code**. This selection highlights the applicable generated function code within `sinefcn.c` as shown in HTML code generation report.

```
21
22  /* Real-time model */
23  RT_MODEL_sinefcn sinefcn_M_;
24  RT_MODEL_sinefcn *sinefcn_M = &sinefcn_M_;
25
26  /* Model step function */
27  void sinefcn_step(void)
28  {
29    /* Outport: '<Root>/Out1' incorporates:
30     *  Inport: '<Root>/In1'
31     *  Trigonometry: '<Root>/Trigonometric Function'
32     */
33    sinefcn_Y.Out1 = sin_dbl(sinefcn_U.In1);
34  }
35
36  /* Model initialize function */
37  void sinefcn_initialize(void)
38  {
39    /* Registration code */
```

Inspect the generated code and see if the function or operator replacement occurred as you expected.

Note If a function or operator was not replaced as you expected, it means that a call site request was not matched as you intended by your table entry attributes. Either a higher-priority (lower priority value) match was used or no match was found. You can analyze the TFL table entry matching behavior by using the following resources together:

- TFL Viewer, as described in "Using the Target Function Library Viewer to Examine Your Table" on page 31-140 and "Using the Target Function Library Viewer to Examine Registered TFLs" on page 31-141
- HTML code generation reports, with bidirectional tracing including model-to-code highlighting
- Statistics for TFL cache hits and misses logged during code generation, as described in "Examining TFL Cache Hits and Misses" on page 31-144

Examining TFL Cache Hits and Misses

Target function library (TFL) replacement may behave differently than you expect in some cases. To verify that you are obtaining the function or

operator replacement that you expect, you first inspect the generated code, as described in "Tracing Code Generated Using Your Target Function Library" on page 31-143.

To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, you can view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types and attributes should be registered in order to achieve the desired replacement.

To display the TFL cache hits and misses logged during the most recent code generation session, you specify the model parameter `TargetFcnLibHandle` in a `get_param` call, as follows:

```
>> tfl=get_param('model', 'TargetFcnLibHandle')
```

The resulting display includes the following fields:

Field	Description
`HitCache`	Table containing function entries that were successfully matched during a code generation session. These entries represent function implementations that should appear in the generated code.
`MissCache`	Table containing function entries that failed to match during a code generation session. These entries are created by the code generation process for the purpose of querying the TFL to locate a registered implementation. If there is a registered implementation that you feel should have been used in the generated code and was not, examining the `MissCache` for entries that are similar but did not match can help you locate discrepancies in a conceptual argument list or in table entry attributes.

Note You also can view cache hits and misses in the TFL Viewer, using the TFL handle returned by the `get_param` call. For example:

```
>> tfl=get_param('model', 'TargetFcnLibHandle')
>> RTW.viewTfl(tfl)
```

This opens the TFL viewer. You can then examine the cache hits and misses by clicking on the entries under those caches.

In the following example, the most recent code generation session logged one cache hit and zero cache misses. You can examine the logged `HitCache` entry using its table index.

```
>> a=get_param( sinefcn , TargetFcnLibHandle )

a =

RTW.TflControl
        Version: '1.0'
       HitCache: [1x1 RTW.TflCFunctionEntry]
      MissCache: [0x1 handle]
    TLCCallList: [0x1 handle]
      TflTables: [2x1 RTW.TflTable]

>> a.HitCache(1)

ans =

RTW.TflCFunctionEntry
                        Key: 'sin'
                   Priority: 100
             ConceptualArgs: [2x1 RTW.TflArgNumeric]
             Implementation: [1x1 RTW.CImplementation]
          RTWmakecfgLibName: ''
                GenCallback: ''
                GenFileName: ''
             SaturationMode: 'RTW_SATURATE_UNSPECIFIED'
               RoundingMode: 'RTW_ROUND_UNSPECIFIED'
            AcceptExprInput: 1
```

```
               SideEffects: 0
                UsageCount: 2
          SharedUsageCount: 0
               Description: ''
                  ImplType: 'FCN_IMPL_FUNCT'
     AdditionalHeaderFiles: {0x1 cell}
    AdditionalIncludePaths: {0x1 cell}
     AdditionalSourceFiles: {0x1 cell}
     AdditionalSourcePaths: {0x1 cell}
        AdditionalLinkObjs: {0x1 cell}
   AdditionalLinkObjsPaths: {0x1 cell}

>>
```

Registering Target Function Libraries

In this section...

"Overview of TFL Registration" on page 31-148

"Using the sl_customization API to Register a TFL with Simulink Software" on page 31-149

"Using the rtwTargetInfo API to Register a TFL with MATLAB® Coder Software" on page 31-153

"Registering Multiple TFLs" on page 31-154

Overview of TFL Registration

After you define function and operator replacements in a target function library (TFL) table definition file, your table can be included in a TFL that you register either with Simulink software or with MATLAB Coder software. When a TFL is registered, it appears in the **Target function library** drop-down list on the **Interface** pane of either the Simulink Configuration Parameters dialog box or the MATLAB Coder Project Settings dialog box. You can select it from the **Target function library** drop-down list for use in code generation.

To register TFLs with Simulink software, use the Simulink customization file `sl_customization.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard Simulink user interface. The Simulink software reads the `sl_customization.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the Simulink session. For more information on the `sl_customization.m` customization file, see "Customizing the Simulink User Interface" in the Simulink documentation.

To register TFLs with MATLAB Coder software, use the MATLAB Coder customization file `rtwTargetInfo.m`. This file is a mechanism that allows you to use MATLAB code to perform customizations of the standard MATLAB Coder project settings. The MATLAB Coder software reads the `rtwTargetInfo.m` file, if present on the MATLAB path, when it starts and the customizations specified in the file are applied to the MATLAB Coder session.

Using the sl_customization API to Register a TFL with Simulink Software

To register a TFL, you create an instance of `sl_customization.m` and include it on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function accepts one argument: a handle to a customization manager object. The function is declared as follows:

```
function sl_customization(cm)
```

The body of the `sl_customization` function invokes the `registerTargetInfo(tfl)` method to register one or more TFLs with the Simulink software. Typically, the `registerTargetInfo` function call references a local function that defines the TFLs to be registered. For example:

```
  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

Below the `sl_customization` function, the referenced local function describes one or more TFLs to be registered. For example, you can declare the local function as follows:

```
% Local function to define a TFL
function thisTfl = locTflRegFcn
```

In the local function body, for each TFL to be registered, you instantiate a TFL registry entry using `tfl = RTW.TflRegistry`. For example,

```
  thisTfl = RTW.TflRegistry;
```

Then, you define the TFL properties shown in the following table within the registry entry.

<table>
<tr><th>TFL Property</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>String specifying the name of the TFL, as it should be displayed in the Target function library drop-down list on the Interface pane of the Configuration Parameters dialog box.</td></tr>
<tr><td><code>Description</code></td><td>String specifying a text description of the TFL, as it should be displayed in the tool tip for the TFL in the Configuration Parameters dialog box.</td></tr>
<tr><td><code>TableList</code></td><td>Cell array of strings specifying the tables that make up the TFL, in descending priority order. Tables can be specified in any of the following ways:

• Name of a TFL table file on the MATLAB search path
• Absolute path to a table file
• Path to a table file relative to <code>$(MATLAB_ROOT)</code>

See “Registering Multiple TFLs” on page 31-154 for examples of each type of table specification.</td></tr>
<tr><td><code>BaseTfl</code></td><td>String specifying the name of the TFL on which this TFL is based.

Note To ensure that functions, macros, and constants used by built-in blocks are available in your TFL, and to help ensure compatibility between releases, you must specify one of the default MathWorks libraries as the base TFL: <code>'C89/C90 (ANSI)'</code>, <code>'C99 (ISO)'</code>, <code>'GNU99 (GNU)'</code>, <code>'C++ (ISO)'</code>, or an equivalent alias.</td></tr>
</table>

TFL Property	Description
`TargetHWDeviceType`	Always specify `{'*'}`.
`LanguageConstraint`	Cell array of strings specifying language constraint keywords. You must specify `{'C++'}` if your TFL includes C++ function entries or a mix of C and C++ function entries. Otherwise you can omit the field or specify it as empty.

For example:

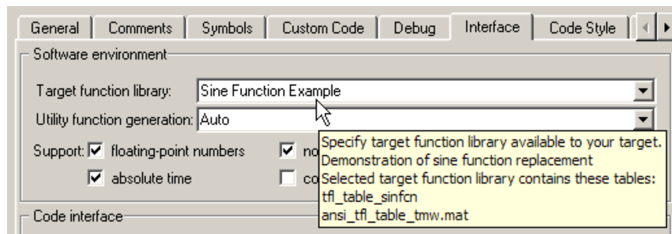
```
    thisTfl.Name = 'Sine Function Example';
    thisTfl.Description = 'Demonstration of sine function replacement';
    thisTfl.TableList = {'tfl_table_sinfcn'};
    thisTfl.BaseTfl = 'C89/C90 (ANSI)';
    thisTfl.TargetHWDeviceType = {'*'};

  end % End of LOCTFLREGFCN
```

Combining the elements described in this section, the complete `sl_customization` function for the `'Sine Function Example'` TFL would appear as follows:

```
function sl_customization(cm)
% sl_customization function to register a target function library (TFL)
% for use with Simulink

  % Register the TFL defined in local function locTflRegFcn
  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION


% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn

  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
```

```
    thisTfl.Name = 'Sine Function Example';
    thisTfl.Description = 'Demonstration of sine function replacement';
    thisTfl.TableList = {'tfl_table_sinfcn'};
    thisTfl.BaseTfl = 'C89/C90 (ANSI)';
    thisTfl.TargetHWDeviceType = {'*'};

  end % End of LOCTFLREGFCN
```

If you place the `sl_customization.m` file containing this function in the MATLAB search path or in the current working folder, the TFL is registered at each Simulink startup. The Simulink software will display the TFL in the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box. For example, the following figure shows the Configuration Parameters dialog box display, including tool tip, for the `'Sine Function Example'` TFL.



Tip

- To refresh Simulink customizations within the current MATLAB session, use the command `sl_refresh_customizations`.
- To list all `sl_customization` files in the current search path, use the command `which sl_customization -all`.
- If you disable a TFL registration (for example, by renaming the registration file `sl_customization.m` and then issuing `sl_refresh_customizations`), you may want to reset and save the **Target function library** option setting in any saved models that selected the disabled TFL.

Using the rtwTargetInfo API to Register a TFL with MATLAB Coder Software

To register a TFL for use with MATLAB Coder software, you create an instance of `rtwTargetInfo.m` and include it on the MATLAB path of the MATLAB Coder installation that you want to customize. The `rtwTargetInfo` function accepts one argument: a handle to a target registration object. The function is declared as follows:

```
function rtwTargetInfo(tr)
```

The body of the `rtwTargetInfo` function invokes the `registerTargetInfo(tfl)` method provided by the target registry object to register one or more TFLs with the MATLAB Coder software. Typically, the `registerTargetInfo` function call references a local function that defines the TFLs to be registered. For example:

```
  % Register the TFL defined in local function locTflRegFcn
  tr.registerTargetInfo(@locTflRegFcn);

end % End of RTWTARGETINFO
```

Below the `rtwTargetInfo` function, the referenced local function describes one or more TFLs to be registered. The format exactly matches the TFL description format previously described for Simulink use. For example, here is the MATLAB Coder equivalent of the complete TFL registration file displayed in "Using the sl_customization API to Register a TFL with Simulink Software" on page 31-149.

```
function rtwTargetInfo(tr)
% rtwTargetInfo function to register a target function library (TFL)
% for use with codegen

  % Register the TFL defined in local function locTflRegFcn
  tr.registerTargetInfo(@locTflRegFcn);

end % End of RTWTARGETINFO


% Local function to define a TFL containing tfl_table_sinfcn
function thisTfl = locTflRegFcn
```

```
  % Instantiate a TFL registry entry
  thisTfl = RTW.TflRegistry;

  % Define the TFL properties
  thisTfl.Name = 'Sine Function Example';
  thisTfl.Description = 'Demonstration of sine function replacement';
  thisTfl.TableList = {'tfl_table_sinfcn'};
  thisTfl.BaseTfl = 'C89/C90 (ANSI)';
  thisTfl.TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

If you place the `rtwTargetInfo.m` file containing this function in the MATLAB search path or in the current working folder, the TFL is registered at each MATLAB Coder startup. The MATLAB Coder software will display the TFL in the **Target function library** drop-down list on the **Code Generation > Interface** pane of the Configuration Parameter dialog box.

Tip To refresh MATLAB Coder TFL registration information within the current MATLAB session, use the command `RTW.TargetRegistry.getInstance('reset');`.

Registering Multiple TFLs

For an example of a TFL registration file that registers multiple TFLs, see the `sl_customization.m` file used in the TFL demo, `rtwdemo_tfl_script`. The following example illustrates the general approach, which applies equally to Simulink and MATLAB Coder TFL registration files. In this example, the three TFL tables referenced in the `TableList` fields reside at different locations, either on the MATLAB search path or at locations specified using path strings.

```
function sl_customization(cm)

  cm.registerTargetInfo(@locTflRegFcn);

end % End of SL_CUSTOMIZATION
```

```
% Local function(s)
function thisTfl = locTflRegFcn
  % Register a Target Function Library for use with model: rtwdemo_tfladdsub.mdl
  thisTfl(1) = RTW.TflRegistry;
  thisTfl(1).Name = 'Addition & Subtraction Examples';
  thisTfl(1).Description = 'Demonstration of addition/subtraction op replacement';
  thisTfl(1).TableList = {'tfl_table_addsub'};
  thisTfl(1).BaseTfl = 'C89/C90 (ANSI)';
  thisTfl(1).TargetHWDeviceType = {'*'};

  % Register a Target Function Library for use with model: rtwdemo_tflmuldiv.mdl
  thisTfl(2) = RTW.TflRegistry;
  thisTfl(2).Name = 'Multiplication & Division Examples';
  thisTfl(2).Description = 'Demonstration of mult/div op repl for built-in integers';
  thisTfl(2).TableList = {'c:/work_tfl/tfl_table_muldiv'};
  thisTfl(2).BaseTfl = 'C89/C90 (ANSI)';
  thisTfl(2).TargetHWDeviceType = {'*'};

  % Register a Target Function Library for use with model: rtwdemo_tflfixpt.mdl
  thisTfl(3) = RTW.TflRegistry;
  thisTfl(3).Name = 'Fixed-Point Examples';
  thisTfl(3).Description = 'Demonstration of fixed-point operator replacement';
  thisTfl(3).TableList = ...
{fullfile('$(MATLAB_ROOT)','toolbox','rtw','rtwdemos','tfl_demo','tfl_table_fixpt')};
  thisTfl(3).BaseTfl = 'C89/C90 (ANSI)';
  thisTfl(3).TargetHWDeviceType = {'*'};

end % End of LOCTFLREGFCN
```

Target Function Library Limitations

- Target function library (TFL) replacement may behave differently than you expect in some cases. For example, data types that you observe in a model do not necessarily match what the code generator determines to use as intermediate data types in an operation. To verify whether you are obtaining the function or operator replacement that you expect, inspect the generated code.
- To analyze replacement behavior, in addition to referencing the generated code and examining your TFL tables in the TFL Viewer, view the TFL cache hits and misses logged during the most recent code generation session. This approach provides information on what data types should be registered in order to achieve the desired replacement. For more information on analyzing TFL table entries, see "Examining and Validating Function Replacement Tables" on page 31-139.
- You must register a TFL in either an `sl_customization.m` file or an `rtwTargetInfo` file, but not in both files.

Setting Up Generated Code To Interface With Components in the Run-Time Environment

- Chapter 32, "Configuring the Target Hardware Environment"
- Chapter 33, "Model Entry Points"
- Chapter 34, "Interfacing With Hardware That is Not Running an Operating System (Bare Board)"
- Chapter 35, "Wind River Systems VxWorks Example Main Program"

32

Configuring the Target Hardware Environment

- “Configuring Support for Numeric Data” on page 32-2
- “Configuring Support for Time Values” on page 32-3
- “Setting Up Support for Non-Inlined S-Functions” on page 32-4
- “Configuring Model Function Generation and Argument Passing” on page 32-5
- “Setting Up Support for Code Reuse” on page 32-7
- “Configuring Target Function Libraries” on page 32-8

Configuring Support for Numeric Data

By default, ERT targets support code generation for integer, floating-point, nonfinite, and complex numbers.

To Generate Code that Supports...	Do...
Integer data only	Deselect **Support floating-point numbers**. If any noninteger data or expressions are encountered during code generation, an error message reports the offending blocks and parameters.
Floating-point data	Select **Support floating-point numbers**.
Nonfinite values (for example, `NaN`, `Inf`)	Select **Support floating-point numbers** and **Support non-finite numbers** .
Complex data	Select **Support complex numbers** .

For more information, see “Code Generation Pane: Interface” in the Simulink Coder reference documentation.

Configuring Support for Time Values

Certain blocks require the value of absolute time (that is, the time from the start of program execution to the present time) , elapsed time (for example, the time elapsed between two trigger events), or continuous time. Depending on the blocks used, you might need to adjust the configuration settings for supported time values.

To...	Select...
Generate code that creates and maintains integer counters for blocks that use absolute or elapsed time values (default)	**Support absolute time**. For further information on the allocation and operation of absolute and elapsed timers, see the "Using Timers" chapter of the Simulink Coder documentation. If you do not select this parameter and the model includes block that use absolute or elapsed time values, the build process generates an error.
Generate code for blocks that rely on continuous time	**Support continuous time**. If you do not select this parameter and the model includes continuous-time blocks, the build process generates an error.

For more information, see "Code Generation Pane: Interface" in the Simulink Coder reference documentation.

Setting Up Support for Non-Inlined S-Functions

To generate code for noninlined S-Functions in a model, select **Support noninlined S-functions**. The generation of noninlined S-functions requires floating-point and nonfinite numbers. Thus, when you select **Support non-inlined S-functions**, the ERT target automatically selects **Support floating-point numbers** and **Support non-finite numbers**.

When you select **Support non-finite numbers**, the build process generates an error if the model includes a C MEX S-function that does not have a corresponding TLC implementation (for inlining code generation).

Note that inlining S-functions is highly advantageous in production code generation, for example in implementing device drivers. To enforce the use of inlined S-functions for code generation, deselect **Support non-inlined S-functions**.

For more information, see "Code Generation Pane: Interface" in the Simulink Coder reference documentation.

Configuring Model Function Generation and Argument Passing

For ERT targets, you can configure how a model's functions are generated and how arguments are passed to the functions.

To...	Do...
Generate model function calls that are compatible with the main program module of the GRT target (`grt_main.c` or `.cpp`)	Select **GRT compatible call interface** and **MAT-file logging** . In addition, deselect **Suppress error status in real-time model data structure**. **GRT compatible call interface** provides a quick way to use ERT target features with a GRT-based custom target by generating wrapper function calls that interface to the ERT target's `Embedded-C` formatted code.
Reduce overhead and use more local variables by combining the output and update functions in a single *`model`*`_step` function	Select **Single output/update function** Errors or unexpected behavior can occur if a Model block is part of a cycle and "Single output/update function" is enabled (the default). See "Model Blocks and Direct Feedthrough" for details.
Generate a *`model`*`_terminate` function for a model not designed to run indefinitely	Select **Terminate function required**. For more information, see the description of `model_terminate`.
Generate reusable, reentrant code from a model or subsystem	Select **Generate reusable code**. See "Setting Up Support for Code Reuse" on page 32-7 for details.
Statically allocate model data structures and access them directly in the model code	Deselect **Generate reusable code**. The generated code is not reusable or reentrant. See Chapter 33, "Model Entry Points" for information on the calling interface generated for model functions in this case.

To...	Do...
Suppress the generation of an error status field in the real-time model data structure, `rtModel`, for example, if you do not need to log or monitor error messages	Select **Suppress error status in real-time model data structure**. Selecting this parameter can also cause the `rtModel` structure to be omitted completely from the generated code. When generating code for multiple integrated models, set this parameter the same for all of the models. Otherwise, the integrated application might exhibit unexpected behavior. For example, if you select the option in one model but not in another, the error status might not be registered by the integrated application. Do not select this parameter if you select the **MAT-file logging** option. The two options are incompatible.
Launch the Model Step Functions dialog box (see "Configuring Model Function Prototypes" on page 29-4) preview and modify the model's *`model`*`_step` function prototype	Click **Configure Step Function**. Based on the **Function specification** value you select for your *`model`*`_step` function (supported values include `Default model-step function` and `Model specific C prototype`), you can preview and modify the function prototype. Once you validate and apply your changes, you can generate code based on your function prototype modifications. For more information about using the **Configure Step Function** button and the Model Step Functions dialog box, see Chapter 29, "Controlling Generation of Function Prototypes".

For more information, see "Code Generation Pane: Interface" in the Simulink Coder reference documentation.

Setting Up Support for Code Reuse

For ERT targets, you can configure how a model reuses code using the **Generate reusable code** parameter.

Pass root-level I/O as provides options that control how model inputs and outputs at the root level of the model are passed to the `model_step` function.

To...	Select...
Pass each root-level model input and output argument to the`model_step` function individually (the default)	**Generate reusable code** and **Pass root-level I/O as** > `Individual arguments`.
Pack root-level input arguments and root-level output arguments into separate structures that are then passed to the `model_step` function	**Generate reusable code** and **Pass root-level I/O as** > `Structure reference`

In some cases, selecting **Generate reusable code** can generate code that compiles but is not reentrant. For example, if any signal, `DWork` structure, or parameter data has a storage class other than `Auto`, global data structures are generated. To handle such cases, use the **Reusable code error diagnostic** parameter to choose the severity levels for diagnostics

In some cases, the Embedded Coder software is unable to generate valid and compilable code. For example, if the model contains any of the following, the code generated would be invalid.

- An S-function that is not code-reuse compliant
- A subsystem triggered by a wide function call trigger

In these cases, the build terminates after reporting the problem.

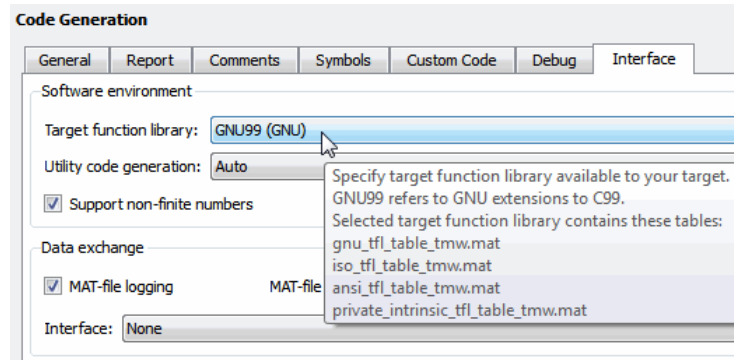
For more information, see “Code Generation Pane: Interface” in the Simulink Coder reference documentation.

Configuring Target Function Libraries

A *target function library* (TFL) is a set of one or more function replacement tables that define the target-specific implementations of math functions and operators to be used in generating code for your Simulink model. The Simulink Coder product provides default TFLs, which you can select from the **Target function library** drop-down list on the **Interface** pane of the Configuration Parameters dialog box.

TFL	Description	Contains tables...
C89/C90 (ANSI)	Generates calls to the ISO/IEC 9899:1990 C standard math library for floating-point functions.	ansi_tfl_table_tmw.mat
C99 (ISO)	Generates calls to the ISO/IEC 9899:1999 C standard math library.	iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat
GNU99 (GNU)	Generates calls to the Free Software Foundation's GNU gcc math library, which provides C99 extensions as defined by compiler option -std=gnu99.	gnu_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat
C++ (ISO)	Generates calls to the ISO/IEC 14882:2003 C++ standard math library.	iso_cpp_tfl_table_tmw.mat private_iso_cpp_tfl_table_tmw.mat iso_tfl_table_tmw.mat ansi_tfl_table_tmw.mat

TFL tables provide the basis for replacing default math functions and operators in your model code with target-specific code. If you select a library and then hover over the selected library with the cursor, a tool tip is displayed that describes the TFL and lists the function replacement tables it contains. Tables are listed in the order in which they are searched for a function or operator match.



The Simulink Coder product allows you to view the content of TFL function replacement tables using the Target Function Library Viewer, as described in "Selecting and Viewing Target Function Libraries". The Embedded Coder product allows you to additionally create and register the function replacement tables that make up a TFL, as described in Chapter 31, "Replacing Math Functions and Operators Using Target Function Libraries".

Model Entry Points

The following functions represent entry points in the generated code for a Simulink model.

Function	Description
`model_initialize`	Initialization entry point in generated code for Simulink model
`model_SetEventsForThisBaseStep`	Set event flags for multirate, multitasking operation before calling *`model`*`_step` for Simulink model — not generated as of Version 5.1 (R2008a)
`model_step`	Step routine entry point in generated code for Simulink model
`model_terminate`	Termination entry point in generated code for Simulink model

Note that the calling interface generated for each of these functions differs significantly depending on how you set the **Generate reusable code** option (see Chapter 32, "Configuring the Target Hardware Environment").

By default, **Generate reusable code** is off, and the model entry point functions access model data with statically allocated global data structures.

When **Generate reusable code** is on, model data structures are passed in (by reference) as arguments to the model entry point functions. For efficiency, only those data structures that are actually used in the model are passed in. Therefore when **Generate reusable code** is on, the argument lists generated for the entry point functions vary according to the requirements of the model.

The entry points are exported with `model.h`. To call the entry-point functions from your hand-written code, add an `#include model.h` directive to your code. If **Generate reusable code** is on, you must examine the generated code to determine the calling interface required for these functions.

For more information, see the reference pages for the listed functions.

Note The function reference pages document the default (**Generate reusable code** off) calling interface generated for these functions.

34

Interfacing With Hardware That is Not Running an Operating System (Bare Board)

- "About Standalone Program Execution" on page 34-2
- "Generating a Standalone Program" on page 34-3
- "Standalone Program Components" on page 34-4
- "Main Program" on page 34-5
- "rt_OneStep and Scheduling Considerations" on page 34-7
- "Static Main Program Module" on page 34-14
- "Rate Grouping Compliance and Compatibility Issues" on page 34-19

About Standalone Program Execution

By default, the Embedded Coder software generates *standalone* programs that do not require an external real-time executive or operating system. A standalone program requires minimal modification to be adapted to the target hardware. The standalone program architecture supports execution of models with either single or multiple sample rates.

Generating a Standalone Program

To generate a standalone program:

1. In the **Custom templates** section of the **Code Generation > Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (is on by default). This enables the **Target operating system** menu.

2. From the **Target operating system** menu, select `BareBoardExample` (the default selection).

3. Generate the code.

Different code is generated for multirate models depending on the following factors:

- Whether the model executes in single-tasking or multitasking mode.
- Whether or not reusable code is being generated.

These factors affect the scheduling algorithms used in generated code, and in some cases affect the API for the model entry point functions. The following sections discuss these variants.

Standalone Program Components

The core of a standalone program is the main loop. On each iteration, the main loop executes a background or null task and checks for a termination condition.

The main loop is periodically interrupted by a timer. The function `rt_OneStep` is either installed as a timer interrupt service routine (ISR), or called from a timer ISR at each clock step.

The execution driver, `rt_OneStep`, sequences calls to the *`model`*`_step` functions. The operation of `rt_OneStep` differs depending on whether the generating model is single-rate or multirate. In a single-rate model, `rt_OneStep` simply calls the *`model`*`_step` function. In a multirate model, `rt_OneStep` prioritizes and schedules execution of blocks according to the rates at which they run.

Main Program

In this section...

"Overview of Operation" on page 34-5

"Guidelines for Modifying the Main Program" on page 34-5

Overview of Operation

The following pseudocode shows the execution of a main program.

```
main()
{
  Initialization (including installation of rt_OneStep as an
    interrupt service routine for a real-time clock)
  Initialize and start timer hardware
  Enable interupts
  While(not Error) and (time < final time)
    Background task
  EndWhile
  Disable interrupts (Disable rt_OneStep from executing)
  Complete any background tasks
  Shutdown
}
```

The pseudocode is a design for a harness program to drive your model. The `ert_main.c` or `.cpp` program only partially implements this design. You must modify it according to your specifications.

Guidelines for Modifying the Main Program

This section describes the minimal modifications you should make in your production version of `ert_main.c` or `.cpp` to implement your harness program.

1. Call *`model`*`_initialize`.

2. Initialize target-specific data structures and hardware, such as ADCs or DACs.

3. Install `rt_OneStep` as a timer ISR.

4 Initialize timer hardware.

5 Enable timer interrupts and start the timer.

Note `rtModel` is not in a valid state until *`model`*`_initialize` has been called. Servicing of timer interrupts should not begin until *`model`*`_initialize` has been called.

---

6 Optionally, insert background task calls in the main loop.

7 On termination of the main loop (if applicable):

- Disable timer interrupts.
- Perform target-specific cleanup such as zeroing DACs.
- Detect and handle errors. Note that even if your program is designed to run indefinitely, you may need to handle severe error conditions, such as timer interrupt overruns.

  You can use the macros `rtmGetErrorStatus` and `rtmSetErrorStatus` to detect and signal errors.

rt_OneStep and Scheduling Considerations

In this section...

"Overview of Operation" on page 34-7

"Single-Rate Single-tasking Operation" on page 34-8

"Multirate Multitasking Operation" on page 34-9

"Multirate Single-Tasking Operation" on page 34-11

"Guidelines for Modifying rt_OneStep" on page 34-12

Overview of Operation

The operation of `rt_OneStep` depends upon

- Whether your model is single-rate or multirate. In a single-rate model, the sample times of all blocks in the model, and the model's fixed step size, are the same. Any model in which the sample times and step size do not meet these conditions is termed multirate.
- Your model's solver mode (`SingleTasking` versus `MultiTasking`)

Permitted Solver Modes for Embedded Coder™ Targeted Models on page 34-7 summarizes the permitted solver modes for single-rate and multirate models. Note that for a single-rate model, only `SingleTasking` solver mode is allowed.

Permitted Solver Modes for Embedded Coder Targeted Models

Mode	Single-Rate	Multirate
`SingleTasking`	Allowed	Allowed
`MultiTasking`	Disallowed	Allowed
`Auto`	Allowed (defaults to `SingleTasking`)	Allowed (defaults to `MultiTasking`)

The generated code for rt_OneStep (and associated timing data structures and support functions) is tailored to the number of rates in the model and to the solver mode. The following sections discuss each possible case.

Single-Rate Single-tasking Operation

The only valid solver mode for a single-rate model is SingleTasking. Such models run in “single-rate” operation.

The following pseudocode shows the design of rt_OneStep in a single-rate program.

```
rt_OneStep()
{
  Check for interrupt overflow or other error
  Enable "rt_OneStep" (timer) interrupt
  Model_Step()  -- Time step combines output,logging,update
}
```

For the single-rate case, the generated *model*_step function is

```
void model_step(void)
```

Single-rate rt_OneStep is designed to execute *model*_step within a single clock period. To enforce this timing constraint, rt_OneStep maintains and checks a timer overrun flag. On entry, timer interrupts are disabled until the overrun flag and other error conditions have been checked. If the overrun flag is clear, rt_OneStep sets the flag, and proceeds with timer interrupts enabled.

The overrun flag is cleared only upon successful return from *model*_step. Therefore, if rt_OneStep is reinterrupted before completing *model*_step, the reinterruption is detected through the overrun flag.

Reinterruption of rt_OneStep by the timer is an error condition. If this condition is detected rt_OneStep signals an error and returns immediately. (Note that you can change this behavior if you want to handle the condition differently.)

Note that the design of rt_OneStep assumes that interrupts are disabled before rt_OneStep is called. rt_OneStep should be noninterruptible until the interrupt overflow flag has been checked.

Multirate Multitasking Operation

In a multirate multitasking system, code generation uses a prioritized, preemptive multitasking scheme to execute the different sample rates in your model.

The following pseudocode shows the design of `rt_OneStep` in a multirate multitasking program.

```
rt_OneStep()
{
  Check for base-rate interrupt overrun
  Enable "rt_OneStep" interrupt
  Determine which rates need to run this time step

  Model_Step0()        -- run base-rate time step code

  For N=1:NumTasks-1   -- iterate over sub-rate tasks
    If (sub-rate task N is scheduled)
    Check for sub-rate interrupt overrun
      Model_StepN()    -- run sub-rate time step code
    EndIf
  EndFor
}
```

Task Identifiers

The execution of blocks having different sample rates is broken into tasks. Each block that executes at a given sample rate is assigned a *task identifier* (`tid`), which associates it with a task that executes at that rate. Where there are `NumTasks` tasks in the system, the range of task identifiers is 0..`NumTasks`-1.

Prioritization of Base-Rate and Subrate Tasks

Tasks are prioritized, in descending order, by rate. The *base-rate* task is the task that runs at the fastest rate in the system (the hardware clock rate). The base-rate task has highest priority (`tid` 0). The next fastest task (`tid` 1) has the next highest priority, and so on down to the slowest, lowest priority task (`tid` `NumTasks`-1).

The slower tasks, running at submultiples of the base rate, are called *subrate* tasks.

Rate Grouping and Rate-Specific model_step Functions

In a single-rate model, all block output computations are performed within a single function, `model_step`. For multirate, multitasking models, code generation uses a different strategy (whenever possible). This strategy is called *rate grouping*. Rate grouping generates separate `model_step` functions for the base rate task and each subrate task in the model. The function naming convention for these functions is

```
model_stepN
```

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

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

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

Scheduling model_stepN Execution

On each clock tick, `rt_OneStep` and `model_step0` maintain scheduling counters and *event flags* for each subrate task. The counters are implemented in the `Timing.TaskCounters.TIDn` fields of `rtModel`. The event flags are implemented as arrays, indexed on `tid`.

The scheduling counters are maintained by the `rate_monotonic_scheduler` function, which is called by `model_step0` (that is, in the base-rate task). The function updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data—and assumes the use of a rate monotonic scheduler. The scheduling counters are, in effect, clock rate dividers that count up the sample period associated with each subrate task.

The event flags indicate whether or not a given task is scheduled for execution. `rt_OneStep` maintains the event flags based on a task counter that

is maintained by code in the model's example main program (`ert_main.c`). When a counter indicates that a task's sample period has elapsed, the example main code sets the event flag for that task.

On each invocation, `rt_OneStep` updates its scheduling data structures and steps the base-rate task (`rt_OneStep` always calls *`model`*`_step0` because the base-rate task must execute on every clock step). Then, `rt_OneStep` iterates over the scheduling flags in `tid` order, unconditionally calling *`model`*`_stepN` for any task whose flag is set. This ensures that tasks are executed in order of priority.

Preemption

Note that the design of `rt_OneStep` assumes that interrupts are disabled before `rt_OneStep` is called. `rt_OneStep` should be noninterruptible until the base-rate interrupt overflow flag has been checked (see pseudocode above).

The event flag array and loop variables used by `rt_OneStep` are stored as local (stack) variables. This ensures that `rt_OneStep` is reentrant. If `rt_OneStep` is reinterrupted, higher priority tasks preempt lower priority tasks. Upon return from interrupt, lower priority tasks resume in the previously scheduled order.

Overrun Detection

Multirate `rt_OneStep` also maintains an array of timer overrun flags. `rt_OneStep` detects timer overrun, per task, by the same logic as single-rate `rt_OneStep`.

Note If you have developed multirate S-functions, or if you use a customized static main program module, see "Rate Grouping Compliance and Compatibility Issues" on page 34-19 for information about how to adapt your code for rate grouping compatibility. This adaptation lets your multirate, multitasking models generate more efficient code.

Multirate Single-Tasking Operation

In a multirate single-tasking program, by definition, all sample times in the model must be an integer multiple of the model's fixed-step size.

In a multirate single-tasking program, blocks execute at different rates, but under the same task identifier. The operation of `rt_OneStep`, in this case, is a simplified version of multirate multitasking operation. Rate grouping is not used. The only task is the base-rate task. Therefore, only one *model*_step function is generated:

```
void model_step(int_T tid)
```

On each clock tick, `rt_OneStep` checks the overrun flag and calls *model*_step, passing in `tid 0`. The scheduling function for a multirate single-tasking program is `rate_scheduler` (rather than `rate_monotonic_scheduler`). The scheduler maintains scheduling counters on each clock tick. There is one counter for each sample rate in the model. The counters are implemented in an array (indexed on tid) within the `Timing` structure within `rtModel`.

The counters are, in effect, clock rate dividers that count up the sample period associated with each subrate task. When a counter indicates that a sample period for a given rate has elapsed, `rate_scheduler` clears the counter. This condition indicates that all blocks running at that rate should execute on the next call to *model*_step, which is responsible for checking the counters.

Guidelines for Modifying rt_OneStep

`rt_OneStep` does not require extensive modification. The only required modification is to reenable interrupts after the overrun flags and error conditions have been checked. If applicable, you should also

- Save and restore your FPU context on entry and exit to `rt_OneStep`.
- Set model inputs associated with the base rate before calling *model*_step0.
- Get model outputs associated with the base rate after calling *model*_step0.

Note If you modify `rt_OneStep` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline below.

- In a multirate, multitasking model, set model inputs associated with subrates before calling *model*_step*N* in the subrate loop.

- In a multirate, multitasking model, get model outputs associated with subrates after calling *model*_step*N* in the subrate loop.

Comments in rt_OneStep indicate the appropriate place to add your code.

In multirate rt_OneStep, you can improve performance by unrolling for and while loops.

In addition, you may choose to modify the overrun behavior to continue execution after error recovery is complete.

Also observe the following cautionary guidelines:

- You should not modify the way in which the counters, event flags, or other timing data structures are set in rt_OneStep, or in functions called from rt_OneStep. The rt_OneStep timing data structures (including rtModel) and logic are critical to correct operation of any generated program.
- If you have customized ert_main.c or .cpp to read model outputs after each base-rate model step, be aware that selecting model options **Support: continuous time** and **Single output/update function** together may cause output values read from ert_main for a continuous output port to differ slightly from the corresponding output values in the model's logged data. This is because, while logged data is a snapshot of output at major time steps, output read from ert_main after the base-rate model step potentially reflects intervening minor time steps. To eliminate the discrepancy, either separate the generated output and update functions (clear the **Single output/update function** option) or place a Zero-Order Hold block before the continuous output port.

Static Main Program Module

In this section...

"Overview" on page 34-14
"Rate Grouping and the Static Main Program" on page 34-15
"Modifying the Static Main Program" on page 34-16

Overview

In most cases, the easiest strategy for deploying generated code is to use the **Generate an example main program option** to generate the `ert_main.c` or `.cpp` module (see "Generating a Standalone Program" on page 34-3).

However, if you turn the **Generate an example main program** option off, you can use the module *`matlabroot`*`/rtw/c/ert/ert_main.c` as a template example for developing your embedded applications. The module is not part of the generated code; it is provided as a basis for your custom modifications, and for use in simulation. If your existing applications, developed prior to this release, depend upon a static `ert_main.c`, you may need to continue using this module.

When developing applications using a static `ert_main.c`, you should copy this module to your working directory and rename it to *`model`*`_ert_main.c` before making modifications. Also, you must modify the template makefile such that the build process creates *`model`*`_ert_main.obj` (on UNIX, *`model`*`_ert_main.o`) in the build directory.

The static `ert_main.c` contains

- `rt_OneStep`, a timer interrupt service routine (ISR). `rt_OneStep` calls *`model`*`_step` to execute processing for one clock period of the model.
- A skeletal `main` function. As provided, `main` is useful in simulation only. You must modify `main` for real-time interrupt-driven execution.

For single-rate models, the operation of `rt_OneStep` and the main function are essentially the same in the static version of `ert_main.c` as they are in the autogenerated version described in "About Standalone Program Execution"

on page 34-2. For multirate, multitasking models, however, the static and generated code is slightly different. The next section describes this case.

Rate Grouping and the Static Main Program

Targets based on the ERT target sometimes use a static `ert_main` module and disallow use of the **Generate an example main program** option. This may be necessary because target-specific modifications have been added to the static `ert_main.c`, and these modifications would not be preserved if the main program were regenerated.

Your `ert_main` module may or may not use rate grouping compatible *`model`*`_step`*`N`* functions. If your `ert_main` module is based on the static `ert_main.c` module, it does not use rate-specific *`model`*`_step`*`N`* function calls. The static `ert_main.c` module uses the old-style *`model`*`_step` function, passing in a task identifier:

```
void model_step(int_T tid);
```

By default, when the **Generate an example main program** option is off, the ERT target generates a *`model`*`_step` "wrapper" for multirate, multitasking models. The purpose of the wrapper is to interface the rate-specific *`model`*`_step`*`N`* functions to the old-style call. The wrapper code dispatches to the appropriate *`model`*`_step`*`N`* call with a `switch` statement, as in the following example:

```
void mymodel_step(int_T tid) /* Sample time:  */
{

  switch(tid) {
   case 0 :
    mymodel_step0();
    break;
   case 1 :
    mymodel_step1();
    break;
   case 2 :
    mymodel_step2();
    break;
   default :
    break;
```

```
    }
  }
```

The following pseudocode shows how rt_OneStep calls *model*_step from the static main program in a multirate, multitasking model.

```
rt_OneStep()
{
  Check for base-rate interrupt overflow
  Enable "rt_OneStep" interrupt
  Determine which rates need to run this time step

  ModelStep(tid=0)     --base-rate time step

  For N=1:NumTasks-1  -- iterate over sub-rate tasks
    Check for sub-rate interrupt overflow
    If (sub-rate task N is scheduled)
      ModelStep(tid=N)    --sub-rate time step
    EndIf
  EndFor
}
```

You can use the TLC variable RateBasedStepFcn to specify that only the rate-based step functions are generated, without the wrapper function. If your target calls the rate grouping compatible *model*_step*N* function directly, set RateBasedStepFcn to 1. In this case, the wrapper function is not generated.

You should set RateBasedStepFcn prior to the %include "codegenentry.tlc" statement in your system target file. Alternatively, you can set RateBasedStepFcn in your target_settings.tlc file.

Modifying the Static Main Program

As in the generated ert_main.c, a few modifications to the main loop and rt_OneStep are necessary. See "Guidelines for Modifying the Main Program" on page 34-5 and "Guidelines for Modifying rt_OneStep" on page 34-12.

Also, you should replace the rt_OneStep call in the main loop with a background task call or null statement.

Other modifications you may need to make are

- If your model has multiple rates, the generated code does not operate correctly unless:
  - The multirate scheduling code is removed. The relevant code is tagged with the keyword `REMOVE` in comments (see also the Version 3.0 comments in `ert_main.c`).
  - Use the `MODEL_SETEVENTS` macro (defined in `ert_main.c`) to set the event flags instead of accessing the flags directly. The relevant code is tagged with the keyword `REPLACE` in comments.
- If applicable, follow comments in the code regarding where to add code for reading/writing model I/O and saving/restoring FPU context.

Note If you modify `ert_main.c` to read a value from a continuous output port after each base-rate model step, see the relevant cautionary guideline in "Guidelines for Modifying rt_OneStep" on page 34-12.

- When the **Generate an example main program** option is off `autobuild.h` is generated to provide an interface between the main module and generated model code. If you create your own static main program module, you would normally include `autobuild.h`.

  Alternatively, you can suppress generation of `autobuild.h`, and include *`model`*`.h` directly in your main module. To suppress generation of `autobuild.h`, use the following statement in your system target file:

  ```
  %assign AutoBuildProcedure = 0
  ```

- If you have cleared the **Terminate function required** option, remove or comment out the following in your production version of `ert_main.c`:
  - The `#if TERMFCN...` compile-time error check
  - The call to `MODEL_TERMINATE`
- If you do *not* want to combine output and update functions, clear the **Single output/update function** option and make the following changes in your production version of `ert_main.c`:

  - Replace calls to MODEL_STEP with calls to MODEL_OUTPUT and MODEL_UPDATE.
  - Remove the #if ONESTEPFCN... error check.
- The static ert_main.c module does not support the **Generate Reusable Code** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **Generate Reusable Code** is used illegally.

```
#if MULTI_INSTANCE_CODE==1
```

- The static ert_main.c module does not support the **External mode** option. Use this option only if you are generating a main program. The following error check raises a compile-time error if **External mode** is used illegally.

```
#ifdef EXT_MODE
```

Rate Grouping Compliance and Compatibility Issues

In this section...
"Main Program Compatibility" on page 34-19
"Making Your S-Functions Rate Grouping Compliant" on page 34-19

Main Program Compatibility

When the **Generate an example main program** option is off, code generation produces slightly different rate grouping code, for compatibility with the older static `ert_main.c` module. See "Rate Grouping and the Static Main Program" on page 34-15 for details.

Making Your S-Functions Rate Grouping Compliant

All built-in Simulink blocks, as well as all DSP System Toolbox blocks, are compliant with the requirements for generating rate grouping code. However, user-written multirate inlined S-functions may not be rate grouping compliant. Noncompliant blocks generate less efficient code, but are otherwise compatible with rate grouping. To take full advantage of the efficiency of rate grouping, your multirate inlined S-functions must be upgraded to be fully rate grouping compliant. You should upgrade your TLC S-function implementations, as described in this section.

Use of noncompliant multirate blocks to generate rate-grouping code generates dead code. This can cause two problems:

- Reduced code efficiency.
- Warning messages issued at compile time. Such warnings are caused when dead code references temporary variables before initialization. Since the dead code never runs, this problem does not affect the run-time behavior of the generated code.

To make your S-functions rate grouping compliant, you can use the following TLC functions to generate `ModelOutputs` and `ModelUpdate` code, respectively:

```
OutputsForTID(block, system, tid)
UpdateForTID(block, system, tid)
```

The code listings below illustrate generation of output computations without rate grouping (Listing 1) and with rate grouping (Listing 2). Note the following:

- The `tid` argument is a task identifier (`0..NumTasks-1`).
- Only code guarded by the `tid` passed in to `OutputsForTID` is generated. The `if (%<LibIsSFcnSampleHit(portName)>)` test is not used in `OutputsForTID`.
- When generating rate grouping code, `OutputsForTID` and/or `UpdateForTID` is called during code generation. When generating non-rate-grouping code, `Outputs` and/or `Update` is called.
- In rate grouping compliant code, the top-level `Outputs` and/or `Update` functions call `OutputsForTID` and/or `UpdateForTID` functions for each rate (`tid`) involved in the block. The code returned by `OutputsForTID` and/or `UpdateForTID` must be guarded by the corresponding `tid` guard:

  ```
  if (%<LibIsSFcnSampleHit(portName)>)
  ```

  as in Listing 2.

Listing 1: Outputs Code Generation Without Rate Grouping

```
%% multirate_blk.tlc

%implements "multirate_blk" "C"


%% Function: mdlOutputs =====================================================
%% Abstract:
%%
%%  Compute the two outputs (input signal decimated by the
%%  specified parameter). The decimation is handled by sample times.
%%  The decimation is only performed if the block is enabled.
%%  Each ports has a different rate.
%%
%%  Note, the usage of the enable should really be protected such that
%%  Neach task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
```

```
%%
  %function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign enable = LibBlockInputSignal(0, "", "", 0)
  {
    int_T *enabled = &%<LibBlockIWork(0, "", "", 0)>;

    %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
      %% Only check the enable signal on a major time step.
      if (%<LibIsMajorTimeStep()> && ...
                           %<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %else
      if (%<LibIsSFcnSampleHit("InputPortIdx0")>) {
        *enabled = (%<enable> > 0.0);
      }
    %endif

    if (*enabled) {
      %assign signal = LibBlockInputSignal(1, "", "", 0)
      if (%<LibIsSFcnSampleHit("OutputPortIdx0")>) {
        %assign y = LibBlockOutputSignal(0, "", "", 0)
        %<y> = %<signal>;
      }
      if (%<LibIsSFcnSampleHit("OutputPortIdx1")>) {
        %assign y = LibBlockOutputSignal(1, "", "", 0)
        %<y> = %<signal>;
      }
    }
  }

  %endfunction
%% [EOF] sfun_multirate.tlc
```

Listing 2: Outputs Code Generation With Rate Grouping

```
%% example_multirateblk.tlc

%implements "example_multirateblk" "C"
```

```
%% Function: mdlOutputs =====================================================
%% Abstract:
%%
%% Compute the two outputs (the input signal decimated by the
%% specified parameter). The decimation is handled by sample times.
%% The decimation is only performed if the block is enabled.
%% All ports have different sample rate.
%%
%% Note: the usage of the enable should really be protected such that
%% each task has its own enable state. In this example, the enable
%% occurs immediately which may or may not be the expected behavior.
%%
%function Outputs(block, system) Output



%assign portIdxName = ["InputPortIdx0","OutputPortIdx0","OutputPortIdx1"]
%assign portTID     = [%<LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")>, ...
                      %<LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")>]
%foreach i = 3
  %assign portName = portIdxName[i]
  %assign tid      = portTID[i]
  if (%<LibIsSFcnSampleHit(portName)>) {
                     %<OutputsForTID(block,system,tid)>
  }
%endforeach

%endfunction

%function OutputsForTID(block, system, tid) Output
/* %<Type> Block: %<Name> */
%assign enable = LibBlockInputSignal(0, "", "", 0)
%assign enabled = LibBlockIWork(0, "", "", 0)
%assign signal = LibBlockInputSignal(1, "", "", 0)

%switch(tid)
  %case LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx0")
```

```
                                 %if LibGetSFcnTIDType("InputPortIdx0") == "continuous"
                                   %% Only check the enable signal on a major time step.
                                   if (%<LibIsMajorTimeStep()>) {
                                     %<enabled> = (%<enable> > 0.0);
                                   }
                                 %else
                                   %<enabled> = (%<enable> > 0.0);
                                 %endif
                                 %break
      %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx0")
                                 if (%<enabled>) {
                                   %assign y = LibBlockOutputSignal(0, "", "", 0)
                                   %<y> = %<signal>;
                                 }
                                 %break
      %case LibGetGlobalTIDFromLocalSFcnTID("OutputPortIdx1")
                                 if (%<enabled>) {
                                   %assign y = LibBlockOutputSignal(1, "", "", 0)
                                   %<y> = %<signal>;
                                 }
                                 %break
      %default
                                 %% error it out
    %endswitch

    %endfunction

  %% [EOF] sfun_multirate.tlc
```

Wind River Systems VxWorks Example Main Program

- "Introduction to the VxWorks Example Main Program" on page 35-2
- "Task Management" on page 35-3

Introduction to the VxWorks Example Main Program

The Embedded Coder product provides a Wind River Systems VxWorks example main program as a template for the deployment of generated code in a real-time operating system (RTOS) environment. You should read the preceding sections of this chapter as a prerequisite to working with the VxWorks example main program. An understanding of the Embedded Coder scheduling and tasking concepts and algorithms, described in "About Standalone Program Execution" on page 34-2, is essential to understanding how generated code is adapted to an RTOS.

In addition, an understanding of how tasks are managed under the VxWorks RTOS is required. See your VxWorks documentation.

To generate a VxWorks example program:

1. In the **Custom templates** subpane of the **Code Generation > Templates** pane of the Configuration Parameters dialog box, select the **Generate an example main program** option (this option is on by default).

2. When **Generate an example main program** is selected, the **Target operating system** menu is enabled. Select `VxWorksExample` from this menu.

Some modifications to the generated code are required; comments in the generated code identify the required modifications.

Task Management

In this section...

"Overview of Operation" on page 35-3

"Single-Rate Single-tasking Operation" on page 35-3

"Multirate Multitasking Operation" on page 35-4

"Multirate Single-tasking Operation" on page 35-4

Overview of Operation

In a VxWorks example program, the main program and the base rate and subrate tasks (if any) run as prioritized tasks. The logic of a VxWorks example program parallels that of a stand-alone program; the main difference lies in the fact that base rate and subrate tasks are activated by clock semaphores managed by the operating system, rather than directly by timer interrupts.

Your application code must spawn `model_main()` as an independent VxWorks task. The task priority you specify is passed in to `model_main()`.

As with a stand-alone program, the VxWorks example program architecture is tailored to the number of rates in the model and to the solver mode (see Permitted Solver Modes for Embedded Coder™ Targeted Models on page 34-7). The following sections discuss each possible case.

Single-Rate Single-tasking Operation

In a single-rate, single-tasking model, `model_main()` spawns a base rate task, `tBaseRate`. In this case `tBaseRate` is the functional equivalent to `rtOneStep`. The base rate task is activated by a clock semaphore provided by the VxWorks RTOS, rather than by a timer interrupt. On each activation, `tBaseRate` calls `model_step`.

Note that the clock rate granted by the VxWorks RTOS may not be the same as the rate requested by `model_main`.

Multirate Multitasking Operation

In a multirate, multitasking model, `model_main()` spawns a base rate task and subrate tasks. Task priorities are assigned by rate.

As in a stand-alone program, rate grouping code is used (where possible) for multirate, multitasking models. The base rate task calls `model_step0`, while the subrate tasks call `model_stepN`. The base rate task calls a function that updates flags—an active task flag for each subrate and rate transition flags for tasks that exchange data. This function assumes the use of a rate-monotonic scheduler.

Multirate Single-tasking Operation

In a multirate, single-tasking model, `model_main()` spawns only a base rate task, `tBaseRate`. All rates run under this task. The base rate task is activated by a clock semaphore provided by the VxWorks RTOS, rather than by a timer interrupt. On each activation, `tBaseRate` calls `model_step`.

`model_step` in turn calls the `rate_scheduler` utility, which maintains the scheduling counters that determine which rates should execute. `model_step` is responsible for checking the counters.

Verifying Generated Code Applications

- Chapter 36, “Tracing Generated Code to Requirements”
- Chapter 37, “Verifying Generated Code”
- Chapter 38, “Rapid Prototyping On a Target System”
- Chapter 39, “Verifying Generated Code With SIL and PIL Simulations”
- Chapter 40, “Verifying a Component in the Target Environment”
- Chapter 41, “Verifying a Component by Building a Complete Real-Time Target Environment”
- Chapter 42, “Verifying Numerical Equivalence of Results with Code Generation Verification API”

Tracing Generated Code to Requirements

- "About Generated Code and Requirements Traceability" on page 36-2
- "Goals of Generated Code and Requirements Traceability" on page 36-3

About Generated Code and Requirements Traceability

Assuming that you have captured application requirements in a document, spreadsheet, data base, or requirements management tool, using a tool such as Simulink Report Generator, Microsoft Word, Microsoft Excel, raw HTML, Telelogic®, or DOORS®, you can use interactive traceability and traceability reports to validate whether generated code meets the documented requirements. These mechanisms provide a way to trace generated code back to documented requirements and generate traceability reports.

Goals of Generated Code and Requirements Traceability

For example, you can

- Associate requirements documents with objects in a concept model and generate a report on requirements associated with that model. For more information, see:
  - `slvnvdemo_fuelsys_docreq`
  - “Requirements Linking and Traceability” in the Simulink Verification and Validation documentation
  - Bidirectional tracing in Microsoft Word, Microsoft Excel, HTML, and Telelogic DOORS
- Include requirements tags in generated code. For more information, see:
  - rtwdemo_requirements
  - “Including Requirements Information with Generated Code” in the Simulink Verification and Validation documentation
- Trace model blocks and subsystems to generated code and vice versa. For more information, see:
  - rtwdemo_hyperlinks
  - “About Traceability” on page 37-2

37

Verifying Generated Code

- "Traceability for Production Code Generation" on page 37-2
- "Checking Code Correctness" on page 37-11

Traceability for Production Code Generation

In this section...

"About Traceability" on page 37-2
"Tracing Code to Model Objects Using Hyperlinks" on page 37-2
"Tracing Model Objects to Generated Code" on page 37-4
"Reloading Existing Traceability Information" on page 37-6
"Customizing Traceability Reports" on page 37-8
"Generating a Traceability Matrix (DO Qualification Kit or IEC Certification Kit)" on page 37-9
"Traceability Limitations" on page 37-10

About Traceability

The Simulink Coder product introduces traceability capabilities:

- "About Code Traceability"
- "Format of Traceability Tags"
- "Examples of Tagged Code"
- "Tracing Code To Blocks Using hilite_system"
- "Traceability Limitations"

The Embedded Coder product includes traceability capabilities to support:

- "Tracing Code to Model Objects Using Hyperlinks" on page 37-2
- "Tracing Model Objects to Generated Code" on page 37-4
- "Reloading Existing Traceability Information" on page 37-6
- "Customizing Traceability Reports" on page 37-8

Tracing Code to Model Objects Using Hyperlinks

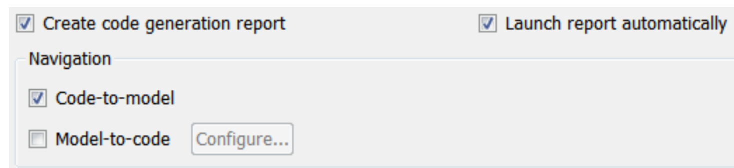
When using the Simulink Coder product, you can trace code to model objects using the `hilite_system` command. The Embedded Coder product simplifies

traceability with the use of hyperlinks in HTML code generation reports. The reports display hyperlinks in “Regarding,” “Outport,” and other comment lines in generated code. You can highlight the corresponding block or subsystem in the model diagram by clicking the hyperlinks.

To use hyperlinks for tracing code to model objects:

1 Open the model and make sure it is configured for an ERT target.

2 In the Configuration Parameters dialog box, select **Code Generation > Report Create code generation report**. The parameter is selected by default. When selected, the parameter enables and selects **Launch report automatically** and **Code-to-model**.

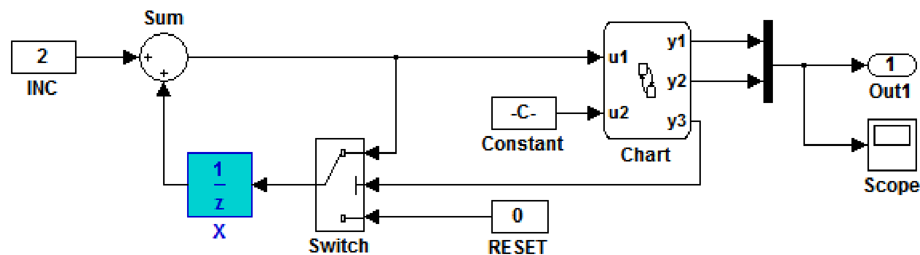


3 Build or generate code for the model. An HTML code generation report is displayed.

4 In the HTML report window, click hyperlinks to highlight source blocks. For example, generate an HTML report for model `rtwdemo_hyperlinks`. In the generated code for the model step function in `rtwdemo_hyperlinks.c`, click the first UnitDelay block hyperlink .

```
130  /* Model step function */
131  void rtwdemo_hyperlinks_step(void)
132  {
133    static real_T tmp[5] = { 10.0, -20.0, 30.0, -40.0, 50.0 };
134
135    /* Sum: '<Root>/Sum' incorporates:
136     *  Constant: '<Root>/INC'
137     *  Switch: '<Root>/Switch'
138     *  UnitDelay: '<Root>/X'
139     *  Update for UnitDelay: '<Root>/X'
140     */
141    rtDWork.X += 2.0;
```

In the model window, the corresponding UnitDelay block is highlighted.



For more information on generating HTML code generation reports or using the `hilite_system` command to trace code to blocks, see the following topics in the Simulink Coder documentation:

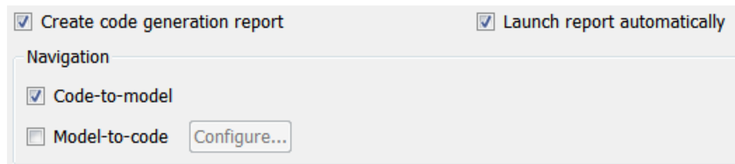
- "Generating Reports for Code Reviews and Traceability Analysis"
- "Tracing Generated Code"

Tracing Model Objects to Generated Code

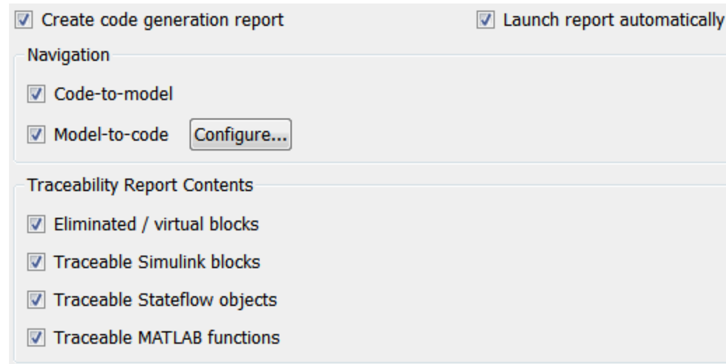
To trace model objects to generated code:

1. Open the model and make sure it is configured for an ERT target.

2. In the Configuration Parameters dialog box, select **Code Generation > Report > Create code generation report**. The parameter is selected by default. When selected, the parameter enables and selects the **Launch report automatically** and **Code-to-model** parameters.



3. Select **Model-to-code**.



This parameter:

- Enables the **Configure** button, which opens a dialog box for loading existing trace information.
- Enables and selects parameters for customizing the content of a traceability report.

4 Build or generate code for the model. An HTML code generation report is displayed.

5 In the model window, right-click a model object.

6 In the context menu, select **Code Generation > Navigate to Code**. In the HTML code generation report, you see the first instance of highlighted code generated for the model object. In the left pane of the report, numbers that appear to the right of generated file names indicate the total number of highlighted lines in each file. The following figure shows the result of tracing the Unit Delay block in model `rtwdemo_hyperlinks`.

```
130 /* Model step function */
131 void rtwdemo_hyperlinks_step(void)
132 {
133   static real_T tmp[5] = { 10.0, -20.0, 30.0, -40.0, 50.0 };
134
135   /* Sum: '<Root>/Sum' incorporates:
136    *  Constant: '<Root>/INC'
137    *  Switch: '<Root>/Switch'
138    *  UnitDelay: '<Root>/X'
139    *  Update for UnitDelay: '<Root>/X'
140    */
141   rtDWork.X += 2.0;
142
143   /* Stateflow: '<Root>/Chart' incorporates:
144    *  Constant: '<Root>/Constant'
145    *  Switch: '<Root>/Switch'
146    *  Update for UnitDelay: '<Root>/X'
147    */
```

To navigate through multiple instances of highlighted lines, click **Previous** and **Next**.

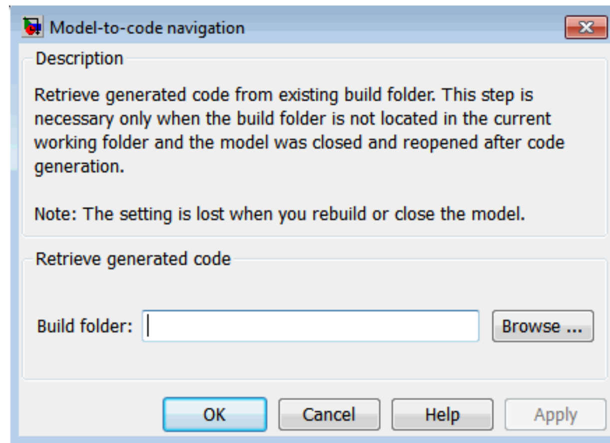
If you close and reopen a model, the **Navigate to Code** context menu option might not be available. This occurs because Embedded Coder cannot find a build directory for your model in the current working directory. To address this, do one of the following:

- Reset the current working directory to the parent directory of the existing build directory.
- Select **Model-to-code** and rebuild the model. This regenerates the build directory into the current working directory.
- Click **Configure** and in the Model-to-code navigation dialog box, reload the existing trace information.

Reloading Existing Traceability Information

To reload existing traceability information for a model:

1. In the Configuration Parameters dialog box, click **Code Generation > Report > Configure**. The Model-to-code navigation dialog box opens.



2 In the **Build directory** field, type or browse to the build directory that contains the existing traceability information.

If you close and reopen a model, the **Navigate to Code** context menu option might not be available. This occurs because Embedded Coder cannot find a build directory for your model in the current working directory. To fix this without having to reset the current working directory or rebuild the model, do the following:

1 Click **Configure** to open the Model-to-code navigation dialog box.

2 In the Model-to-code navigation dialog box, click **Browse**.

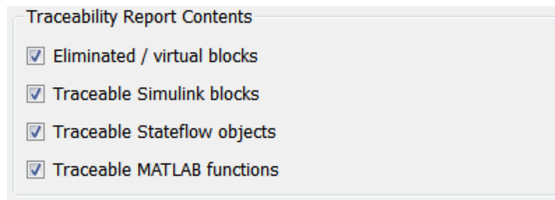
3 Browse to the build directory for your model, and select the directory. The build directory path is displayed in the **Build directory** field, as shown in the preceding figure.

4 Click **Apply** or **OK**. This loads traceability information from the earlier build into your Simulink session, provided that you selected **Model-to-code** for the build.

5 Right-click **Code Generation > Navigate to Code** to open the context menu and trace a model object to corresponding code.

Customizing Traceability Reports

In the Configuration Parameters dialog box, the **Code Generation > Report > Traceability Report Contents** section lists parameters you can select and clear to customize the content of your traceability reports. By default, all parameters are selected, as shown in the following figure.



Select or clear any combination of the following:

- **Eliminated / virtual blocks** (account for blocks that are untraceable)
- **Traceable Simulink blocks**
- **Traceable Stateflow objects**
- **Traceable MATLAB functions**

If you select all parameters, you get a complete mapping between model elements and the generated code.

The following figure shows the top section of the traceability report generated by selecting all traceability content parameters for model `rtwdemo_hyperlinks`.

Traceability Report for rtwdemo_hyperlinks

Generate Traceability Matrix

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - rtwdemo_hyperlinks
   - rtwdemo_hyperlinks/Chart
   - rtwdemo_hyperlinks/Chart:43

Eliminated / Virtual Blocks

Block Name	Comment
<Root>/Build ERT	Empty SubSystem
<Root>/Mux	Mux
<Root>/Scope	Unused code path elimination
<Root>/View Code Generation Report	Empty SubSystem

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Root system: rtwdemo_hyperlinks

Object Name	Code Location
<Root>/Chart	rtwdemo_hyperlinks.c:20, 43, 85, 103, 112, 143, 257 rtwdemo_hyperlinks.h:39, 40, 41, 43, 45, 46, 47, 48, 49, 52, 53
<Root>/Constant	rtwdemo_hyperlinks.c:144

Generating a Traceability Matrix (DO Qualification Kit or IEC Certification Kit)

If you are licensed for either DO Qualification Kit software or IEC Certification Kit software and are using a Windows host, you can generate a traceability matrix into Microsoft Excel format directly from the traceability report described in "Customizing Traceability Reports" on page 37-8.

To do this, go to the **Traceability Report** section of the HTML code generation report and click the **Generate Traceability Matrix** button.

Generate Traceability Matrix

When you click the button, a Generate Traceability Matrix dialog box appears. Use this dialog to select an existing matrix file to update or specify a new matrix file to create. Optionally, you can use this dialog to select and order the columns that appear in the generated matrix. For more information, see “Generating a Traceability Matrix” in either the DO Qualification Kit documentation or the IEC Certification Kit documentation.

Traceability Limitations

In addition to the Simulink Coder traceability limitations, the following limitations apply to reports generated by Embedded Coder software.

- Under the following conditions, model-to-code traceability is disabled for a block if the block name contains:
  - A single quote (`'`).
  - An asterisk (`*`), that causes a name-mangling ambiguity relative to other names in the model. This name-mangling ambiguity occurs if in a block name or at the end of a block name, an asterisk precedes or follows a slash (`/`).
  - The character (`char(255)`).
- You cannot trace blocks representing the following types of subsystems to generated code:
  - Virtual subsystems
  - Masked subsystems
  - Nonvirtual subsystems for which code has been optimized away

  If you cannot trace a subsystem at subsystem level, you might be able to trace individual blocks within the subsystem.

Checking Code Correctness

In this section...
"About Checking Code Correctness" on page 37-11
"How To Check Code Correctness" on page 37-11

About Checking Code Correctness

Checking code correctness involves verifying that no compile-time, link-time, or run-time errors (for example, an overflow, divide by zero, or out-of-bounds array access) are in the source code.

How To Check Code Correctness

You can rely on integrated development environment (IDE) tools to detect and facilitate correction of compile-time and link-time errors. However, it is more difficult to detect and correct run-time errors, such as overflows and division by zero. Some methods of detecting run-time errors include

- Running the executable and analyzing the results
- Inserting code instrumentation, such as print statements
- Writing and running tests

When checking the correctness of the generated code, you also have the option of using Polyspace products. Polyspace products are based on verification technology that uses formal methods to detect and mathematically prove whether classes of run-time errors, such as overflows, exist.

In addition, the Polyspace Model Link™ SL product lets you trace the results reported by Polyspace® Client™ for C/C++ software back to your Simulink model.

For more information about using Polyspace products, see the Polyspace documentation.

Rapid Prototyping On a Target System

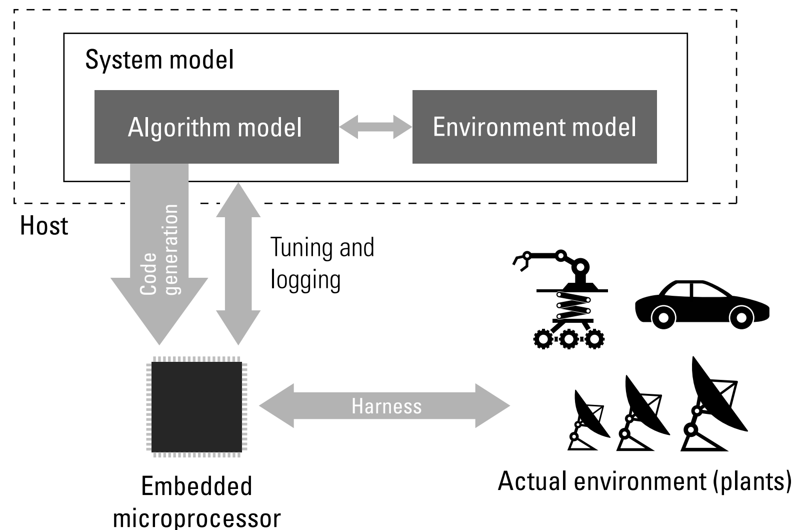
- "About On-Target Rapid Prototyping" on page 38-2
- "Goals of On-Target Rapid Prototyping" on page 38-3
- "Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping" on page 38-4

About On-Target Rapid Prototyping

After you refine a detailed software design, you are ready to generate code to run on an embedded microprocessor and optimize the code with on-target rapid prototyping. During on-target rapid prototyping, you run generated code in real time, tune parameters, and monitor real-time data on the same processor that you plan to use in mass production, or a close equivalent to it.

Code generation provides a framework for on-target rapid prototyping. You can generate code from your model and then assess, interact with, and optimize the code using real embedded compilers and hardware. This effort can help determine whether your algorithm can fit on or run fast enough for production devices, which typically have limited processor resources.

The following figure shows an example of an on-target rapid prototyping environment.



Goals of On-Target Rapid Prototyping

Assuming that you have a detailed software design and an embedded microprocessor target, you can use on-target rapid prototyping to:

- Refine the concept model of your component or system
- Test and validate model functionality in real time
- Test hardware
- Obtain real-time profiles and code metrics for analysis and sizing based on an embedded processor
- Assess the feasibility of an algorithm based on integration with environment or plant hardware

Optimizing Generated Code for an Embedded Processor With On-Target Rapid Prototyping

To do on-target rapid prototyping:

1. Generate the source code for your models, integrate the code into your production build environment, and run it on existing hardware. For more information see:
   - "Testing and Refining a Model With Rapid Prototyping" in the Simulink Coder documentation
   - "Selecting and Configuring a Target"
   - "Interfacing With a Real-Time Operating System "
2. Integrate existing, externally written C or C++ code with your model for simulation and code generation. For more information, see "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" in the Simulink documentation.
3. Use a third-party integrated development environment (IDE) or a makefile with the IDE link capability, a third-party product, or custom integration to build an executable for the embedded microprocessor.
4. To monitor signals, tune parameters, and log data as the embedded microprocessor controls the actual environment or plant, see the section on "Signal Monitoring and Parameter Tuning."
5. If using custom integration, use a Embedded Coder runtime interface option, such as external mode, C API, or ASAP2 file generation, to monitor and tune signals. Also consider using the Vehicle Network Toolbox™ product if you are developing a solution based on a controller-area network (CAN).

Verifying Generated Code With SIL and PIL Simulations

- "About SIL and PIL Simulations" on page 39-2
- "How SIL and PIL Simulations Work" on page 39-6
- "Comparison of SIL and PIL Simulation" on page 39-7
- "Choosing a SIL or PIL Approach" on page 39-9
- "Configuring a SIL or PIL Simulation" on page 39-16
- "Code Coverage" on page 39-25
- "Code Execution Profiling" on page 39-34
- "Running a Top Model as a SIL or PIL Simulation" on page 39-41
- "Running a Referenced Model as a SIL or PIL Simulation" on page 39-43
- "SIL and PIL Code Interfaces" on page 39-47
- "Configuring Hardware Implementation Settings for SIL" on page 39-49
- "Programming PIL Support for Third-Party Tools and Target Hardware" on page 39-53
- "Creating a Connectivity Configuration for a Target" on page 39-54
- "SIL and PIL Simulation Support and Limitations" on page 39-60

About SIL and PIL Simulations

In this section...

"Overview" on page 39-2
"What are SIL and PIL Simulations?" on page 39-2
"Why Use SIL and PIL" on page 39-3

Overview

Embedded Coder supports *software-in-the-loop* (SIL) and *processor-in-the-loop* (PIL) simulations, which allow you to verify generated source code and compiled object code.

A SIL simulation involves compiling and running production source code on your host computer, while a PIL simulation involves cross-compiling and running production object code on a target processor or an equivalent instruction set simulator.

You can use SIL and PIL simulations to verify the numerical correctness of your code, optimize your code, collect code metrics such as code coverage and execution profiling data, and achieve IEC 61508, ISO 26262, or DO-178 certification. See "Why Use SIL and PIL" on page 39-3.

For examples of SIL and PIL verification, see `rtwdemo_sil_pil_script`. For information about how you verify that your model is correctly configured for a SIL or PIL simulation, see "Verifying a SIL or PIL Configuration" on page 39-22.

What are SIL and PIL Simulations?

The Embedded Coder product supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations.

A SIL simulation involves compiling and running production source code on your host computer to verify the source code. SIL provides a convenient alternative to processor-in-the-loop (PIL) simulation as no target hardware (for example, an evaluation board or instruction set simulator) is required. For examples of SIL verification, see `rtwdemo_sil_pil_script.`

A PIL simulation involves cross-compiling and running production object code on a target processor or an equivalent instruction set simulator.

You can run a SIL or PIL simulation using:

- The **Software-in-the-Loop (SIL)** or **Processor-in-the-Loop (PIL)** simulation mode for top models and Model blocks
- A SIL or PIL block

For more information, see "Choosing a SIL or PIL Approach" on page 39-9.

The following features enable you to verify the generated code:

- Compare the output of regular simulation modes, for example, Normal or Accelerator, against the output of SIL and PIL simulation modes.
- Easily switch between regular simulation, SIL, and PIL modes.

You can model and test your embedded software component in Simulink and then reuse your test suites across simulation and compiled production code. This approach avoids the time-consuming process of leaving the Simulink software environment and verifying production code on a separate test infrastructure.

Why Use SIL and PIL

You can achieve early verification and fixing of defects when you use SIL and PIL. See "V-Model for System Development" in the Simulink Coder documentation.

The following table describes situations when you should use SIL and PIL.

Situation	Use...
You want to reuse test vectors developed for Normal mode simulation to verify numerical correctness of generated (or legacy) code. For example, reusing test cases generated by Simulink® Design Verifier™. See “Generating Test Cases” in Simulink Design Verifier documentation.	SIL and PIL
You want to collect metrics for generated code: • Code coverage. See “Using a Code Coverage Tool in a SIL Simulation” on page 39-25. • Execution profiling. See “Code Execution Profiling” on page 39-34 • Stack profiling. See “Stack Profiling” on page 44-21.	SIL and PIL
You want to achieve IEC 61508, ISO 26262, and DO-178 certification. See “Verification and Validation at the Code Level (Code Verification)” in the IEC Certification Kit documentation and Testing of Outputs of Integration Process in the DO Qualification Kit documentation.	SIL and PIL
You do not have target hardware and want a convenient alternative to PIL.	SIL
You have target hardware, for example, an evaluation board or instruction set simulator, and you want to: • Confirm correct behavior of target specific code, for example, Chapter 31, “Replacing Math Functions and Operators Using Target Function Libraries” optimizations, and legacy code. • Optimize the execution speed and memory footprint of your code. See, in this table, row with information about collecting execution profiling and stack profiling metrics. • Investigate effects of compiler settings and optimizations, for example, deviation from ANSI C overflow behavior. Normal simulation techniques do not account for restrictions and requirements that the hardware	PIL

Situation	Use...
imposes, such as limited memory resources or behavior of target-specific optimized code. See “Example Custom Targets” in the Simulink Coder documentation, which gives information about running PIL simulations on specific targets.	

How SIL and PIL Simulations Work

In a SIL/PIL simulation, code is generated for either the top model or part of the model. With SIL, this code is compiled for, and executed on the host computer. With PIL, the code is cross-compiled for the target hardware and runs on the target processor.

Through a communication channel, Simulink sends stimulus signals to the code on the host or target processor for each sample interval of the simulation:

- For a top model, Simulink uses stimulus signals from the base or model workspace.
- If you have designated only part of the model to simulate in SIL/PIL mode, then a part of the model remains in Simulink without the use of code generation. Typically, you configure this part of the model to provide test vectors for the software executing on the hardware. This part of the model can represent other parts of the algorithm or the environment model in which the algorithm operates.

When the host/target processor receives signals from Simulink, the processor executes the SIL/PIL algorithm for one sample step. The SIL/PIL algorithm returns output signals computed during this step to Simulink through a communication channel. At this point, one sample cycle of the simulation is complete and Simulink proceeds to the next sample interval. The process repeats and the simulation progresses. SIL/PIL simulations do not run in real time. At each sample period, Simulink and the object code exchange *all* I/O data. See also “Verifying Internal Signals of a Component” on page 39-43.

Comparison of SIL and PIL Simulation

Use SIL or PIL simulation to verify automatically generated code by comparing the results with a Normal mode simulation. With SIL, you can easily verify the behavior of production source code on your host computer. However, you cannot verify exactly the same code that is subsequently compiled for your target hardware because the code must be compiled for your host computer (that is, a different compiler and different processor architecture than the target). With PIL simulation, you can verify exactly the same code that you intend to deploy in production, and you can run the code on either real target hardware or an instruction set simulator. See "What are SIL and PIL Simulations?" on page 39-2.

You can use any of the following approaches to verification.

Approach	SIL	PIL
Simulation mode (for top model or Model block)	Generated production code compiled and executed on host computer as separate process, independent of the MATLAB process. Execution is host/host and nonreal time	Test the generated code as cross-compiled object code on target processor or instruction set simulator. Exercises same object code used in production software. Execution is host/target and nonreal time.
Block	Create SIL block. Software runs generated code through S-function wrapper on host computer. SIL S-function links directly with generated code, so generated code runs inside MATLAB process. Execution is host/host and nonreal time. See "Using a SIL or PIL Block" on page 39-20.	Create PIL block. Software runs cross-compiled object code through S-function wrapper on host computer. S-function communicates with object code executing as standalone application on target processor or instruction set simulator. Execution is host/target and nonreal time. See "Using a SIL or PIL Block" on page 39-20.

To decide which verification approach you want to use, see "Choosing a SIL or PIL Approach" on page 39-9 .

Choosing a SIL or PIL Approach

In this section...

"About Choosing a SIL or PIL Simulation" on page 39-9
"When to Use Top-Model SIL or PIL" on page 39-9
"When to Use Model Block SIL or PIL" on page 39-9
"When to Use the SIL or PIL Block" on page 39-14

About Choosing a SIL or PIL Simulation

This section describes how to choose the SIL or PIL verification approach for your needs.

For examples, see `rtwdemo_sil_pil_script`, which allow you to compare:

- SIL block for SIL Simulation
- SIL or PIL Simulation for Model Blocks
- SIL or PIL Simulation for Top Models

When to Use Top-Model SIL or PIL

Use the top-model approach if you want to:

- Verify code generated for a top model (standalone code interface).
- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal, SIL, or PIL simulation modes.

For an example, see `rtwdemo_sil_pil_script` .

When to Use Model Block SIL or PIL

Use the Model block approach if you want to:

- Verify code generated for referenced models (model reference code interface).

- Provide a test harness model (or a system model) to generate test vector or stimulus inputs.
- Switch a Model block between normal, SIL, or PIL simulation modes.

See “Modeling Scenarios with the Model Block” on page 39-10.

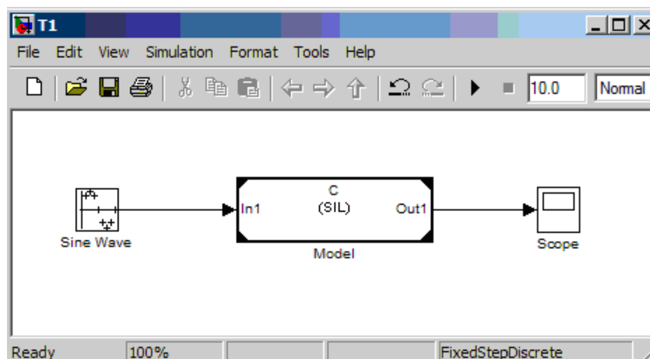
Modeling Scenarios with the Model Block

You can use the Model block to test single components or a whole hierarchy of model reference components. For example, you can select a single leaf component for SIL verification. Later in the development cycle, as your components become integrated into a larger system, you can select a hierarchy of components for SIL verification.

You must deploy your Model block component code as part of a standalone executable. The following examples show ways of testing your component.

- “Testing a Model Reference Component in SIL Mode” on page 39-10
- “Deploying Through an Atomic Subsystem” on page 39-11
- “Deploying Through a Top Model” on page 39-12

Testing a Model Reference Component in SIL Mode. You can test a model reference component or hierarchy of components by placing a Model block in a test harness model, as shown in model `T1`.



To test the component, for example, in SIL mode:

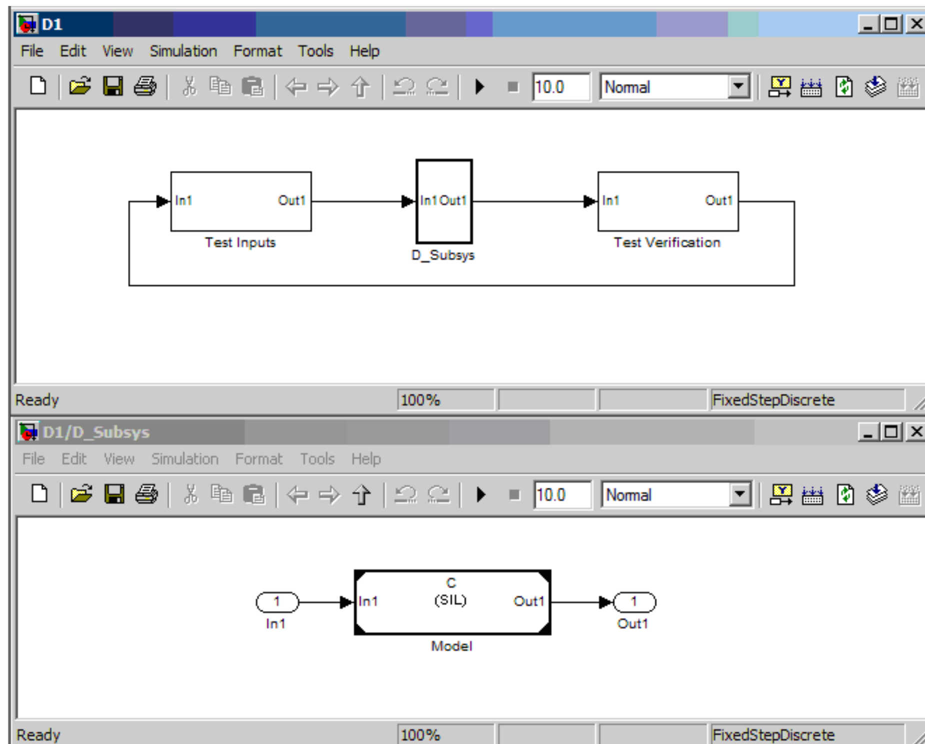
1. Set the simulation mode of component C to SIL mode.

2. Simulate the model to run component C in SIL mode, and test its model reference target.

Note If the model reference target code interface for component C does not already exist, simulating the model generates it. For more information about the model reference target code interface, see “SIL and PIL Code Interfaces” on page 39-47.

The following deployment scenarios reuse the model reference target of component C. This reuse ensures that you test exactly the same object code that you deploy.

Deploying Through an Atomic Subsystem. To generate code with the standalone interface for deployment, place a Model block inside an atomic subsystem, as shown by model `D1`.



To create standalone code, perform a subsystem build of `D_Subsys`. The standalone code calls the model reference target of component C.

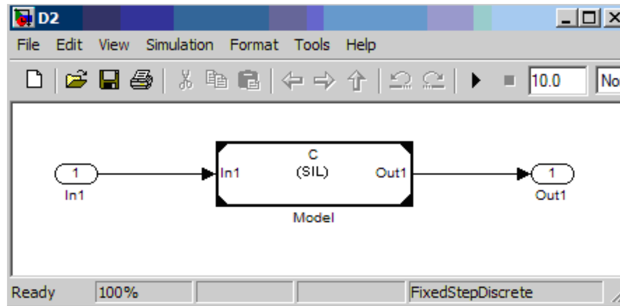
To test the component, for example, in SIL mode:

1. Set the simulation mode of component C to SIL mode.

2. Simulate the model to run component C in SIL mode and test its model reference target.

You can place multiple Model blocks and other blocks into the model to deploy a whole system of components.

Deploying Through a Top Model. To generate code with the standalone interface for deployment, place the Model block inside a top model, as shown by model `D2`.

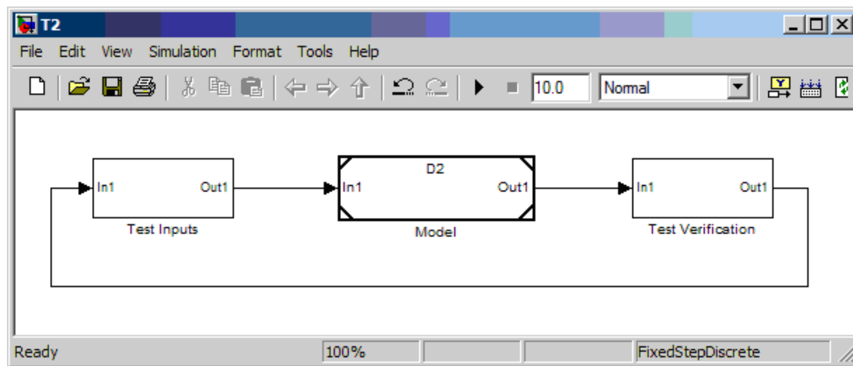


To create standalone code, perform a build of `D2`. The standalone code calls the model reference target of component C.

You can place multiple Model blocks and other blocks into the model to deploy a whole system of components.

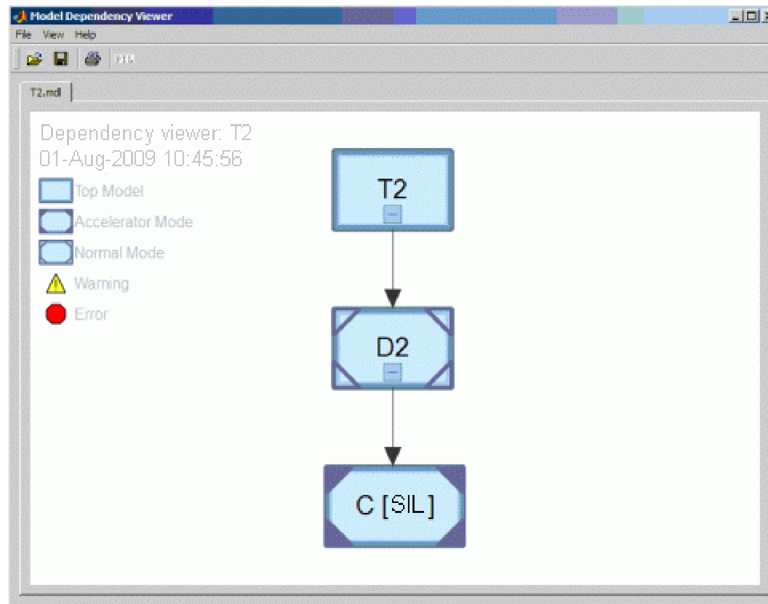
To pass test inputs to component C (running in SIL mode):

1 Create a test harness model that references model `D2` in Normal mode, as shown by model `T2`.



2 Simulate the `T2` model to run component C in SIL mode and test its model reference target.

The Model Dependency Viewer shows the model reference hierarchy of `T2` and the simulation modes of each Model block component.

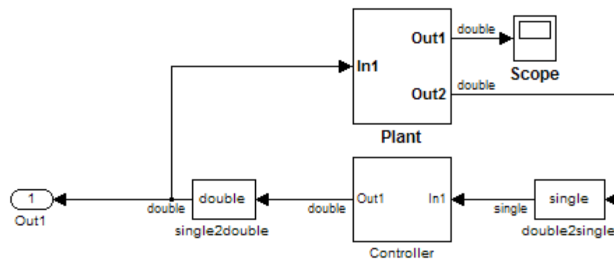


When to Use the SIL or PIL Block

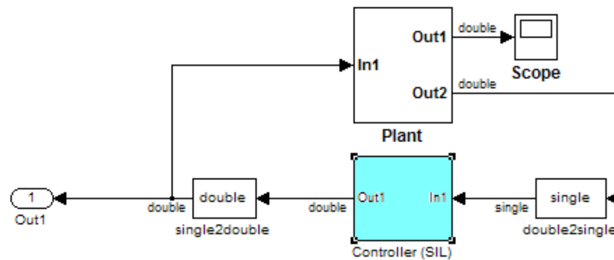
Use the SIL or PIL block if you want to:

- Verify code generated for a top model with a standalone code interface, or a subsystem with a (right-click build) standalone code interface. For more information about the standalone target code interface, see "SIL and PIL Code Interfaces" on page 39-47.
- Change the model and insert a SIL or PIL block to represent a component running in SIL or PIL mode, in a situation where a test harness model or a system model provides test vector or stimulus inputs.

  For example, you can replace the controller subsystem in the following model,



with a SIL block (highlighted) that represents the controller.



For information about how you create a SIL or PIL block, see “Using a SIL or PIL Block” on page 39-20.

Note If you compare a SIL or PIL block simulation with a top-model simulation, you see that you must perform two steps before you can run the simulation. First, you perform a right-click subsystem build to create the SIL or PIL block. Then, you replace the subsystem in the original model with the newly created SIL block.

Configuring a SIL or PIL Simulation

In this section...

"Top-Model SIL or PIL Simulation" on page 39-16
"Model Block SIL or PIL Simulation" on page 39-18
"Using a SIL or PIL Block" on page 39-20
"Verifying a SIL or PIL Configuration" on page 39-22
"Compatible Models" on page 39-23

Top-Model SIL or PIL Simulation

To configure and run a top-model SIL or PIL simulation:

1. Open your model.

2. Select either **Simulation > Software-in-the-Loop (SIL)** or **Simulation > Processor-in-the-Loop (PIL)**.

Note This option is available only if the model is configured for an ERT or AUTOSAR target. See "Code Generation Pane: General" and Chapter 24, "Generating Code for AUTOSAR Software Components" for configuration information.

---

3. If you have not already done so, in the Configuration Parameters dialog box, on the **Data Import/Export** pane:

   - In the **Input** check box and field, specify stimulus signals (or test vectors) for your top model.

   - Configure logging for model outputs, using either *output logging* or *signal logging*:

     – In the **Output** check box and field, specify *output logging*.

     – In the **Signal logging** check box and field, specify *signal logging*.

       The software logs only signals that connect to root-level inports and outports. See "Verifying Internal Signals of a Component" on page

39-43. If the root outports of your model connect to bus signals, then output logging is not available. Use signal logging for bus signals that connect to root outports.

If you name the signals, you can log signals that connect to inports or outports of the top model. If you select **Signal logging** but do not name signals that connect to inports or outports of the top model, then the signal logging object (for example, `logsout`) does not hold signal data.

- Disable logging of Data Store Memory variables. The software does not support this option for this simulation mode. If you do not clear the **Data stores** check box, the software produces a warning when you run the simulation.

For information about the **Data Import/Export** pane, see “Importing and Exporting Simulation Data” and “Data Import/Export Pane”.

4 If you are configuring a SIL simulation, specify one of the following:

- Hardware implementation settings that correspond to the host machine. See “Configuring the Hardware Implementation” in the Simulink Coder documentation.

  For a SIL simulation, you do not have to specify a value for the **Byte ordering** field on the **Hardware Implementation** pane. The software uses the byte ordering that the host computer uses.

- Portable word sizes. See “Configuring Hardware Implementation Settings for SIL” on page 39-49.

5 If required, configure:

- Code coverage. See “Code Coverage” on page 39-25.

- Code execution profiling. See “Configuring Code Execution Profiling” on page 39-34.

6 Start the simulation.

Note You cannot:

- Close the model while the simulation is running. To interrupt the simulation, in the Command Window, press **Ctrl+C**.
- Alter the model during the simulation. You can move blocks and lines as long as it does not alter the behavior of the model.

You can run a top-model SIL or PIL simulation using the command `sim(`*`model`*`)`.

Note The software supports the `sim` command options `SrcWorkspace` and `DstWorkspace` for only the following values:

- `SrcWorkspace` — `'base'`
- `DstWorkspace` — `'base'` or `'current'`

For more information on the `sim` command and its options, see "Simulation" in the Simulink documentation.

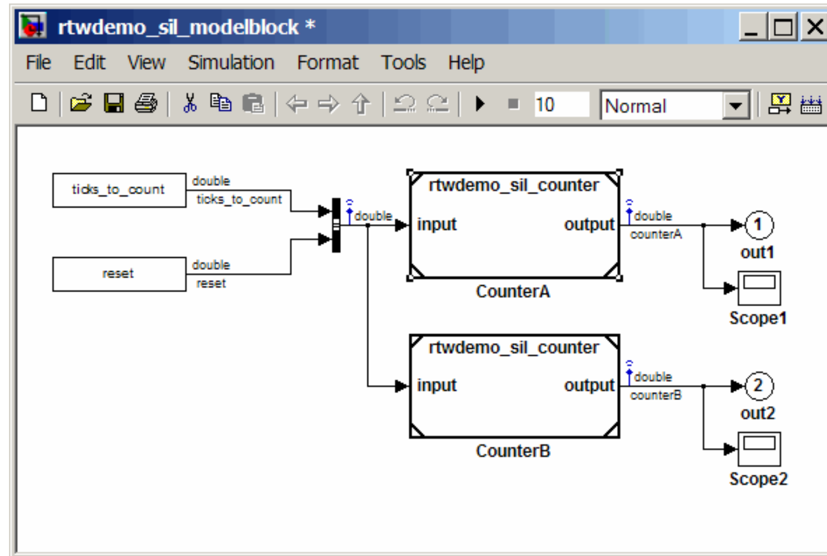
For information about how a simulation behaves when the top model contains a Model block (and this Model block is a parent Model block containing Model blocks at lower levels of its reference hierarchy), see "Simulation Mode Override Behavior in Model Reference Hierarchy" on page 39-44.

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See "Creating a Connectivity Configuration for a Target" on page 39-54.

Model Block SIL or PIL Simulation

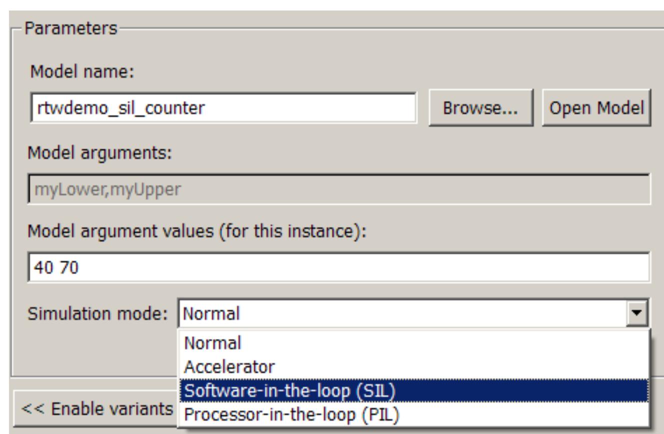
To configure a Model block for a SIL or PIL simulation:

1. Open your model.



2 Right-click your Model block, for example, `Counter A`. In the context menu, select **ModelReference Parameters** to open the Model Reference Parameters dialog box.

3 From the **Simulation Mode** drop-down list, select the required mode, for example, `Software-in-the-loop (SIL)`.



4 If configuring a SIL simulation, select one of the following:

- Hardware implementation settings that correspond to the host machine. See "Configuring the Hardware Implementation" in the Simulink Coder documentation.

  You do not have to specify a value for the **Byte ordering** field on the Hardware Implementation pane. The software uses the byte ordering that the host computer uses.

- Portable word sizes. See Setting Up a Model to Generate Code for Host Simulations and Target Deployment

5 Configure code coverage, if required. See "Code Coverage" on page 39-25.

6 If you require code execution profiling for your Model block, then configure execution profiling for the top model. See "Configuring Code Execution Profiling" on page 39-34.

7 Start the simulation.

Note For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See "Creating a Connectivity Configuration for a Target" on page 39-54.

Using a SIL or PIL Block

You can automatically create a SIL or PIL block from a complete model or a subsystem. You can use this block to test the code generated from your model:

1 In the Configuration Parameters dialog box, select **Code Generation > SIL and PIL Verification**.

2 From the **Create block** drop-down list, select either `SIL` or `PIL`.

3 If you want to enable code execution profiling for a PIL block:

   a Select the **Collect execution time measurements** check box.

   b In the **Workspace variable field**, specify a name.
   The software does not support code execution profiling in SIL blocks. For more information, see "Code Execution Profiling" on page 39-34.

4 Click **OK**.

5 In your model window, right-click the subsystem that you want to simulate.

6 Select **Code Generation > Build Subsystem**.

7 Click **Build** to start a subsystem build that generates a SIL or PIL block for the generated subsystem code.

8 Add the generated block to an environment or test harness model that supplies test vectors or stimulus input.

9 Run simulations with the environment or test harness model to perform SIL or PIL tests.

10 Verify that the generated code captured in the SIL or PIL block provides the same result as the original subsystem.

Note You cannot create a SIL or PIL block (**Create block** appears dimmed) if you do one of the following:

- Disable either the `CreateSILPILBlock` or `GenerateErtSFunction` property
- Select a code coverage tool

For a PIL simulation, you control the way code compiles and executes in the target environment through connectivity configurations. See "Creating a Connectivity Configuration for a Target" on page 39-54.

For an example of how the SIL block is used in testing, see `rtwdemo_sil_pil_script`.

For a description of the SIL block as an S-function wrapper, see Chapter 26, "Generating S-Function Wrappers".

Verifying a SIL or PIL Configuration

You might need to change model settings to configure the model correctly for SIL or PIL. To find out what settings you must change, use the `cgv.Config` class. Using the `cgv.Config` class, you can review your model configuration and determine which settings you must change to configure the model correctly for SIL or PIL. By default, `cgv.Config` changes configuration parameter values to the value that it recommends, but does not save the model. Alternatively, you can specify that `cgv.Config` use one of the following approaches:

- Change configuration parameter values to the values that `cgv.Config` recommends, and save the model. Specify this approach using the `SaveModel` property.
- List the values that `cgv.Config` recommends for the configuration parameters, but do not change the configuration parameters or the model. Specify this approach using the `ReportOnly` property.

Note

- To execute the model in the target environment successfully, you might need to make additional modifications to the configuration parameter values or the model.
- Do not use referenced configuration sets in models that you are changing using `cgv.Config`. If the model uses a referenced configuration set, update the model with a copy of the configuration set. Use the `Simulink.ConfigSetRef.getRefConfigSet` method. For more information, see `Simulink.ConfigSetRef` in the Simulink documentation.
- If you use `cgv.Config` on a model that executes a callback function, the callback function might change configuration parameter values each time the model loads. The callback function might revert changes that `cgv.Config` made. When this change occurs, the model might no longer be set up correctly for SIL or PIL. For more information, see "Using Callback Functions".

For more information about the `cgv.Config` class, see `cgv.Config`.

How To Verify a SIL or PIL Configuration

To verify that your model is configured correctly:

1. Construct a `cgv.Config` object that changes the configuration parameter values without saving the model. For example, to configure your model for SIL:

   ```
   c = cgv.Config('vdp', 'connectivity', 'sil');
   ```

   Tip

   - You can obtain a list of changes without changing the configuration parameter values. When you construct the object, include the `'ReportOnly', 'on'` property name and value pair.
   - You can change the configuration parameter values and save the model. When you construct the object, include the `'SaveModel', 'on'` property name and value pair.

   ---

2. Determine and change the configuration parameter values that the object recommends using the `configModel` method. For example:

   ```
   c.configModel();
   ```

3. Display a report of the changes that `configModel` makes. For example:

   ```
   c.displayReport();
   ```

4. Review the changes.

5. To apply the changes to your model, save the model.

Compatible Models

You can use the SIL simulation mode with any Simulink model, if:

1. The model specifies an ERT-based target.

2. The model specifies one of the following template makefiles:

   a. `ert_default_tmf`

b `ert_unix.tmf`

c `ert_vc.tmf`

d `ert_vcx64.tmf`

e `ert_lcc.tmf`

The software does not support the Watcom compiler template makefile (`ert_watc.tmf`).

3 The host and target word sizes match. For example, if your host is a Windows XP computer, then you must specify, through the **Hardware Implementation** pane, a hardware device with the following word sizes (in bits):

- **char** — 8
- **short** — 16
- **int** — 32
- **long** — 32
- **native word size** — 32

See also "Configuring Hardware Implementation Settings for SIL" on page 39-49.

Code Coverage

In this section...

"Using a Code Coverage Tool in a SIL Simulation" on page 39-25

"Code Coverage for a PIL Simulation" on page 39-32

"Configuring Code Coverage Programmatically" on page 39-32

Using a Code Coverage Tool in a SIL Simulation

During a top-model or Model block SIL simulation, you can collect code coverage metrics for generated code using a third-party tool.

Embedded Coder supports the BullseyeCoverage™ tool from Bullseye Testing Technology™. MathWorks does not supply this tool. For information about installing and using this tool, go to http://www.bullseye.com/cgi-bin/mwEval.

To configure a code coverage tool for a top-model or Model block SIL simulation:

1. Select **Simulation > Configuration Parameters > Code Generation > SIL and PIL Verification**.

2. From the **Code coverage tool** drop-down list, select a tool, for example, `BullseyeCoverage`.

3. Click **Configure Coverage** to open the Code Coverage Settings dialog box.

4. In the **Installation folder** field, specify the location where your coverage tool is installed.

5. Specify the models for which you want code coverage data:

   - To generate coverage data for just the current (top) model, select the **Code coverage for this model** check box.

   - To generate data for models referenced by the current (top) model, select the **Code coverage for referenced models** check box.

Note If you do not select a check box, the software does *not* generate code coverage data.

---

6 Click **OK**. You return to the **SIL and PIL Verification** pane.

7 To view cumulative code coverage results within a code generation report, in **Configuration Parameters > Code Generation > Report**, select the following check boxes:

   - **Create code generation report**
   - **Launch report automatically**

   Note On a Linux® platform, the default browser that displays the report automatically does not support JavaScript. As a result, the code coverage information does not appear in the report. Selecting the **Launch report automatically** check box is unnecessary. At the end of the simulation, a link to the code generation report appears in the Command Window. This link is configured to open the report with an external browser that supports JavaScript. Click this link to view the report with code coverage information.

   .

8 Click **OK**. You return to the model window.

Note During the SIL simulation, the code coverage software checks the time stamp of each source file to determine whether the source file is newer than the code coverage data generated for the file. If the source file is newer, the software produces an error. As you may edit a file manually after compilation (with code coverage configured), this software behavior prevents collection of code coverage data for object code that is stale.

When the simulation is complete, the code generation report opens automatically. This report provides summary data and code annotations with coverage information.

```
Back    Forward

Contents
Summary
Subsystem Report
Code Interface Report
Traceability Report

Generated Files
[-] Main file
    ert_main.c
[-] Model files
    rtwdemo_sil_topmodel.c
    rtwdemo_sil_topmodel.h
[+] Utility files (1)

File: C:\Work\work_r\rtwdemo_sil_topmodel_ert_rtw\rtwdemo_sil_topmodel.c

Function coverage: 100%  Condition/decision coverage: 50%

 1  /*
 2   * File: rtwdemo_sil_topmodel.c
 3   *
 4   * Code generated for Simulink model 'rtwdemo_sil_topmodel'.
 5   *
 6   * Model version                  : 1.195
 7   * Simulink Coder version         : 8.0 (R2011a) 28-Dec-2010
 8   * TLC version                    : 8.0 (Dec 29 2010)
 9   * C/C++ source code generated on : Tue Jan 04 11:09:43 2011
10   *
11   * Target selection: ert.tlc
12   * Embedded hardware selection: Specified
```

The cumulative coverage data in a code generation report is derived from instrumented files associated with your latest top model simulation **and** coverage data collected from simulations with other top models that share referenced models with your current top model.

In the following example, there are two annotations. At line 41, TF indicates that the `if` decision had both true and false outcomes during the simulation. At line 52, =>F indicates that the `if` decision was false only during the simulation.

```
TF   41    if (rtU.reset) {
     42      rtDWork.PreviousOutput_DSTATE = 20U;
     43    }
     44
     45    /* Switch: '<Root>/Switch' incorporates:
     46     *  Constant: '<Root>/C1'
     47     *  Constant: '<Root>/C5'
     48     *  Inport: '<Root>/ ticks_to_count'
     49     *  RelationalOperator: '<Root>/upper GE input'
     50     *  Sum: '<Root>/Add'
     51     */
=>F  52    if ((uint8_T)((uint32_T)rtU.ticks_to_count + (uint32
     53                  rtDWork.PreviousOutput_DSTATE) == 40)
```

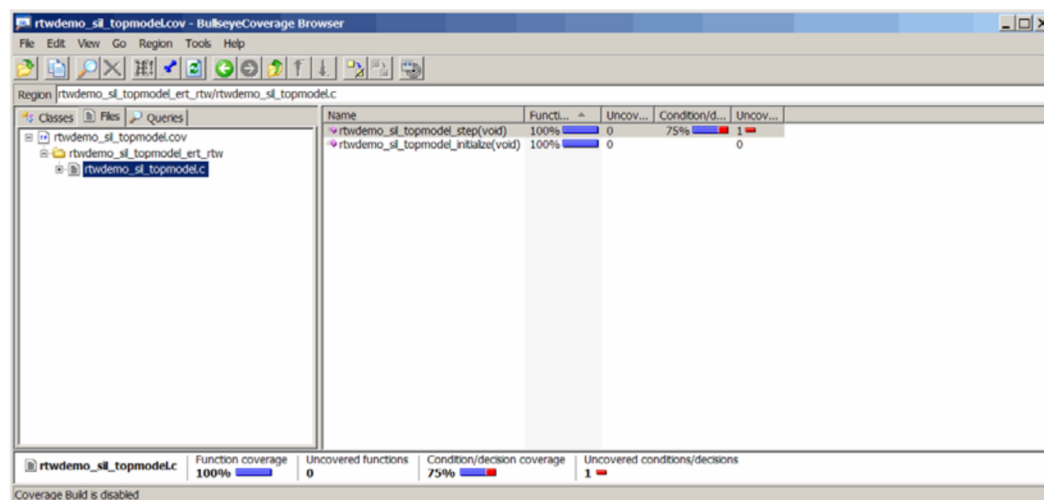
For a list of annotations, see "Code Coverage Annotations in Code Generation Report" on page 39-29.

The code generation report allows you to navigate easily between blocks in your model and the corresponding sections in the source code. See "Tracing Model Objects to Generated Code" on page 37-4 and "Tracing Code to Model Objects Using Hyperlinks" on page 37-2.

When the SIL simulation is complete, you see two hyperlinks in the Command Window.

```
### Starting Real-Time Workshop build procedure for model: rtwdemo_sil_topmodel
### Successful completion of Real-Time Workshop build procedure for model: rtwdemo_sil_topmodel
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_sil_topmodel
### Stopping SIL simulation for model: rtwdemo_sil_topmodel
Code coverage data was collected; you can open the code coverage report in the BullseyeCoverage browser.
You can view code coverage data for rtwdemo_sil_topmodel_ert_rtw in Real-Time Workshop report.
```

To view the coverage report for the simulation in the BullseyeCoverage Browser, click the first link.



The BullseyeCoverage Browser shows cumulative coverage data for all instrumented files associated with your latest top model simulation. The coverage data shown in the browser is not cumulative and pertains only to the most recent simulation.

For information about the BullseyeCoverage Browser, go to www.bullseye.com.

For an example on collecting code coverage metrics, view the demo `rtwdemo_code_coverage_script`.

Code Coverage Annotations in Code Generation Report

The following table describes the code annotations you may see in a code generation report produced by a SIL simulation. For information on configuring code coverage annotation, see “Using a Code Coverage Tool in a SIL Simulation” on page 39-25.

Code feature	Annotation symbol	What happened during simulation
Decision	=>	Decision not executed
	TF	Decision evaluated both true and false
	=>T	Decision evaluated true only
	=>F	Decision evaluated false only
Function	=>	Function not called
	Fcn	Function called
Switch label	=>	Switch command not used
	Sw	Switch command used
Constant	k	Decision or condition was constant, which did not allow any variation in coverage.
Condition	=>	Condition not encountered
	tf	Condition evaluated both true and false
	=>t	Condition evaluated true only
	=>f	Condition evaluated false only

Tips and Limitations

Compilers and Platforms Supported for SIL Code Coverage. For SIL code coverage, the software supports the following compilers and platforms:

- On a Linux platform, `gcc`
- On a Windows platform, `vc`

The software does *not* support the Lcc or Watcom compiler. If either compiler is specified with code coverage, the software produces an error when you build your model. For information on how to specify a compiler, see "Choosing and Configuring a Compiler" in the Simulink Coder documentation.

BullseyeCoverage Tool Support. Embedded Coder is tested with version 7.14.20 of the BullseyeCoverage tool on the following operating systems:

- Windows 32-bit
- Windows 64-bit
- Linux 32-bit
- Linux 64-bit

Embedded Coder support for the BullseyeCoverage tool is untested on the Macintosh® 64-bit operating system.

BullseyeCoverage License Wait. When you build your model, you may have to wait for a BullseyeCoverage license. If you want to see information about the wait, before you build your model, select **Code Generation > Debug > Verbose build**.

Characters in `matlabroot` and File Path. If `matlabroot` or the path to your generated files contains a space or the `.` (period) character, code coverage may fail.

Code Coverage Instrumentation in a Standalone Executable. If you build your model and the model is configured for code coverage, then the generated standalone executable contains instrumentation for collecting code coverage metrics. However, MathWorks recommends that you use only the SIL or PIL simulation mode to generate code coverage metrics.

`.cov` Files in Build Folders. The software uses only the `.cov` file in the code generation working folder to collect code coverage metrics for the entire model hierarchy. You can see other `.cov` files in the build folders for individual model components, for example, the top model and referenced models. However, these files are intermediate files, which the software does not use during a SIL simulation.

Code Coverage for Source Files in Shared Utility Folders. The software supports code coverage for source files generated in shared utility folders. If you configure code coverage for a model that uses shared utility code generation, when you build the model, you also build all source files in the shared utilities folder with code coverage enabled.

Whenever you build a model, the code coverage settings of the model must be consistent with source files that you previously built in the shared utilities folder. Otherwise, the software reports that code in the shared utilities folder is inconsistent with the current model configuration and must be rebuilt. For example, if you run a SIL simulation for a model with code coverage enabled and then a SIL simulation for another model with code coverage disabled, the software must rebuild all source files in the shared utilities folder.

The BullseyeCoverage tool, by default, does not provide code coverage data for inline macros.

For example, if a model generates a file `slprj/ert/_sharedutils/rt_SATURATE.h` that contains the macro

```
#define rt_SATURATE(sig,ll,ul) (((sig) >= (ul)) ?  (ul) :
(((sig) <= (ll)) ?  (ll) :  (sig)) )
```

and the macro is in `sat_ert_rtw/sat.c`, then the coverage report provides a measurement for `sat.c`, but no coverage data for the conditions within the macro `rt_SATURATE`.

To configure the BullseyeCoverage tool to provide code coverage data for inline macros:

1. Open the BullseyeCoverage Browser.
2. Select **Tools > Options** to open the Options dialog box.
3. On the **Build** tab, select the **Instrument macro expansions** check box.
4. Click **OK**.
5. Rerun your simulation.

Alternatively, you can add the text -macro to the BullseyeCoverage configuration file. For more information, go to http://www.bullseye.com/help/ref_covc.html.

Code Coverage for a PIL Simulation

You can configure code coverage for top-model and Model block SIL simulations. See "Using a Code Coverage Tool in a SIL Simulation" on page 39-25. Code coverage is also available for the corresponding PIL simulations provided your PIL application can write directly to the host file system. Your target for the PIL application must provide fopen and fread access to the host file system.

If code coverage is not available when you run the PIL application on your target hardware, you may be able to collect code coverage measurements by running the PIL application on an instruction set simulator that supports direct file I/O with the host file system.

Configuring Code Coverage Programmatically

You can configure code coverage for your model using command line APIs. The following example shows a typical workflow.

1 Using get_param, retrieve the object containing coverage settings for the current model, for example, gcs.

```
>> covSettings = get_param(gcs, 'CodeCoverageSettings')

covSettings =

  cov.CodeCoverageSettings handle
  Package: cov

  Properties:
           TopModelCoverage: 'on'
    ReferencedModelCoverage: 'off'
               CoverageTool: 'BullseyeCoverage'

  Methods, Events, Superclasses
```

Note The property `TopModelCoverage` determines whether the software generates code coverage data for just the top model, while `ReferencedModelCoverage` determines whether the software generates coverage data for models referenced by the top model. If neither property is `'on'`, then no code coverage data is generated during a SIL simulation.

When you save your model, the properties `TopModelCoverage`, `ReferencedModelCoverage`, and `CoverageTool` are also saved.

2 Check the class of `covSettings`.

```
>> class(covSettings)

ans =

cov.CodeCoverageSettings
```

3 Switch on coverage for referenced models.

```
>> covSettings.ReferencedModelCoverage='on';
```

4 Using `set_param`, apply the new coverage settings to the model.

```
>>set_param(gcs,'CodeCoverageSettings', covSettings);
```

5 Assuming you have installed the BullseyeCoverage tool, specify the installation path.

```
>> cov.BullseyeCoverage.setPath('C:\Program Files\BullseyeCoverage')
```

6 Check that the path has been saved as a preference.

```
>> cov.BullseyeCoverage.getPath
```

Code Execution Profiling

In this section...

"About Code Execution Profiling" on page 39-34

"Configuring Code Execution Profiling" on page 39-34

"Viewing and Analyzing Code Execution Profiles" on page 39-35

"Tips and Limitations" on page 39-39

About Code Execution Profiling

Use this feature to collect a profile of execution time for each task within your generated code.

You can collect execution time measurements in a specified base workspace variable during a SIL or PIL simulation. See "Configuring Code Execution Profiling" on page 39-34.

At the end of the simulation, you can view or analyze the measurements within the MATLAB environment. See "Viewing and Analyzing Code Execution Profiles" on page 39-35.

Note The software supports code execution profiling for all types of SIL and PIL simulations, with the exception of *SIL block* simulations.

Configuring Code Execution Profiling

To configure code execution profiling for a SIL or PIL simulation:

1. Select **Simulation > Configuration Parameters > Code Generation > SIL and PIL Verification**.

2. Select the **Collect execution time measurements** check box.

3. In the **Workspace variable** field, specify a name. When you run the simulation, the software generates a variable with this name. The variable

contains the execution time measurements, and is an object of type `rtw.pil.ExecutionProfile`.

4 Click **OK**.

For a PIL simulation, you must configure a hardware-specific timer. When you set up the connectivity configuration for your target, create a timer object. See “Creating a Connectivity Configuration for a Target” on page 39-54. This action is not required for a SIL simulation.

How Profiling Settings Apply to Model and PIL Blocks

If your top model contains a Model block configured for SIL or PIL, the execution profiling settings of the top model also apply to the Model block.

If your top model has a PIL block, the execution profiling settings that apply to the PIL block are the settings from the original model that you used to create the PIL block. See “Using a SIL or PIL Block” on page 39-20. The execution profiling settings of your top model have no effect on the PIL block.

Viewing and Analyzing Code Execution Profiles

After a SIL or PIL simulation, from the Command Window, you can view and analyze execution profile data using methods from the `rtw.pil.ExecutionProfile` and `rtw.pil.ExecutionProfileSection` classes.

The `rtw.pil.ExecutionProfile` class provides the following methods:

- `getNumSectionProfiles` — Get number of code sections for which profiling data is available
- `getSectionProfile` — Get `rtw.pil.ExecutionProfileSection` object
- `getTimerTicksPerSecond` — Get number of timer ticks per second
- `setTimerTicksPerSecond` — Set number of timer ticks per second
- `display` — Display summary in Command Window

The `rtw.pil.ExecutionProfileSection` class provides the following methods:

- `getName` — Get name of profiled code section
- `getSamplePeriod` — Get sample time associated with profiled section of code
- `getSampleOffset` — Get sample offset associated with profiled section of code
- `getTicks` — Get vector of execution times (in timer ticks) for profiled section of code
- `getTimes` — Get vector of execution times (in seconds) for profiled section of code

For more information about these methods, see "Code Execution Profiling".

Code Execution Profiling Example

Suppose you do the following with `rtwdemo_sil_topmodel`:

1. Enable code execution profiling.
2. Specify a workspace variable `myExecutionProfile`.
3. Run a SIL simulation.

The software creates an `rtw.pil.ExecutionProfile` object.

```
>> rtwdemo_sil_topmodel
### Starting build procedure for model: rtwdemo_sil_topmodel
### Successful completion of build procedure for model: rtwdemo_sil_topmodel
### Preparing to start SIL simulation ...
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
>> whos
  Name                    Size        Bytes  Class                          Attributes
  T                       1x1             8  double
  myExecutionProfile      1x1            60  rtw.pil.ExecutionProfile
  out                     1x1           180  Simulink.SimulationOutput
  reset                   1x1          1413  struct
```

```
  ticks_to_count       1x1         1413  struct
```

You can use the method `display` to provide a summary of all profiled code sections. Alternatively, enter the name of the workspace variable.

```
>> myExecutionProfile

                                         Minimum       Average       Maximum

1. rtwdemo_sil_topmodel_step [0.1 0] : 121           141           352

>>
```

In this example, there is only one code section with profiling data. The software displays the following information for the task `rtwdemo_sil_topmodel_step`:

- Sample period (`0.1` seconds)
- Sample offset (`0`)
- Execution times for the code section — minimum (`121`), average (`141`), and maximum (`352`). Since the timer is uncalibrated, the values represent timer ticks.

To get the total number of code sections that have profiling data, use the `getNumSectionProfiles` method.

```
>> no_of_Sections = myExecutionProfile.getNumSectionProfiles

no_of_Sections =

     1
>>
```

To get the `rtw.pil.ExecutionProfileSection` object for a profiled code section, use the method `getSectionProfile`.

```
>> FirstSectionProfile = myExecutionProfile.getSectionProfile(1)

  rtw.pil.ExecutionProfileSection

    Section name = rtwdemo_sil_topmodel_step
```

```
    Sample period = 0.1
    Sample offset = 0

>>
```

Use `rtw.pil.ExecutionProfileSection` methods to extract profiling information for a particular code section. For example, use `getName` to obtain the name of a profiled code section.

```
>> name_of_section = FirstSectionProfile.getName

name_of_section =

rtwdemo_sil_topmodel_step

>>
```

To get the sample time associated with the profiled code section, use the method `getSamplePeriod`.

```
>> sample_time = FirstSectionProfile.getSamplePeriod

sample_time =

    0.1000

>>
```

If the timer is uncalibrated, applying the method `getTimes` returns an empty array.

```
>> execution_times = FirstSectionProfile.getTimes

execution_times =

     []

>>
```

However, if you know the timer rate, for example 2.2 GHz, you can use the `rtw.pil.ExecutionProfile` method `setTimerTicksPerSecond` to calibrate the timer.

```
>> myExecutionProfile.setTimerTicksPerSecond(2.2e9)
>> FirstSectionProfile = myExecutionProfile.getSectionProfile(1);
>>
```

You can then use `getTimes` to generate a vector of execution times for the code section, and extract, for example, the minimum and maximum execution times.

```
>> execution_times = FirstSectionProfile.getTimes;
>> whos execution_times
  Name                  Size             Bytes  Class     Attributes

  execution_times      1x101               808  double

>> minmax(execution_times)

ans =

  1.0e-006 *

    0.0550    0.1600

>>
```

Tips and Limitations

Triggered Model Block

Consider the case where a triggered Model block is configured to run in the SIL or PIL simulation mode. The software generates one execution time measurement each time the referenced model is triggered to run. If there are multiple triggers in a single time step, there are multiple measurements for the triggered Model block. Conversely, if there is no trigger in a given time step, the software generates no time measurements.

Outliers in Execution Profiles

The operating system may preempt a SIL application after the start of a measurement, making the execution profiling result for the time step unreliable. As a consequence of preemption, you may see outliers in your code execution profiles, with execution times that are longer than expected.

Additionally, for execution time measurements greater than 2^{32} ticks, the counter wraps. Counter wrapping occurs when the actual execution time is very long, which results in a *measured* execution time that is shorter than expected.

Execution Times with Separate Output and Update Functions

If you clear the check box **Configuration Parameters > Code Generation > Interface > Single output/update function** before building your model, the generated code has separate output and update functions. The measured execution time for each step is the sum of the execution times for the separate functions.

Running a Top Model as a SIL or PIL Simulation

With a top-model SIL or PIL simulation:

- Simulink generates and executes code that uses the same code interface produced by the standalone build process. See “SIL and PIL Code Interfaces” on page 39-47.
- You can specify external stimulus signals and log output signals, which allows you to verify object code generated from a complete model without creating a separate test harness model. Running the SIL or PIL simulation is a simple operation.

Top-model SIL simulation is an alternative to the block-based approach where you provide a test harness model that wraps a Model block (in SIL mode). Two differences between the block-based approach and top-model SIL simulation are:

- With Model block SIL simulation, the model reference target that is generated does not have the same interface as standalone code (see “SIL and PIL Code Interfaces” on page 39-47).
- With Model block SIL simulations, you cannot directly specify external stimulus signals or enable signal logging. You must use input and output blocks to feed signals into and out of your model. See “Verifying Internal Signals of a Component” on page 39-43 and “Choosing a SIL or PIL Approach” on page 39-9.

To compare all SIL simulation options, see “Choosing a SIL or PIL Approach” on page 39-9.

For the top-model SIL approach, Simulink creates a hidden *wrapper* model. When you run a top-model SIL simulation, the software generates code for the model and creates a hidden wrapper model to call this code at each time step.

- If there are errors during a SIL simulation, you may see messages that refer to the wrapper model. For wrapper model error messages, the wrapper model is made visible to allow you to investigate the error.
- For *signal logging*, the software adds the suffix `_wrapper` to the block path for signals in `logsout`, as shown in the following example:

```
>> logsout.SignalLogging

          Name: 'SignalLogging'
     BlockPath: 'sillogging_wrapper/sillogging'
     PortIndex: 1
    SignalName: 'SignalLogging'
    ParentName: 'SignalLogging'
      TimeInfo: [1x1 Simulink.TimeInfo]
          Time: [11x1 double]
          Data: [11x1 double]
```

- For *output logging*, if the save format is `Structure` or `Structure with time`, the software adds the suffix `_wrapper` to the block name for signals in `yout`, as shown in the following example:

```
>> yout.signals

ans =
        values: [11x1 double]
    dimensions: 1
         label: 'SignalLogging'
     blockName: 'sillogging_wrapper/OutputLogging'
```

  If the save format is `Array`, then the software does not add a wrapper suffix.

Running a Referenced Model as a SIL or PIL Simulation

In addition to the regular simulation modes, Model blocks have a `Software-in-the-loop (SIL)` and `Processor-in-the-loop (PIL)` mode.

You can switch the Model block between regular, SIL, and PIL simulation modes. This allows you to easily verify the generated code by executing the referenced model as compiled code on the host computer or target platform. You can model and test your embedded software component in Simulink and you can reuse your regression test suites across simulation and compiled object code. This capability avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for your production hardware.

The label `(SIL)` or `(PIL)` on the block indicates the mode of the Model block.

To understand how SIL or PIL works in the Model block, see the following information:

- “Simulation Mode Override Behavior in Model Reference Hierarchy” on page 39-44
- “SIL and PIL Code Interfaces” on page 39-47
- “When to Use Model Block SIL or PIL” on page 39-9
- “Modeling Scenarios with the Model Block” on page 39-10

For an introduction to the Model block, see the Model Variants block section in the Simulink reference documentation.

Verifying Internal Signals of a Component

Outputs of the SIL or PIL component are available for verification. If you want to examine an internal signal, you can:

- Manually route the signal to the top level.
- Use global data stores to access internal signals:
  1. Inside the component, connect a Data Store Write block to the required signal.

2 Outside the component, use a Data Store Read block to access the signal value.

   See “Working with Data Stores” and “Global Data Store Example” in the Simulink documentation.

- Use MAT-file logging. See “Setting Up Runtime Logging to MAT-Files” in the Simulink Coder documentation. For PIL, target environment must support MAT-file logging.

For more information on signal support, see “I/O Support” on page 39-71.

Simulation Mode Override Behavior in Model Reference Hierarchy

This section describes simulation behavior when the top model contains a Model block. This Model may also be a parent block containing child Model blocks at lower levels of its reference hierarchy.

Note You can view your model hierarchy in the Model Dependency Viewer. In the Referenced Model Instances view, the software displays Model blocks differently to indicate their simulation modes, for example, Normal, Accelerator, SIL, and PIL. In this view, the software does not indicate the simulation mode of the top model.

You can specify the simulation mode of a top model to be Normal, Accelerator, Rapid Accelerator, SIL, or PIL. With a Model block, you can specify all modes *except* Rapid Accelerator. The configured simulation mode of a Model block may be overridden by the parent simulation mode. The following table shows how the software determines the effective simulation mode of any Model block in the hierarchy.

Mode of top model or parent block	Mode of parent or child block in reference hierarchy			
	Normal	Accelerator	SIL	PIL
Normal	Equivalent	Compatible	Compatible	Compatible
Accelerator	Override	Equivalent	Error	Error
Rapid Accelerator	Override	Override	Error	Error
SIL	Override	Override	Equivalent	Error
PIL	Override	Override	Error	Equivalent

The following list explains the different types of simulation behavior:

- Equivalent – Both parent and child Model block run in the same simulation mode.
- Compatible – If the simulation mode of the top model or parent block is Normal, then the software simulates the child block in the mode specified for it.
- Error – The simulation produces an error. For example, if a top model or parent Model block has simulation mode Accelerator but contains a child block in SIL or PIL mode, then running a simulation produces an error: the Accelerator mode can never override the SIL and PIL mode of child blocks. This behavior avoids the risk of “false positives”, that is, the successful simulation of a model in Accelerator mode will never lead to the conclusion that generated source or object code of child Model blocks has been tested or verified.
- Override – The simulation mode of the top model or parent Model block overrides the simulation mode of the child block. For example, if a top model or parent Model block that is configured for a SIL simulation contains a child Model block with simulation mode Normal or Accelerator, then the software simulates the child block in SIL mode. This override behavior:
  - Allows a Model block anywhere in the reference hierarchy to have the SIL or PIL mode.
  - Ensures that if you simulate the top model or parent Model block in SIL or PIL mode, all lower-level referenced models execute in SIL or PIL

mode. You do not have to switch manually the simulation mode of every model component in the hierarchy.

For an example model hierarchy, see "Modeling Scenarios with the Model Block" on page 39-10.

SIL and PIL Code Interfaces

This section describes and compares the different code interfaces that the code generation products produce.

You generate standalone code when you perform a top-model or right-click subsystem build for a single deployable component. You can compile and link standalone code into a standalone executable or integrate it with other code. For more information on the standalone code interface, see Chapter 33, “Model Entry Points”.

When you generate code for a referenced model hierarchy, the software generates standalone executable code for the top model, and a library module called a *model reference target* for each referenced model. When the code executes, the standalone executable invokes the applicable model reference targets to compute the referenced model outputs. For more information, see “Creating Model Components” in the Simulink Coder documentation.

Note The model reference target does not have the same code interface as standalone code.

If you intend to integrate automatically generated code with legacy code, use standalone code because the standalone code interface (for example, entry points) is fully documented.

SIL/PIL Feature	Standalone Code Interface	Model Reference Code Interface
Top-model	Yes	No (but you can include Model blocks inside your top model)
Model block	No	Yes
SIL or PIL block	Yes	No

Code Interface for Top-Model SIL or PIL

Top-model SIL or PIL generates the *standalone code interface* for the model.

When you run a top-model SIL or PIL simulation, the software calls the standalone code for the model if it already exists. The software generates the standalone code if it does not exist.

Code Interface for Model Block SIL or PIL

Model block SIL or PIL mode generates the *model reference* code interface.

When you run a simulation with a Model block in SIL or PIL mode, the software calls the model reference target for the Model block if it already exists, or generates the model reference target.

If the model reference target does not yet exist, you can generate it in one of three ways:

- Run the simulation.
- Press **Ctrl+B** to build the top model containing the Model block.
- Use the command `slbuild`, specifying the model reference option, for example:

```
slbuild('model','ModelReferenceRTWTargetOnly')
```

You cannot use standalone code with the Model block. You can generate standalone code for a model referenced by a Model block by opening the model and performing a top-level build. However, you cannot use this standalone code with Model block SIL or PIL simulation.

For more information, see the table in "Choosing a SIL or PIL Approach" on page 39-9.

Configuring Hardware Implementation Settings for SIL

In this section...

"Compiling Generated Code That Supports Portable Word Sizes" on page 39-51

"Portable Word Sizes Limitations" on page 39-51

Embedded Coder provides an option to specify portable word sizes. If you select this option for a model, you can use the same generated source code files for:

- Software-in-the-loop (SIL) simulation on the host computer
- Production deployment on the target platform

If you do not specify portable word sizes for a SIL simulation, you can configure the model to use an emulation hardware option. For integer and fixed-point operations, this option guarantees bit-true agreement between host computer simulation and target deployment. See "Configuring the Hardware Implementation" in the Simulink Coder documentation.

Note If processor word sizes differ between host and target platforms, and you do not use any of the preceding options for SIL simulation, there are likely to be differences between host computer results and target execution results. When you use portable word sizes for SIL simulation, subtle differences in host and target processor behavior can still occasionally cause host simulation results to differ from target execution results. For more information, see "Portable Word Sizes Limitations" on page 39-51.

To configure a model to use portable word sizes, set the following model configuration parameters.

Set...	To...
Hardware Implementation > Emulation hardware > None	Selected
Code Generation > SIL and PIL Verification > Create block	SIL
Code Generation > SIL and PIL Verification > Enable portable word sizes	Selected

When you generate code for a model with the preceding parameter settings, the code generator conditionalizes data type definitions:

- `tmwtypes.h` supports SIL simulation on the host system
- Code generation types support deployment on the target system

For example, in the following generated code, the first two lines define types for SIL simulation on a host system. The **bold** lines define types for target deployment.

```
#ifdef PORTABLE_WORDSIZES    /* PORTABLE_WORDSIZES defined */
# include "tmwtypes.h"
#else                        /* PORTABLE_WORDSIZES not defined */
#define __TMWTYPES__
#include <limits.h>
...
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef int int16_T;
typedef unsigned int uint16_T;
typedef long int32_T;
typedef unsigned long uint32_T;
typedef float real32_T;
typedef double real64_T;
...
#endif                       /* PORTABLE_WORDSIZES */
```

For an example of how to configure a model to maintain bit-true agreement between host simulation and target deployment, and generate code that is portable between the host and target systems, see `rtwdemo_sil_hardware_config`.

Compiling Generated Code That Supports Portable Word Sizes

When you compile generated code that supports portable word sizes for SIL testing, pass the definition `PORTABLE_WORDSIZES` to the compiler.

For example:

```
-DPORTABLE_WORDSIZES
```

To build the same code for target deployment, compile the code without the `PORTABLE_WORDSIZES` definition.

Portable Word Sizes Limitations

The following limitations apply to using portable word sizes for SIL simulation:

- Numerical results might differ between generated code executing in a SIL simulation versus executing on the embedded hardware under one of the following conditions:
  - Your model contains blocks implemented in TLC, for which C integral promotion in expressions might behave differently between the MATLAB host and the embedded hardware target. Normal and PIL simulation results will match, but SIL simulation results might be different.
  - Your embedded hardware implements rounding to `Floor` for signed integer division, and divisions in your model use rounding mode `Floor` or `Simplest`. Normal and PIL simulation results will match, but SIL simulation results might be different.
  - The precision of floating-point operations differs between the MATLAB host and the embedded hardware target. In this case, Normal and SIL simulation results will match, but PIL simulation results might be different.

- Compilation warnings might occur for code generated using portable word sizes if all of the following conditions exist:
  - The combination of MATLAB host and embedded hardware target word sizes causes `rtwtypes.h` to redefine the word sizes using preprocessor macros. For example, when the embedded hardware has a 16-bit `int` data type and the MATLAB host has a 16-bit `short` data type, `int16_T` is redefined to be `short` on the host and `int` on the target.
  - The data types are used in pointer arguments to function calls.
  - The called functions are host-based precompiled functions (not compiled using `rtwtypes.h`).

  Under these conditions, the compiler typically issues a warning similar to the following:

  ```
  warning: passing argument 2 of 'frexp' from incompatible pointer type
  ```

  Executing the generated code on the MATLAB host could lead to memory corruption. For example, the function "`double frexp (double value, int *exp);`" expects `'int *'` as the second argument, for which `'int16_T *'` is passed in the generated code. But on the MATLAB host, `int16_T` is redefined to `short`, and during SIL execution, `frexp` will attempt to write 4 bytes to a 2 byte location.

  A potential workaround for the SIL workflow is to provide a custom Target Function Library (TFL) entry for functions that write to address locations obtained through pointer arguments. In the above example, the function `frexp` is called by the reciprocal square root operation (`rSqrt`) and `rSqrt` is replaceable using TFLs. Therefore, you can provide a custom version of `rSqrt` to support SIL execution. The replacement function would perform the necessary change in memory allocation for the data accessed by the pointer variable, perhaps by introducing a temporary variable and transferring the data to and from that variable. For more information about TFLs, see Chapter 31, "Replacing Math Functions and Operators Using Target Function Libraries".

Programming PIL Support for Third-Party Tools and Target Hardware

You can use the Processor-in-the-loop (PIL) Connectivity API to apply the power of PIL verification to object code compiled for your target processor. There are many custom or third-party tools for building, downloading, and communicating with an executable on a target environment. Use the API to integrate your tools for:

- Building the PIL application (an executable for the target hardware)
- Downloading and running the executable
- Communicating with the executable

You can use PIL with any target hardware or instruction set simulator, and any combination of tools that provide the required level of automation. For hardware cases that MathWorks does not support, see "SIL and PIL Simulation Support and Limitations" on page 39-60.

For instructions and demos on PIL and the Target Connectivity API, see:

- "Configuring a SIL or PIL Simulation" on page 39-16
- "Creating a Connectivity Configuration for a Target" on page 39-54
- "Demos of the Target Connectivity API" on page 39-59

Creating a Connectivity Configuration for a Target

In this section...

"What Is a PIL Connectivity Configuration?" on page 39-54
"Overview of the Target Connectivity API" on page 39-55
"Creating a Connectivity API Implementation" on page 39-58
"Registering a Connectivity API Implementation" on page 39-58
"Demos of the Target Connectivity API" on page 39-59

What Is a PIL Connectivity Configuration?

You can use PIL connectivity configurations and the target connectivity API to customize PIL to work with any target environment.

Use a connectivity configuration to define:

- A configuration name
- A connectivity API implementation
- Settings that define the set of Simulink models that the configuration is compatible with, for example, the set of models that have a particular system target file, template makefile, and hardware implementation.

You can use the API to integrate third party tools for:

- Building the PIL application, an executable for the target hardware
- Downloading and running the executable
- Communicating with the executable

A particular connectivity configuration name is associated with a single connectivity API implementation. Many different connectivity configurations can coexist and be available for use with PIL simulations. You register each connectivity configuration to Simulink by creating an `sl_customization.m` file and placing it on the MATLAB path.

To run a PIL simulation, the software must first determine which of the available connectivity configurations to use. The software looks for a connectivity configuration that is compatible with the model under test. If the software finds multiple or no compatible connectivity configurations, you see an error message describing how to resolve the problem.

For information on how you create a connectivity configuration for a target, see:

1 "Overview of the Target Connectivity API" on page 39-55

2 "Creating a Connectivity API Implementation" on page 39-58

3 "Registering a Connectivity API Implementation" on page 39-58

See also Example Custom Targets for information about running PIL simulations on specific targets.

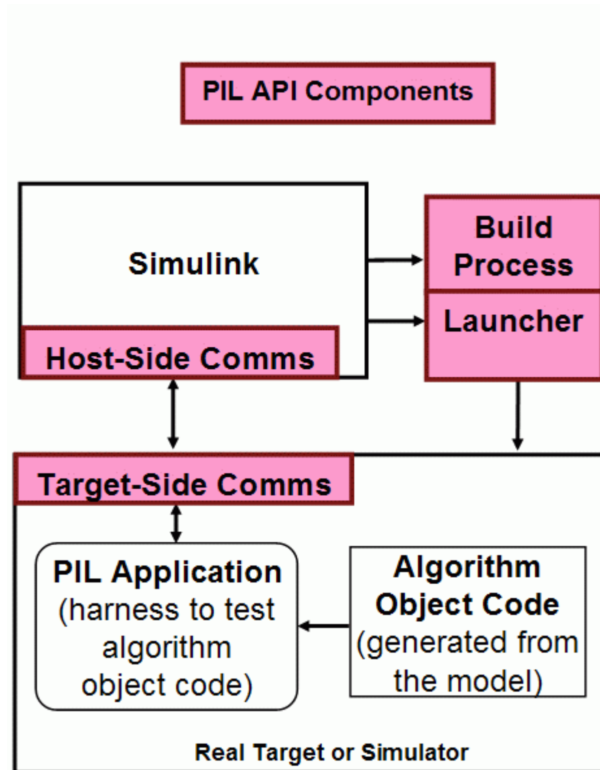
Overview of the Target Connectivity API

- "Target Connectivity API Components" on page 39-55
- "Communications rtiostream API" on page 39-56

Target Connectivity API Components

The following diagram shows what functions the Target Connectivity API components perform:

- Configuring the build process
- Controlling communication between Simulink and the target
- Downloading, starting, and stopping the application on the target



Communications rtiostream API

The communications part of the target connectivity API builds upon the `rtiostream` API, described in this section.

You can use the `rtiostream` API to implement a communication channel to enable exchange of data between different processes. This communication channel is required to enable processor-in-the-loop (PIL) on a new target.

PIL requires a host-target communications channel. This communications channel comprises driver code that runs on the host and target. The `rtiostream` API defines the signature of both target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer that sends the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN).

A full `rtiostream` implementation requires both host-side and target-side drivers. Code generation software includes host-side drivers for the default TCP/IP implementation (all platforms) as well as a Windows only version for serial communications. To use the TCP/IP `rtiostream` communications channel, you must provide, or obtain from a third party, target-specific TCP/IP device drivers. You must also do this if you require serial communications. For other communication channels and platforms, there is no default implementation provided by the code generation software. You must provide both the host-side and the target-side drivers.

The `rtiostream` API comprises the following functions:

- `rtIOStreamOpen`
- `rtIOStreamSend`
- `rtIOStreamRecv`
- `rtIOStreamClose`

You can use `rtiostream_wrapper` to test the `rtiostream` shared library methods from MATLAB code.

To see how the `rtiostream` functions fit into the workflow of creating a connectivity implementation, see the next section, "Creating a Connectivity API Implementation" on page 39-58.

Creating a Connectivity API Implementation

To create a target connectivity API implementation, you must create a subclass of `rtw.connectivity.Config`.

- You must instantiate `rtw.connectivity.MakefileBuilder`. This class configures the build process.
- You must create a subclass of `rtw.connectivity.Launcher`. This class downloads and executes the application using a third-party tool.
- Configure your `rtiostream` communications implementation:
  - On the target-side, integrate the driver code implementing `rtiostream` functions directly into the build process by creating a subclass of `rtw.pil.RtIOStreamApplicationFramework`.
  - On the host-side, compile the driver code into a shared library. You load and initialize this shared library by instantiating (or optionally, customizing) `rtw.connectivity.RtIOStreamHostCommunicator`.
- If you want to carry out code execution profiling *and* your target does not have built-in timer support, you must create a subclass of `rtw.connectivity.Timer` to generate a timer object. This timer object provides details of the hardware-specific timer and any associated source files.

See also:

- "Creating Subclasses — Syntax and Techniques" in MATLAB documentation.
- "Processor-in-the-Loop" for all classes, methods, and functions in the Target Connectivity API
- rtwdemo_custom_pil for a demo that helps you to create a target connectivity configuration using the Target Connectivity API

Registering a Connectivity API Implementation

Register the new connectivity API implementation to Simulink as a connectivity configuration, by creating or adding to an `sl_customization.m` file. By doing this, you also define the set of Simulink models that the new connectivity configuration is compatible with.

For details, see `rtw.connectivity.ConfigRegistry`.

Demos of the Target Connectivity API

For step-by-step examples, see the following demos:

- `rtwdemo_custom_pil`

  This demo shows you how to create a custom PIL implementation using the target connectivity APIs. You can examine the code that configures the build process to support PIL, a tool to use for downloading and execution, and a communication channel between host and target. Follow the steps in the demo to activate a full host-based PIL configuration.

- `rtwdemo_rtiostream`

  This demo shows you how to implement a communication channel for use with the Embedded Coder product and your embedded target. This communication channel enables exchange of data between different processes. PIL simulation requires this because it requires exchange of data between the Simulink software running on your host computer and deployed code executing on target hardware.

  The rtiostream interface provides a generic communication channel that you can implement in the form of target connectivity drivers for a range of connection types. The demo shows how to configure your own target-side driver for TCP/IP, to operate with the default host-side TCP/IP driver. The default TCP/IP communications allow high bandwidth communication between host and target, suitable for transferring data such as video.

  The demo also shows how to implement custom target connectivity drivers, for example, using serial, CAN, or USB for both host and target sides of the communication channel.

SIL and PIL Simulation Support and Limitations

In this section...

"About SIL and PIL Simulation Support and Limitations" on page 39-61
"Code Source Support" on page 39-62
"Block Support" on page 39-65
"Configuration Parameters Support" on page 39-67
"I/O Support" on page 39-71
"Hardware Implementation Support" on page 39-84
"Other Feature Support" on page 39-88

About SIL and PIL Simulation Support and Limitations

Top-model and Model block software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulation modes, and SIL and PIL blocks are Embedded Coder features.

The following tables summarize the support provided for top-model SIL and PIL, Model block SIL and PIL and the SIL or PIL block. "Yes" indicates a supported feature.

Information on selected aspects of SIL and PIL is also provided, especially unsupported features and limitations.

Code Source Support

Code Source	Code Interface	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Top model	Standalone	Yes	No	Yes	Yes
Atomic subsystem	Standalone	No	No	Yes	Yes
Virtual subsystem	Standalone	No	No	Yes, but recommend atomic subsystem. See "Algebraic Loop Issues" on page 39-69	Yes, but recommend atomic subsystem. See "Algebraic Loop Issues" on page 39-69.
Model block	Model reference target	No, but you can include Model blocks inside your top model.	Yes. See "Cannot Use Multirate Model Block SIL/PIL Inside Conditionally Executed Subsystem" on page 39-64	No, but you can include Model blocks inside your model.	No, but you can include Model blocks inside your model.
Enabled/ Triggered subsystem	Standalone	No	No	Yes	Yes
Export Functions subsystem	Export Functions	N/A	N/A	Yes	Yes. See "PIL Block Export Functions" on page 39-64.

Code Source	Code Interface	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Legacy code	Custom	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.
MATLAB Coder	MATLAB Coder	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.	See "Custom Code Interfaces" on page 39-64.

For more information on code interfaces, see "SIL and PIL Code Interfaces" on page 39-47.

Custom Code Interfaces

MathWorks does not provide direct SIL/PIL support for code interfaces such as legacy code and MATLAB Coder. However, you can incorporate these interfaces into Simulink as an S-function (for example, using the Legacy Code Tool, S-Function Builder, or handwritten code), and then verify them using SIL/PIL.

SIL/PIL Does Not Check Simulink Coder Error Status

SIL/PIL does not check the Simulink Coder error status of the generated code under test. This error status flags exceptional conditions during execution of the generated code.

The Simulink Coder error status can also be set by blocks in the model (for example, custom blocks developed by a user). It is a limitation that SIL/PIL cannot check this error status and report back errors.

Cannot Use Multirate Model Block SIL/PIL Inside Conditionally Executed Subsystem

You see an error if you place your Model block (in either SIL or PIL simulation mode) in a conditionally executed subsystem and the referenced model is multirate (that is, has multiple sample times). Single rate referenced models (with only a single sample time) are not affected.

PIL Block Export Functions

The PIL block does not support the export of functions from triggered subsystems. With the PIL block, you can export only function-call subsystems.

Block Support

Blocks	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Model block	Yes, you can include Model blocks inside your top model.	Yes	Yes, you can include Model blocks inside your subsystem or model.	Yes, you can include Model blocks inside your subsystem or model.
DSP System Toolbox	Yes	Yes	Yes	Yes
Computer Vision System Toolbox™	Yes	Yes	Yes	Yes
MATLAB Function block	Yes	Yes	Yes	Yes
Driver blocks	Yes, but not recommended.	Yes, but not recommended.	Yes, but not recommended.	Yes, but not recommended.
To File blocks	Yes, if MAT-file logging is on. MAT-file logging may not be available in PIL mode.	No. MAT-file logging is not supported.	Yes, if MAT-file logging is on.	Yes, if MAT-file logging is supported and on.
To Workspace blocks	Yes, if MAT-file logging is on. MAT-file logging may not be available in PIL mode.	No, MAT-file logging is not supported.	Yes, if MAT-file logging is on.	Yes, if MAT-file logging is supported and on.

Blocks	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Merge blocks	Yes	Yes. Cannot connect SIL/PIL outputs to Merge blocks. See "Merge Block Issue" on page 39-66.	Yes. Cannot connect SIL outputs to Merge blocks. See "Merge Block Issue" on page 39-66.	Yes. Cannot connect PIL outputs to Merge blocks. See "Merge Block Issue" on page 39-66.
Stop block	No. SIL/PIL ignores the Stop Simulation block and continues simulating.	No. SIL/PIL ignores the Stop Simulation block and continues simulating.	No. SIL ignores the Stop Simulation block and continues simulating.	No. PIL ignores the Stop Simulation block and continues simulating.
Scope blocks, and all types of run-time display For example, display of port values and signal values	No	No	No	No

Merge Block Issue

If you connect SIL/PIL outputs to a Merge block, you see an error because S-function memory is not reusable.

Other Top-Model SIL/PIL Limitations

SIL/PIL does not support the callbacks (model or block) `StartFcn` and `StopFcn`.

Note Top-model SIL/PIL supports the callback `InitFcn`.

Configuration Parameters Support

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
ERT-based system target file	Yes	Yes	Yes	Yes
AUTOSAR system target file	Yes. See "AUTOSAR Top Model SIL and PIL Support" on page 24-65.	Yes. See "AUTOSAR Model Block SIL and PIL Support" on page 24-66.	Yes. See "AUTOSAR SIL and PIL Block Support" on page 24-67.	Yes. See "AUTOSAR SIL and PIL Block Support" on page 24-67.
GRT-based system target file	No	No	No	No
GRT compatible call interface	No; see "Missing Code Interface Description File Errors" on page 39-68.	No; see "Missing Code Interface Description File Errors" on page 39-68.	No	No; see "Missing Code Interface Description File Errors" on page 39-68.
Function Prototype Control	Yes	Yes	Yes	Yes
Reusable code format	Yes, but see the special cases in "Imported Data Definitions" on page 39-77.	N/A	Yes	Yes, but see the special cases in "Imported Data Definitions" on page 39-77.
Target Function Library	Yes	Yes	Yes	Yes

Configuration Parameters	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
C++	No; see “Missing Code Interface Description File Errors” on page 39-68.	No; see “Missing Code Interface Description File Errors” on page 39-68.	Yes	No; see “Missing Code Interface Description File Errors” on page 39-68.
Generate ASAP2 file	Yes	Yes	Yes	Yes
Generate example main	N/A	N/A	N/A	N/A
MAT-file logging	Yes. For PIL, the target	No	Yes	Yes, if the target
Signal logging	Yes, but only for signals	No, but see “Verifying Internal Signals of a	No, but see “Verifying	No, but see “Verifying
’Simplified’ model initialization	Yes	Yes	Yes	Yes
Single output/update	Yes, but see “Algebraic Loop Issues” on page 39-69.	Yes, but see “Algebraic Loop Issues” on page 39-69.	Yes, but see “Algebraic Loop Issues” on page 39-69.	Yes, but see “Algebraic Loop Issues” on page 39-69.
Configuration set reference	Yes	Yes	Yes	Yes

- “Missing Code Interface Description File Errors” on page 39-68
- “Algebraic Loop Issues” on page 39-69

Missing Code Interface Description File Errors

SIL/PIL requires a code interface description file, which is generated during the code generation process for the component under test. If the code interface description file is missing, the SIL/PIL simulation cannot proceed and you see an error reporting that the file does not exist. This error can occur if you select these unsupported options in your configuration parameters:

- **GRT compatible call interface**

- **Target Language** option **C++ encapsulated**

Do not select these options.

Algebraic Loop Issues

For more information on algebraic loops, see:

- "Algebraic Loops" in the Simulink documentation.
- The Algebraic Loops section in "Simulation Considerations That Affect Code Generation" in the Simulink Coder documentation.
- The Introduction section in "Creating Subsystems" in the Simulink Coder documentation.

There are three ways that PIL simulation can introduce algebraic loops that do not exist for a normal simulation:

- "Algebraic Loops Caused by Code Generation for a Virtual Subsystem" on page 39-69
- "Algebraic Loops Caused by "Single output/update function"" on page 39-69
- "Algebraic Loops Caused by SIL/PIL Scheduling Limitations" on page 39-70

Algebraic Loops Caused by Code Generation for a Virtual Subsystem. If you generate code for a virtual subsystem, code generation treats the subsystem as atomic and generates the code accordingly. The resulting code can change the execution behavior of your model, for example, by applying algebraic loops, and introduce inconsistencies to the simulation behavior.

Declare virtual subsystems as atomic subsystems to ensure consistent simulation and execution behavior for your model.

See "Creating Subsystems" in the Simulink Coder documentation.

Algebraic Loops Caused by "Single output/update function". The "single output/update function" in code generation optimization can introduce algebraic loops because it introduces direct feedthrough via a combined output and update function.

This option is not compatible with the **Minimize algebraic loop occurrences** option (in the Subsystem Parameters dialog box and **Model Referencing** pane of the Configuration Parameters dialog box). This option allows code generation to remove algebraic loops by partitioning generated code appropriately between output and update functions to avoid direct feedthrough.

Algebraic Loops Caused by SIL/PIL Scheduling Limitations. The S-function scheduling mechanism that the software uses to execute the SIL/PIL component has the following limitations:

- Direct feedthrough is always set to true.
- Separate output and update functions in the SIL/PIL component are always executed from the `mdlOutputs` S-function callback.

These limitations mean that SIL/PIL can introduce algebraic loops that do not exist in normal simulation, and you might get incorrect results. If this happens, you see a warning or error about the introduced algebraic loop and SIL/PIL results may differ from simulation results. You do not see or warning or error if the algebraic loop setting is “none” in the Configuration Parameters dialog box (under Diagnostics on the Solver pane).

A workaround is to break the algebraic loop by inserting a Unit Delay block so that the algebraic loop does not occur. You can then use SIL/PIL successfully.

I/O Support

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Tunable parameters (Model reference arguments)	N/A	Yes. See "Tunable Parameters and SIL/PIL" on page 39-75.	N/A	N/A
Tunable parameters (Workspace variables)	No	Yes. See "Tunable Parameters and SIL/PIL" on page 39-75.	Yes	Yes. See "Tunable Parameters and SIL/PIL" on page 39-75.
Virtual buses	No	Yes	Yes	Yes, but some limitations at PIL component boundary; see "PIL Block Virtual Bus Support Limitations" on page 39-82.
Nonvirtual buses	Yes, but see "Top-Model SIL/PIL Bus Limitations" on page 39-81.	Yes	Yes	Yes
MUX/DEMUX	No	Yes	Yes	Yes, but see "PIL Block MUX Support Limitations" on page 39-82.
Vector/2D/ Multidimensional	Yes	Yes	Yes	Yes

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Complex data	Yes	Yes	Yes	Yes
Fixed-point data	Yes	Yes	Yes	Yes
Complex fixed-point data	Yes	Yes	Yes	Yes
Fixed-point data type override	Not at SIL or PIL component boundary. See "Fixed-Point Tool Data Type Override" on page 39-80	Not at SIL or PIL component boundary. See "Fixed-Point Tool Data Type Override" on page 39-80.	Yes	Not at PIL component boundary. See "Fixed-Point Tool Data Type Override" on page 39-80.
Data type replacement	Yes, but see "Data Type Replacement Limitation" on page 39-81	Yes, but see "Data Type Replacement Limitation" on page 39-81	Yes	Yes, but see "Data Type Replacement Limitation" on page 39-81
Goto/From I/O	N/A	N/A	Yes	Goto / From blocks must not cross the PIL component boundary. You can use Goto / From blocks to route buried signals up to top-level Inports and Outports *inside* the PIL component.

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Global data store I/O	Yes. See "Global Data Store Support" on page 39-76 and "Imported Data Definitions" on page 39-77.	Yes. See "Global Data Store Support" on page 39-76 and "Imported Data Definitions" on page 39-77.	Yes. See "Global Data Store Support" on page 39-76.	Yes. See "Global Data Store Support" on page 39-76 and "Imported Data Definitions" on page 39-77.
Local data store I/O	No. See "Imported Data Definitions" on page 39-77.	No. See "Imported Data Definitions" on page 39-77.	Yes	No. See "Imported Data Definitions" on page 39-77.
Non-port-based sample times	Yes	Yes	Yes	Yes
Continuous sample times	Not at SIL or PIL component boundary.	No	No	Not at PIL component boundary.
Outputs with constant sample time	Yes	No	Yes	Yes
Non-auto-storage classes for data (such as signals, parameters, data stores)	Yes. See "Imported Data Definitions" on page 39-77.	Yes. See "Imported Data Definitions" on page 39-77.	Yes	Yes. See "Imported Data Definitions" on page 39-77.
Simulink data objects	Yes	Yes	Yes	Yes
Simulink numeric type and alias type	Yes	Yes	Yes	Yes

I/O	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Simulink enumerated data	Yes	Yes	Yes	Yes
Custom storage classes	Yes, but see “Imported Data Definitions” on page 39-77, and “Unsupported Custom Storage Classes” on page 39-78.	Yes, but see “Imported Data Definitions” on page 39-77, and “Unsupported Custom Storage Classes” on page 39-78.	Yes	Yes, but see “Imported Data Definitions” on page 39-77, and “Unsupported Custom Storage Classes” on page 39-78.
Variable-size signals	No. See “Variable-Size Signals and SIL/PIL” on page 39-80.	No. See “Variable-Size Signals and SIL/PIL” on page 39-80.	Yes	No. See “Variable-Size Signals and SIL/PIL” on page 39-80.
Noninlined S-functions	Yes	No	Yes	Yes

- “Tunable Parameters and SIL/PIL” on page 39-75
- “Global Data Store Support” on page 39-76
- “Imported Data Definitions” on page 39-77
- “Unsupported Custom Storage Classes” on page 39-78
- “Unsupported Implementation Errors” on page 39-78
- “Variable-Size Signals and SIL/PIL” on page 39-80
- “Fixed-Point Tool Data Type Override” on page 39-80
- “Data Type Overrides Unavailable for Most Blocks in Embedded Targets and Desktop Targets” on page 39-81

- “Data Type Replacement Limitation” on page 39-81
- “Top-Model SIL/PIL Bus Limitations” on page 39-81
- “PIL Block Virtual Bus Support Limitations” on page 39-82
- “PIL Block MUX Support Limitations” on page 39-82
- “Incremental Build for Top-Model SIL/PIL” on page 39-82
- “Top-Model SIL/PIL Logging Limitations” on page 39-83
- “Exported Functions in Feedback Loops” on page 39-83

Tunable Parameters and SIL/PIL

You can tune parameters during a SIL/PIL mode simulation the same way that you tune parameters during a Normal mode simulation.

For more information, see “Global Tunable Parameters” and “Using Model Arguments” in the Simulink documentation.

Limitations. During a PIL block simulation, the software supports the tuning of *tunable workspace* parameters but not *tunable block dialog* parameters.

During a SIL/PIL simulation, the software cannot define, initialize, or tune the following types of tunable workspace parameters and produces warnings or errors.

Tunable Workspace Parameters Not Supported	Software response for ...		
	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Parameters with storage class that applies `"static"` scope or `"const"` keyword. For example, `Custom`, `Const`, or `ConstVolatile`	Warning	Warning	Warning

Tunable Workspace Parameters Not Supported	Software response for ...		
	Top-Model SIL/PIL	Model Block SIL/PIL	PIL Block
Fixed-point parameters with data type size greater than 32 bits	Warning	Error	Warning
Parameters with data types that have different sizes on host and target	Warning	Error	Warning
Structure parameters with storage class `SimulinkGlobal`	Warning	*Supported, so no warning or error*	Warning

If you select the configuration parameter **Generate reusable code** but do not select **Inline parameters** and the model contains parameters, then top-Model SIL/PIL and the PIL block can produce errors . If these conditions apply, then the software produces an error similar to the following:

```
Parameter Dialog:InitialOutput in 'rtwdemo_sil_topmodel/CounterTypeA/count'
is part of the imported "rtP" structure in the generated code but cannot be
initialized by SIL or PIL. To avoid this error, make sure the parameter
corresponds to a tunable base workspace variable with a storage class such
as SimulinkGlobal and is supported for dynamic parameter initialization /
tuning with SIL/PIL.
```

Global Data Store Support

SIL/PIL supports global data stores. PIL components that access global data stores must be single rate. If your SIL/PIL component has multiple sample times and accesses global data stores, you see an error. To avoid the error, either remove accesses to global data stores or make the component single rate.

Imported Data Definitions

You can use signals, parameters, data stores, etc., that specify storage classes with imported data definitions.

Model Block SIL/PIL. The SIL/PIL application automatically defines storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Parameters. See Tunable Parameters and SIL/PIL Limitations.
- Global data stores

A limitation is that SIL/PIL does not define storage for other imported data storage. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the PIL application does not define imported data storage for data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores

Top-Model SIL/PIL and SIL/PIL Block. The top-model SIL/PIL or PIL block application automatically defines storage for imported data associated with:

- Signals at the root level of the component (on the I/O boundary)
- Global data stores
- Parameters. See Tunable Parameters and SIL/PIL Limitations.

A limitation is that SIL/PIL does not define storage for other imported data storage. You must define the storage through custom code included by the component under test or through the PIL `rtw.pil.RtIOStreamApplicationFramework` API. For example, the SIL/PIL application does not define imported data storage for data associated with:

- Internal signals (not on the I/O boundary)
- Local data stores

Unsupported Custom Storage Classes

SIL/PIL does not support the following non-addressable custom storage classes:

- `BitField` — The software generates a compilation error. For example:

```
pil_interface.c: In function 'pilGetUIOData':
pil_interface.c:103: error: expected expression before ')' token
pil_interface.c:104: error: cannot take address of bit-field 'In1'
pil_interface.c: In function 'pilGetYIOData':
pil_interface.c:226: error: expected expression before ')' token
pil_interface.c:227: error: cannot take address of bit-field 'Out1'
pil_interface.c:287: error: expected expression before ')' token
pil_interface.c:288: error: cannot take address of bit-field 'Out1'
pil_interface.c:348: error: expected expression before ')' token
pil_interface.c:349: error: cannot take address of bit-field 'Out1'
```

- `GetSet` — The software generates the following error:

```
An unsupported imported grouped custom storage class (e.g. a user-defined
custom storage class) has been detected. Signals and parameters with imported
grouped custom storage classes are not supported by SIL or PIL. Check the
storage classes of signals and parameters at the SIL or PIL component boundary."
```

SIL/PIL also does not support signals and parameters with imported grouped custom storage classes.

Unsupported Implementation Errors

If you use a data store, signal, or parameter implementation that SIL/PIL does not support, you may see errors like the following:

```
The following data interfaces have
implementations that are not supported by SIL or PIL.
```

`data interfaces` may be global data stores, `inports`, `outports` or `parameters`.

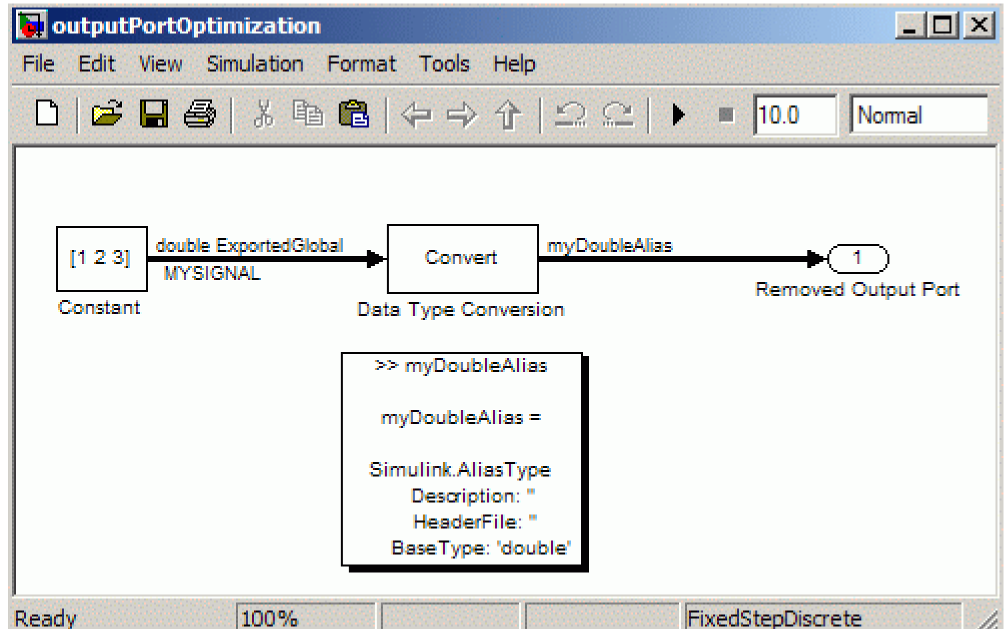
You see this error message because the model output port has been optimized through virtual output port optimization. See “Using Virtualized Output Ports Optimization” on page 21-24. The error occurs because the properties

(for example, data type, dimensions) of the signal or signals entering the virtual root output port have been modified by routing the signals in one of the following ways:

- Through a Mux block
- Through a block that changes the signal data type. To check the consistency of data types in the model, display Port Data Types by selecting **Format > Port/Signal Displays > Port Data Types**.
- Through a block that changes the signal dimensions. To check the consistency of data types in the model, display dimensions by selecting **Format > Port/Signal Displays > Signal Dimensions**.

Note Dimension changes from scalar (1) to matrix `[1x1]`, and, matrix `[1x1]` to scalar (1), can lead to this error. Furthermore, it is difficult to inspect the model for such changes because the **Format > Port/Signal Displays > Signal Dimensions** feature does not distinguish between (1) and `[1x1]` dimensions. The software shows both signals as scalar signals. Check your model and workspace objects carefully to ensure that scalar dimensions are specified consistently.

The following example illustrates a model that causes this error due to changing the output port signal data type.



Variable-Size Signals and SIL/PIL

SIL/PIL treats variable-size signals at the I/O boundary of the SIL/PIL component as fixed-size signals, which can lead to errors during propagation of signal sizes. To avoid such errors, use only fixed-size signals at the I/O boundary of the SIL/PIL component.

There may be cases where no error occurs during propagation of signal sizes. In these cases, the software treats variable-size input signals as zero-size signals.

Fixed-Point Tool Data Type Override

SIL/PIL does not support signals with data types overridden by the Fixed-Point Tool **Data type override** parameter at the SIL/PIL component boundary.

You may see an exception message like the following:

```
Simulink.DataType object 'real_T' is not in scope
```

```
from 'mpil_mtrig_no_ic_preread/TmpSFcnForModelReference_unitInTopMdl'.
 This error message is related to a hidden S-Function block.
```

There is no resolution for this issue.

Data Type Overrides Unavailable for Most Blocks in Embedded Targets and Desktop Targets

When you attempt to perform a datatype override on a block, you may get an error message similar to the following example:

```
Error reported by S-function 'sfun_can_frame_splitter' in
'c2000_host_CAN_monitor/CAN Message Unpacking/CAN Message
Unpacking':  Incompatible DataType or Size specified.
```

Data type overrides using the Fixed point tool are not available for those blocks in Simulink Coder > Desktop Targets and Embedded Coder > Embedded Targets libraries that support fixed-point.

There is no resolution for this issue.

Data Type Replacement Limitation

The software does not support replacement data type names that you define for the built-in data type `boolean` if these names map to either the `int` or `uint` built-in data type.

Top-Model SIL/PIL Bus Limitations

The software does not support grounded or unconnected signals at the outputs of a top model.

You must enable the strict bus mode for top-model SIL/PIL:

1. In the model window, select **Simulation > Configuration Parameters > Diagnostics > Connectivity**.

2. Set **Mux blocks used to create bus signals** to `error`.

PIL Block Virtual Bus Support Limitations

The PIL block supports virtual buses except for the following cases:

- You see an error if the PIL component is a top model with a root level outport that is configured to output a virtual bus. A root level outport outputs a virtual bus, regardless of the type of the bus that drives it, if it specifies a bus object and the **Output as nonvirtual bus in parent model** check box is not selected.
- You see an error if a right-click subsystem build expands the bus into individual signals.
- For right-click subsystem builds only, the PIL block changes the output of outports driven by virtual buses (with associated bus objects) into nonvirtual buses. You do not see an error message in this case.

To avoid these limitations, use nonvirtual buses at the PIL component boundary.

PIL Block MUX Support Limitations

The PIL block supports mux signals, except mixed data-type mux signals that expand into individual signals during a right-click subsystem build. You see an error for unsupported cases.

Incremental Build for Top-Model SIL/PIL

When you start a top-model SIL/PIL simulation, the software regenerates code if it detects changes to your model. The software detects changes by using a checksum for the model. However, the software does not detect changes that you make to:

- The `HeaderFile` property of a `Simulink.AliasType` object
- Legacy S-functions

Therefore, if you make these changes, you must build (**Ctrl-B**) your model again before starting the next PIL simulation.

Top-Model SIL/PIL Logging Limitations

Signal Logging. Top-model SIL/PIL supports signal logging for signals connected to root-level inports and outports with the following limitations.

- The characteristics of the logged data such as data type and dimensions match the characteristics of the root-level inports and outports rather than the characteristics of the connected signal.

  In some cases, there may be differences in data type and dimensions between the signal being logged and the root inport or outport that the signal is connected to. Consider the following examples.

  - If a signal being logged has matrix dimensions `[1x5]` but the outport connected to the signal has vector dimensions (`5`), then the data logged during a SIL or PIL simulation has vector dimensions (`5`).
  - If a signal being logged has scalar dimensions but the outport connected to the signal has matrix dimensions `[1x1]`, then the data logged during a SIL or PIL simulation has matrix dimensions `[1x1]`.

- The software adds the suffix `_wrapper` to the block path for signals in `logsout`. See "Running a Top Model as a SIL or PIL Simulation" on page 39-41.

Output Logging. If the save format is `Structure` or `Structure with time`, the software adds the suffix `_wrapper` to the block name for signals in `yout`. See "Running a Top Model as a SIL or PIL Simulation" on page 39-41.

Exported Functions in Feedback Loops

If your model has function-call subsystems and you export a subsystem that has context-dependent inputs (for example, feedback signals), then the results of a SIL/PIL simulation with the generated code may not match the results of the Normal mode simulation of your model. One approach to ensure that SIL/PIL and Normal mode simulations yield identical results is to use Function-Call Feedback Latch blocks in your model. This approach allows you to make context-dependent inputs become context-independent.

Note The software generates a warning identifying context-dependent inputs of exported function-call subsystems, regardless of your setting for the **Context-dependent inputs** field in the **Configuration Parameters > Diagnostics > Connectivity** pane. See also "Context-dependent inputs" in the Simulink Reference documentation.

Hardware Implementation Support

Hardware Implementation	Embedded Coder	Embedded Targets
Different host and target data-type size	Not at PIL component boundary. See "Hardware Implementation Settings" on page 39-84.	Not at PIL component boundary. See "Hardware Implementation Settings" on page 39-84.
Word-addressable targets	No	Yes
Multiword data type word order different to target byte order	No	Yes
Multiword	No	No
Size of target `'long'` > 32 bits	No	No

Hardware Implementation Settings

PIL requires that, in the Simulink Configuration Parameters dialog box, you correctly configure the Hardware Implementation settings for the target environment.

Specify byte ordering for non-8-bit targets.

For more information, see the following sections:

- "Host/Target Data Type Size Mismatch" on page 39-85
- "Data Type Size Mismatch Issues (Embedded Coder)" on page 39-85

- "Data Type Size Mismatch Issues (Embedded Targets)" on page 39-86

Host/Target Data Type Size Mismatch. PIL supports only data types that have the same size on the host and the target at the PIL I/O boundary.

The data types used at the PIL I/O boundary are restricted based on the following rule: PIL supports the data type only if the data-type size on the host (Simulink) is the same as the data-type size on the target.

- For `boolean`, `uint8`, and `int8`, the size is 8-bits.
- For `uint16` and `int16`, the size is 16-bits.
- For `uint32` and `int32`, the size is 32-bits.
- For `single`, the size is 32-bits.
- For `double`, the size is 64-bits.

Examples of unsupported data types are:

- `single` and `double` on targets with 24-bit floating-point types
- `double` on targets with 32-bit double, that is, the same size as `single`

Warning **PIL does not always detect unsupported data types (see "Data Type Size Mismatch Issues (Embedded Coder)" on page 39-85 and "Data Type Size Mismatch Issues (Embedded Targets)" on page 39-86). In such cases, data transfer between host and target is incorrect and unexpected data transfer errors occur during the PIL simulation.**

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Data Type Size Mismatch Issues (Embedded Coder). PIL mode makes the following assumptions about the target environment:

- The target is byte addressable.
- Sizes of data types on the host and target match.

- Word order of multiword data types on the target is the same as the target byte order.

Warning **PIL does not detect violations of these assumptions. If the settings for the target environment violate any of these assumptions, unexpected data transfer errors occur during the PIL simulation.**

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Some known violations with predefined hardware implementation settings are:

- Unsupported word addressable targets: TI's C2000™, Freescale™ DSP563xx
- Unsupported data types owing to word order: TASKING® Infineon® C166® (single and double have reversed word order)
- Unsupported data types owing to size mismatch: TASKING 8051 (double > is only 4 bytes)

Data Type Size Mismatch Issues (Embedded Targets). The embedded targets registers the following information about the target environment to Simulink:

- Whether the target is byte addressable.
- The sizes of data types on the target.
- The word order of multiword data types on the target.

This allows PIL to support target environments with unusual hardware characteristics.

PIL can detect data types used at the PIL component boundary that have a host/target data type size mismatch. In such cases, an error indicates that an unsupported data type is being used. This avoids unexpected data transfer errors during simulation.

To resolve issues with Simulink data types that have different sizes on the host and target, do not use them at the PIL I/O boundary. Instead, use a Simulink data type that maps directly onto a target data type. This resolution is more efficient.

Some target compilers allow the sizes of target data types to be changed from their default size. For example, an IEEE® double data type is most likely 8 bytes by default, but an optimization option may be provided to treat it as a 4 byte IEEE single precision type instead. The registration of target data type sizes done by the embedded targets features is typically statically defined to match the compiler's default data type size, and therefore does not support changing the data type size from that default size.

Warning **If you use a nondefault compiler configuration, the target data type sizes registered by the embedded targets features and the actual target data type sizes may differ. In such cases, data transfer between host and target is incorrect and unexpected data transfer errors occur during the PIL simulation.**

To resolve this issue, either:

- Do not use compiler options that change the default size of target data types.
- Use the single data type in Simulink rather than double, if your aim is to treat double-precision floating-point types (8 bytes) as single-precision floating-point types (4 bytes).

Other Feature Support

Other Features	Top-Model SIL/PIL	Model Block SIL/PIL	SIL Block	PIL Block
Multiplatform support (such as Linux)	Yes	Yes	Yes	Yes
Execution profiling	Yes	Yes	No	Yes
Stack profiling	SIL: No. PIL: Depends on target connectivity configuration and third-part product support.	SIL: No. PIL: Depends on target connectivity configuration and third-party product support.	No	Depends on target connectivity configuration and third-party product support.
C code coverage report	SIL: Yes, BullseyeCoverage. PIL: Yes, BullseyeCoverage (host file-system restriction). Also, other coverage metrics may be available depending on target connectivity configuration and third-party product support.	SIL: Yes, BullseyeCoverage. PIL: Yes, BullseyeCoverage (host file-system restriction). Also, other coverage metrics may be available depending on target connectivity configuration and third-party product support.	No	Depends on target connectivity configuration and third-party product support.

Verifying a Component in the Target Environment

- “About Component Verification in the Target Environment” on page 40-2
- “Goals of Component Verification in the Target Environment” on page 40-3
- “Maximizing Code Portability and Configurability” on page 40-4
- “Simplifying Code Integration and Maximizing Code Efficiency” on page 40-5
- “Running Component Tests in the Target Environment” on page 40-7

About Component Verification in the Target Environment

After you generate production code for a component design, you need to integrate, compile, link, and deploy the code as a complete application on the embedded system. One approach is to manually integrate the code into an existing software framework that consists of an operating system, device drivers, and support utilities. The algorithm can include externally written legacy or custom code.

An easier and more recommended approach to verifying a component in a target environment, is to use processor-in-the-loop (PIL) simulation. For details on applying PIL simulations, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations".

Goals of Component Verification in the Target Environment

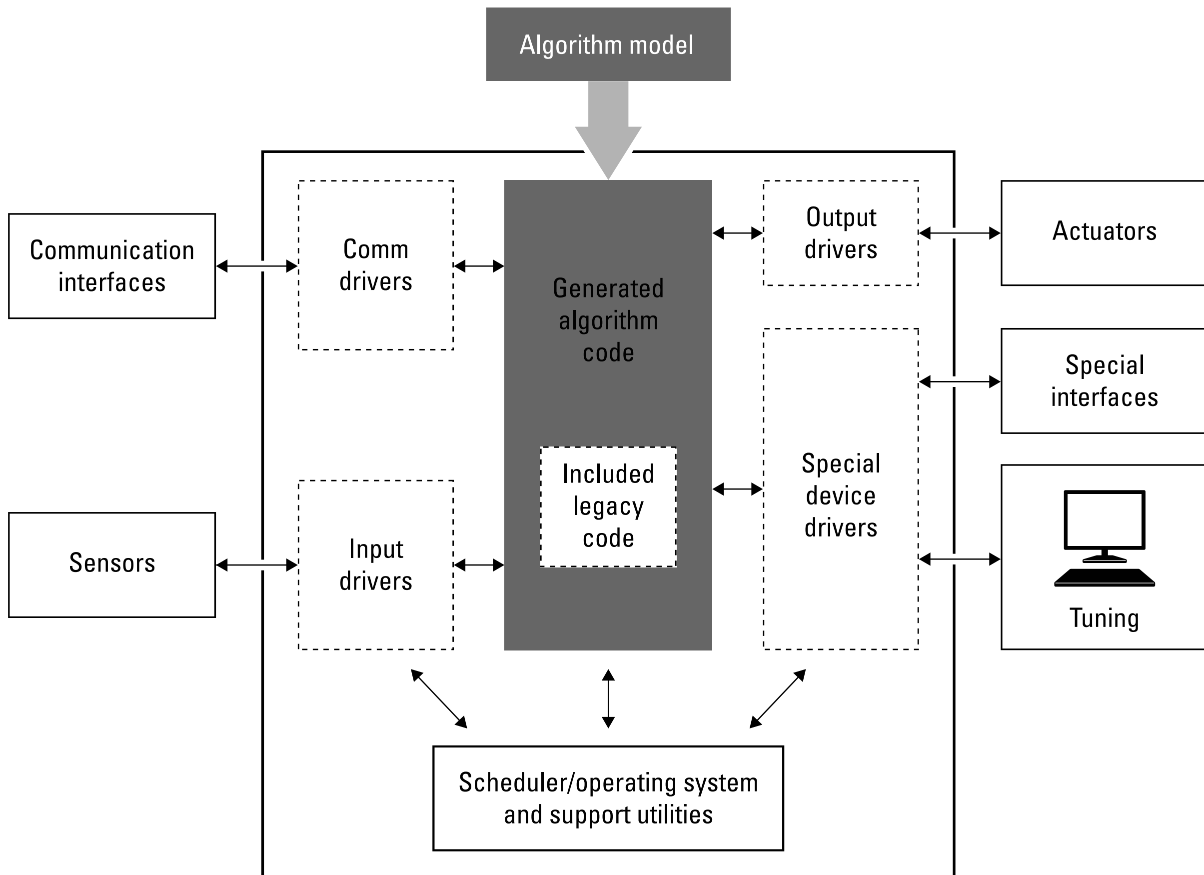
Assuming that you have generated production quality source code and integrated necessary externally written code, such as drivers and a scheduler, you can verify that the integrated software operates correctly by testing it in the target environment. During testing, you can achieve either of the following goals, depending on whether you export code that is strictly ANSI C/C++ or mixes ANSI C/C++ with code optimized for a target environment.

Goal	Type of Code Export
Maximize code portability and configurability	ANSI C/C++
Simplify integration and maximize use of processor resources and code efficiency	Mixed code

Regardless of your goal, you must integrate any required external driver and scheduling software. To achieve real-time execution, you must integrate the necessary real-time scheduling software.

Maximizing Code Portability and Configurability

To maximize code portability and configurability, limit the application code to ANSI/ISO C or C++ code only, as the following figure shows.

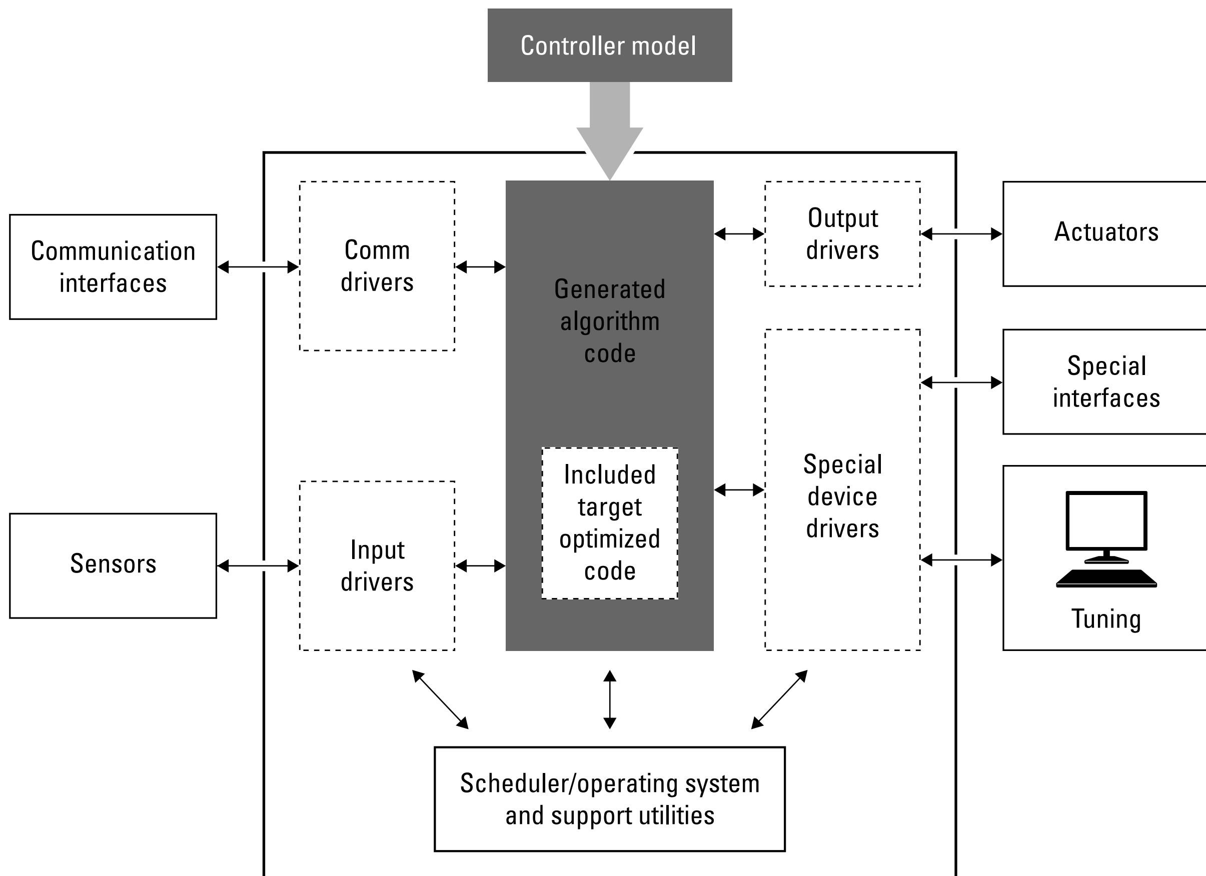


Simplifying Code Integration and Maximizing Code Efficiency

To simplify code integration and maximize code efficiency for a target environment, use Embedded Coder features for:

- Controlling code interfaces
- Exporting subsystems
- Including target-specific code, including compiler optimizations

The following figure shows a mix of ANSI C/C++ code with code that is optimized for a target environment.


Controller model
Communication interfaces
Comm drivers
Generated algorithm code
Output drivers
Actuators
Special interfaces
Included target optimized code
Special device drivers
Sensors
Input drivers
Tuning
Scheduler/operating system and support utilities

Running Component Tests in the Target Environment

The workflow for running software component tests in the target environment is:

1. Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see Integrating External Code and Generated C and C++ Code on page 1 and "Integrating External Code With Generated C and C++ Code" in the Simulink Coder documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	"Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" in the Simulink documentation. Also, open `rtwdemos` and navigate to the **Custom Code** folder.
Mixed code integration	• Chapter 27, "Exporting Function-Call Subsystems" and examplertwdemo_export_functions • Chapter 29, "Controlling Generation of Function Prototypes", Chapter 30, "Controlling Generation of Encapsulated C++ Model Interfaces", and example rtwdemo_fcnprotoctrl • Chapter 31, "Replacing Math Functions and Operators Using Target Function Libraries" and example `rtwdemo_tfl_script`

2. Simulate the integrated component model.

3. Generate code for the integrated component model.

4. Connect to data interfaces for the generated C code data structures. See "Interacting with Target Application Data Using the C API" and "Generating Model Information for Host-Based ASAP2 Data Measurement

and Calibration" in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.

5. Customize and control the build process, as necessary. See "Customizing Code Generation and the Build Process", in the Simulink Coder documentation, and example `rtwdemo_buildinfo` .

6. Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See "Relocating Code to Another Development Environment", in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

41

Verifying a Component by Building a Complete Real-Time Target Environment

- "About Component Verification With a Complete Real-Time Target Environment" on page 41-2
- "Goals of Component Verification With a Complete Real-Time Target Environment" on page 41-4
- "Testing a Component as Part of a Complete Real-Time Target Environment" on page 41-5

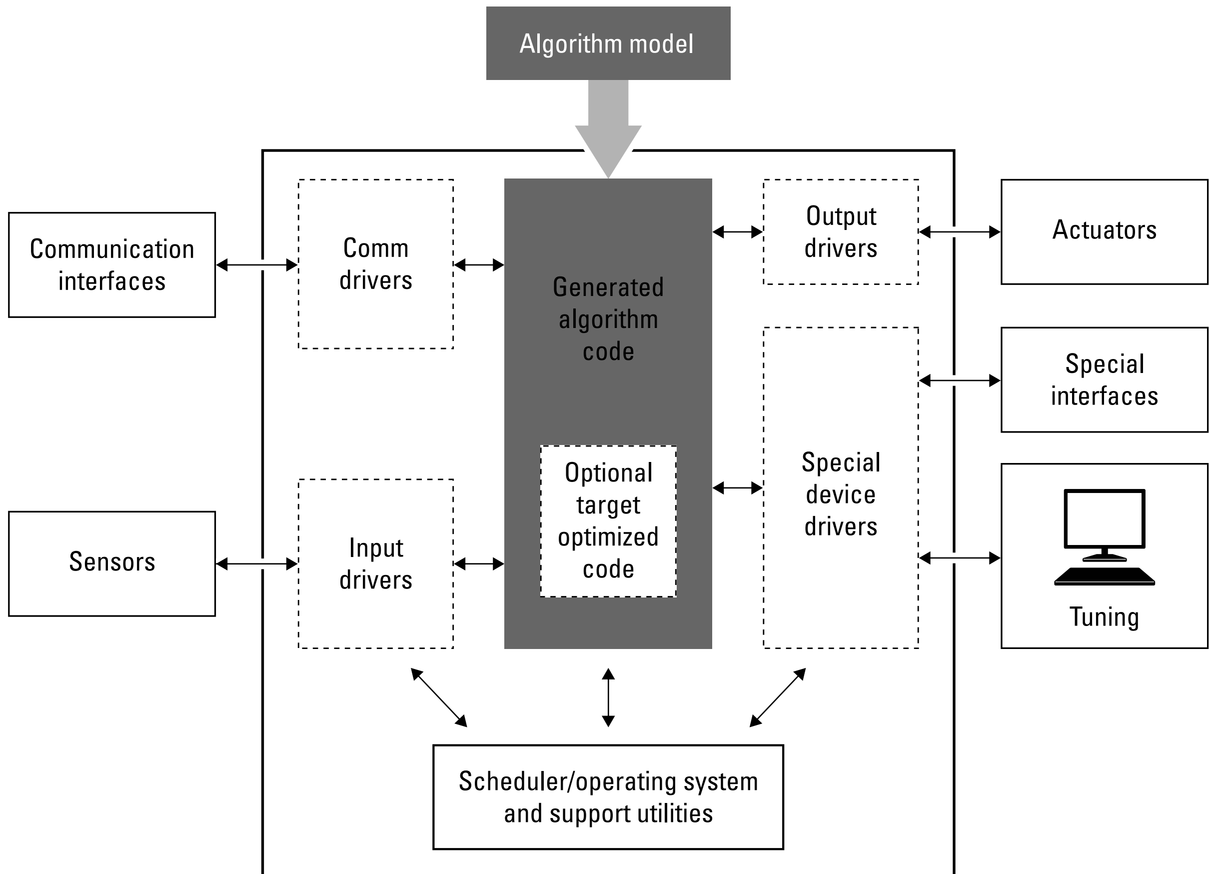
About Component Verification With a Complete Real-Time Target Environment

One approach to verifying a software component is to build the component into a complete software system that can execute in real time in the target environment. A complete software system includes:

- Algorithm for the software component
- Scheduling algorithms
- Calls to drivers for board-specific devices

This single build approach is more time consuming to set up, but makes it easier to get the complete application running in the target environment.

The following figure shows code generated for an algorithm being built into a complete system executable for the target environment.


Algorithm model
Communication interfaces
Comm drivers
Generated algorithm code
Optional target optimized code
Output drivers
Actuators
Special interfaces
Special device drivers
Sensors
Input drivers
Tuning
Scheduler/operating system and support utilities

Goals of Component Verification With a Complete Real-Time Target Environment

Assuming that you have generated production quality source code and integrated necessary externally written code, such as drivers and a scheduler, you can verify that component software operates correctly in the context of a complete system for testing in the target environment.

Testing a Component as Part of a Complete Real-Time Target Environment

The workflow for testing component software as part of a complete real-time target environment is:

1. Develop a component model and generate source code for production.

   For information on building in scheduling and real-time system support, see:

   - "Scheduling Considerations" in the Simulink Coder documentation. For an example, open `rtwdemos` and navigate to the **Multirate Support** folder.
   - "Handling Asynchronous Events" in the Simulink Coder documentation and example `rtwdemo_async`
   - "Interfacing With a Real-Time Operating System " in the Simulink Coder documentation and Chapter 35, "Wind River Systems VxWorks Example Main Program"
   - Chapter 34, "Interfacing With Hardware That is Not Running an Operating System (Bare Board)"
   - Chapter 24, "Generating Code for AUTOSAR Software Components" and examples rtwdemo_autosar_legacy_script, rtwdemo_autosar_mulitrunnables_script, and rtwdemo_autosar_clientserver_script
   - Example `rtwdemo_osek`

2. Optimize generated code for a specific run-time environment, using specialized function libraries. For more information, see Chapter 31, "Replacing Math Functions and Operators Using Target Function Libraries" and example `rtwdemo_tfl_script`.

3. Customize post code generation build processing to accommodate third-party tools and processes, as necessary. See "Customizing Code Generation and the Build Process" in the Simulink Coder documentation and example `rtwdemo_buildinfo`.

4 Integrate external code, for example, for device drivers and a scheduler, with the generated C or C++ code for your component model. For more information, see Integrating External Code and Generated C and C++ Code on page 1 and “Integrating External Code With Generated C and C++ Code” in the Simulink Coder documentation. For more specific references depending on your verification goals, see the following table.

For...	See...
ANSI C/C++ code integration	“Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” in the Simulink documentation. Also, open `rtwdemos` and navigate to the **Custom Code** folder.
Mixed code integration	• Chapter 27, “Exporting Function-Call Subsystems” and examplertwdemo_export_functions • Chapter 29, “Controlling Generation of Function Prototypes”, Chapter 30, “Controlling Generation of Encapsulated C++ Model Interfaces”, and example rtwdemo_fcnprotoctrl • Chapter 31, “Replacing Math Functions and Operators Using Target Function Libraries” and example `rtwdemo_tfl_script`

5 Simulate the integrated model.

6 Generate code for the integrated model.

7 Connect to data interfaces for the generated C code data structures. See “Interacting with Target Application Data Using the C API” and “Generating Model Information for Host-Based ASAP2 Data Measurement and Calibration” in the Simulink Coder documentation. Also see examples `rtwdemo_capi` and `rtwdemo_asap2`.

8. Customize and control the build process, as necessary. See “Customizing Code Generation and the Build Process”, in the Simulink Coder documentation, and example `rtwdemo_buildinfo` .

9. Create a zip file that contains generated code files, static files, and dependent data to build the generated code in an environment other than your host computer. See “Relocating Code to Another Development Environment”, in the Simulink Coder documentation, and example `rtwdemo_buildinfo`.

42

Verifying Numerical Equivalence of Results with Code Generation Verification API

Verifying Numerical Equivalence with Code Generation Verification

In this section...

"Code Generation Verification API Overview" on page 42-2

"Verifying Numerical Equivalence with CGV Workflow" on page 42-2

"Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model" on page 42-3

"Example of Plotting Output Signals" on page 42-10

Code Generation Verification API Overview

When you execute a model in different modes of execution, you can use the Code Generation Verification (CGV) API to verify the numerical equivalence of results. CGV supports executing the model in simulation, Software-In-the-Loop (SIL), and Processor-In-the-Loop (PIL). For more information about SIL and PIL, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations". The CGV demo, rtwdemo_cgv_script, shows CGV configuration, execution, and comparison support.

Note CGV helps you verify the numerical equivalence of results for a given set of inputs. CGV can detect numerical deviations for the given set of inputs only. CGV by itself cannot prove the numerical correctness of the generated code. The completeness of the input data that you provide to CGV determines the validity of the results.

Verifying Numerical Equivalence with CGV Workflow

Before verifying numerical equivalence:

- Configure your model for SIL or PIL simulation. For more information, see "Configuring a SIL or PIL Simulation" on page 39-16.
- Use the `cgv.Config` class of the CGV API to verify the model configuration for SIL or PIL simulation. For more information, see "Verifying a SIL or PIL Configuration" on page 39-22.

- Configure your model for code generation. For more information, see Preparing Models for Code Generation on page 1.
- Save your model. If you modify a model without saving it, CGV might issue an error.

To verify numerical equivalence:

- Set up the tests for the first execution environment. For example, simulation.
- Use cgv.CGV.run to run the tests for the first execution environment.
- Set up the tests for the second execution environment. For example, top-model PIL.
- Use `cgv.CGV.run` to run the tests for the second execution environment.
- Use cgv.CGV.getOutputData to get the output data for each execution environment.
- Use cgv.CGV.getSavedSignals to display the signal names in the output data. (optional)
- Build a list of signal names for input to other `cgv.CGV` methods. (optional)
- Use cgv.CGV.createToleranceFile to create a file correlating tolerance information with output signal names. (optional)
- Use cgv.CGV.compare to compare the output signals of the first and second execution environments for numerical equivalence.

Example of Verifying Numerical Equivalence Between Two Modes of Execution of a Model

The following example describes configuring, executing, and comparing the results of the rtwdemo_cgv model in simulation and SIL modes.

This example contains the following tasks:

- "Configuring the Model" on page 42-4
- "Executing the Model" on page 42-5
- "Comparing All Output Signals" on page 42-6

- “Comparing Individual Output Signals” on page 42-9

Configuring the Model

The first task for verifying numerical equivalence is to check that your model is configured correctly.

1. Open the `rtwdemo_cgv` model.

```
cgvModel = 'rtwdemo_cgv';
load_system(cgvModel);
```

2. Save the model to a working directory.

```
save_system(cgvModel, fullfile(pwd, cgvModel));
close_system(cgvModel); % avoid original model shadowing saved model
```

3. Use the cgv.Config to create a `cgv.Config` object. Specify parameters that check and modify configuration parameter values and save the model for top-model SIL mode of execution.

```
cgvCfg = cgv.Config('rtwdemo_cgv', 'connectivity', 'sil', 'SaveModel', 'on');
```

4. Use the cgv.Config.configModel method to review your model configuration and to change the settings to configure your model for SIL. When `'connectivity'` is set to `'sil'`, the system target file is automatically set to `'ert.tlc'`. If you specified the parameter/value pair, (`'SaveModel', 'on'`) when you created the `cgvCfg` object, the `cgv.Config.configModel` method saves the model.

Note CGV runs on models that are open. If you modify a model without saving it, CGV might issue an error.

```
cgvCfg.configModel(); % Evaluate, change, and save your model for SIL
```

5. Display a report of the changes that `cgv.Config.configModel` makes to the model.

```
cgvCfg.displayReport(); % In this example, this reports no changes
```

Executing the Model

Use the CGV API to execute the model in two modes. The two modes in this example are normal mode simulation and SIL mode. In each execution of the model, the CGV object for each mode captures the output data and writes the data to a file.

1 If you have not already done so, follow the steps described in “Configuring the Model” on page 42-4.

2 Create a `cgv.CGV` object that specifies the `rtwdemo_cgv` model in normal mode simulation.

```
cgvSim = cgv.CGV(cgvModel, 'connectivity', 'sim');
```

Note When the top model is set to Normal simulation mode, any referenced models set to PIL mode will be changed to Accelerator mode.

---

3 Provide the input file to the `cgvSim` object.

```
cgvSim.addInputData(1, [cgvModel '_data']);
```

4 Before execution of the model, specify the MATLAB files to execute or MAT-files to load. This step is optional.

```
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
```

5 Specify a location where the object writes all output data and metadata files for execution. This step is optional.

```
cgvSim.setOutputDir('cgv_output');
```

6 Execute the model.

```
result1 = cgvSim.run();

*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
 connectivity sim, InputData rtwdemo_cgv_data.mat
End CGV execution: status completed
```

7 Get the output data associated with the input data.

```
outputDataSim = cgvSim.getOutputData(1);
```

8 For the next mode of execution, SIL, repeat steps 2–7.

```
cgvSil = cgv.CGV( cgvModel, 'Connectivity', 'sil');
cgvSil.addInputData(1, [cgvModel '_data']);
cgvSil.addPostLoadFiles({[cgvModel '_init.m']});
cgvSil.setOutputDir('cgv_output');
result2 = cgvSil.run();
```

At the MATLAB command line, the result is:

```
*** handling PostLoad file rtwdemo_cgv_init.m
Start CGV execution of model rtwdemo_cgv, ComponentType topmodel, ...
     connectivity sil, InputData rtwdemo_cgv_data.mat

### Starting build procedure for model: rtwdemo_cgv
### Successful completion of build procedure for ...
    model: rtwdemo_cgv
### Preparing to start SIL simulation ...
### Starting SIL simulation for model: rtwdemo_cgv
### Stopping SIL simulation for model: rtwdemo_cgv
End CGV execution: status completed
```

Comparing All Output Signals

After setting up and running the test, compare the outputs by doing the following:

1 If you have not already done so, configure and test the model, as described in “Configuring the Model” on page 42-4 and “Executing the Model” on page 42-5.

2 Test that the execution of the model ran successfully:

```
if ~result1 || ~result2
    error('Execution of model failed.');
end
```

3 Use the cgv.CGV.getOutputData method to get the output data from the cgv.CGV objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

4 Display a list of signals by name using the cgv.CGV.getSavedSignals method.

```
cgvSim.getSavedSignals(simData);
```

At the MATLAB command line, the result it:

```
simData.hi0.Data(:,1)
simData.hi0.Data(:,2)
simData.Vector.Data(:,1)
simData.Vector.Data(:,2)
simData.Vector.Data(:,3)
simData.Vector.Data(:,4)
simData.BusOutputs.hi0.Data(:,1)
simData.BusOutputs.hi0.Data(:,2)
simData.BusOutputs.hi1.mid0.lo0.Data(1,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(1,2,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,1,:)
simData.BusOutputs.hi1.mid0.lo0.Data(2,2,:)
simData.BusOutputs.hi1.mid0.lo1.Data
simData.BusOutputs.hi1.mid0.lo2.Data
simData.BusOutputs.hi1.mid1.Data(:,1)
simData.BusOutputs.hi1.mid1.Data(:,2)
simData.ErrorsInjected.Data
```

5 Using the list of signals, build a list of signals in a cell array of strings. The signal list can contain any number of signals.

```
signalList = {'simData.ErrorsInjected.Data'};
```

6 Use the cgv.CGV.createToleranceFile method to create a file, in this example, `'localtol'`, correlating tolerance information with output signal names.

```
toleranceList = {{'absolute', 0.5}};
```

```
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
```

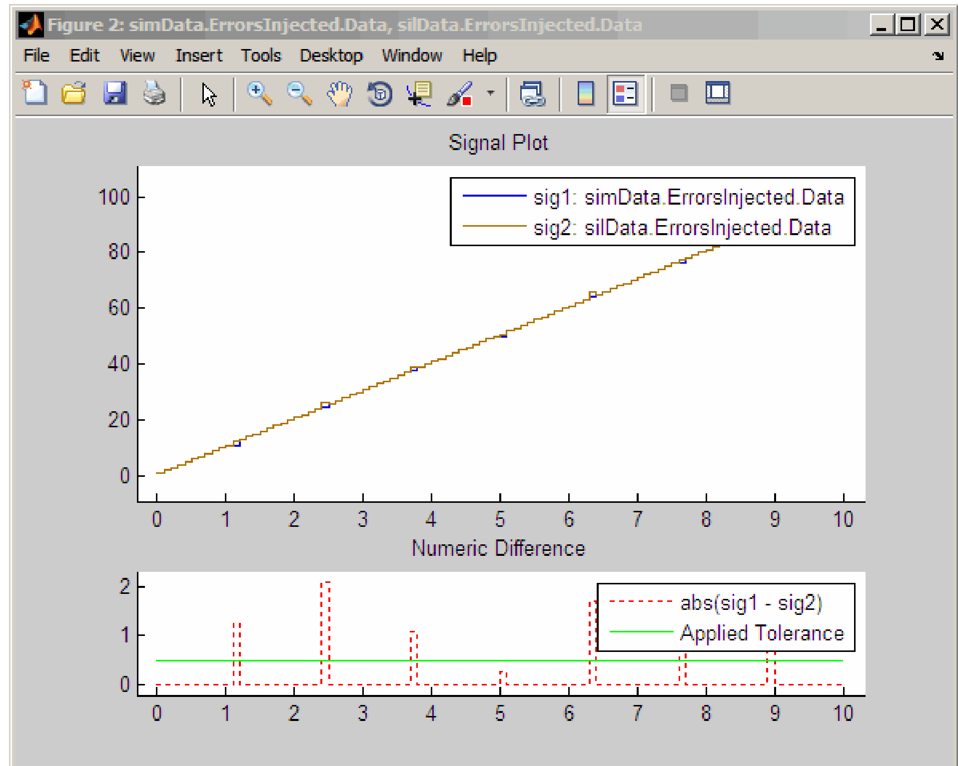
7 Compare the output data signals. By default, the cgv.CGV.compare method looks at all signals which have a common name between both executions. If a tolerance file is present, `cgv.CGV.compare` uses the associated tolerance for a specific signal during comparison; otherwise the tolerance is zero. In this example, the `'Plot'` parameter is set to `'mismatch'`. Therefore, only mismatched signals produce a plot.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'Tolerancefile', 'localtol');
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
disp('Mismatched Signal Names:');
disp(mismatchNames);
```

At the MATLAB command line, the result is:

```
14 Signals match, 1 Signals mismatch
Mismatched Signal Names:
    'simData.ErrorsInjected.Data'
```

A plot results from the mismatch on signal `simData.ErrorsInjected.Data`.



The lower plot displays the numeric difference between the results.

Comparing Individual Output Signals

After setting up and running the test, compare the outputs of individual signals by doing the following:

1 If you have not already done so, configure and test the model, as described in "Configuring the Model" on page 42-4 and "Executing the Model" on page 42-5.

2 Use the cgv.CGV.getOutputData method to get the output data from the `cgv.CGV` objects.

```
simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);
```

3 Use the cgv.CGV.getSavedSignals method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain any number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.BusOutputs.hi1.mid0.lo1.Data', ...
'simData.BusOutputs.hi1.mid0.lo2.Data', 'simData.Vector.Data(:,3)'};
```

4 Use the specified signals as input to the cgv.CGV.compare method to compare the signals from separate runs.

```
[matchNames, ~, mismatchNames, ~] = ...
    cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
    'signals', signalList);
fprintf( '%d Signals match, %d Signals mismatch\n', ...
    length(matchNames), length(mismatchNames));
if ~isempty(mismatchNames)
    disp( 'Mismatched Signal Names:');
    disp(mismatchNames);
end
```

At the MATLAB command line, the result is:

```
3 Signals match, 0 Signals mismatch
```

Example of Plotting Output Signals

After setting up and running the test, use the cgv.CGV.plot method to plot output signals.

1 If you have not already done so, configure and test the model, as described in “Configuring the Model” on page 42-4 and “Executing the Model” on page 42-5.

2 Use the cgv.CGV.getOutputData method to get the output data from the `cgv.CGV` objects.

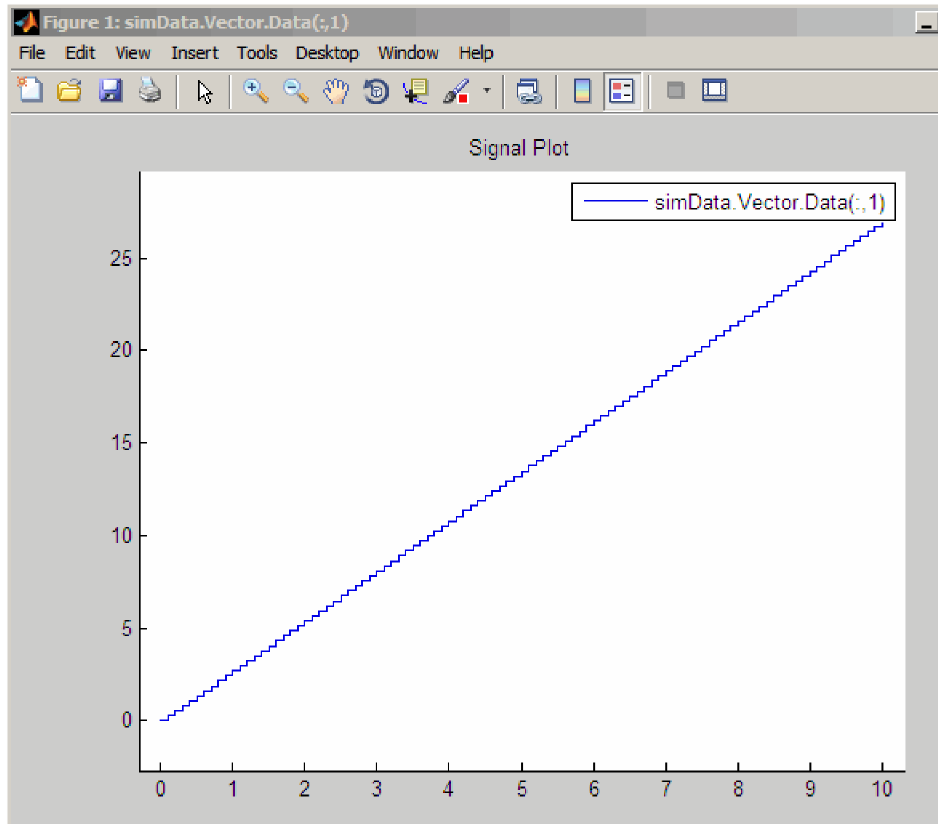
```
simData = cgvSim.getOutputData(1);
```

3 Use the cgv.CGV.getSavedSignals method to display the output data signal names. Build a list of specific signal names in a cell array of strings. The signal list can contain any number of signals.

```
cgv.CGV.getSavedSignals(simData);
signalList = {'simData.Vector.Data(:,1)'};
```

4 Use the specified signal list as input to the cgv.CGV.plot method to compare the signals from separate runs.

```
[signalNames, signalFigures] = cgv.CGV.plot(simData, ...
     'Signals', {'simData.Vector.Data(:,1)'});
```



Embedded IDEs and Embedded Targets

- Chapter 43, “Project and Build Configurations”
- Chapter 44, “Verification and Profiling”
- Chapter 45, “Processor-Specific Optimizations”
- Chapter 46, “Working with Altium TASKING IDE”
- Chapter 47, “Working with Analog Devices™ VisualDSP++ IDE”
- Chapter 48, “Working with Eclipse IDE”
- Chapter 49, “Working with Freescale MPC5xx Processors”
- Chapter 50, “Working with Green Hills® MULTI IDE”
- Chapter 51, “Working with Infineon C166 Processors”
- Chapter 52, “Working with Linux Target”
- Chapter 53, “Working with Microsoft Windows Target”
- Chapter 54, “Working with Texas Instruments Code Composer Studio IDE”
- Chapter 55, “Working with Texas Intruments C2000 Processors”
- Chapter 56, “Working with Texas Instruments C6000 Processors”
- Chapter 57, “Working with Wind River VxWorks Target”

43

Project and Build Configurations

- "Model Setup" on page 43-2
- "IDE Projects" on page 43-18
- "Makefiles" on page 43-24

Model Setup

In this section...

"Block Selection" on page 43-2
"Target Preferences" on page 43-4
"Configuration Parameters" on page 43-7
"Model Reference" on page 43-17

Block Selection

You can create models for targeting the same way you create other Simulink models—by combining standard blocks and C-MEX S-functions.

You can use blocks from the following sources:

- The Embedded Targets library (`embeddedtargetslib`) in the Embedded Coder product.
- Blocks from the DSP System Toolbox product
- The Discrete-time Integrator block from the Simulink® product
- Blocks that provide functions you need from any available blockset
- Custom blocks

In general, avoid using blocks that do not work in the generated code, such as the scope, source, and sink blocks. These blocks waste time in the generated code. They cause the executable to send data to, or wait for data from, your MATLAB workspace. They slow your application without adding instrumentation value.

Specifically, avoid using the following blocks in models from which you will be generating code.

Block Name/Category	Library	Description
Scope	Simulink, DSP System Toolbox software	Provides oscilloscope view of your output. Do not use the **Save data to workspace** option on the **Data history** pane in the Scope Parameters dialog box.
To Workspace	Simulink	Return data to your MATLAB workspace.
From Workspace	Simulink	Send data to your model from your MATLAB workspace.
Spectrum Scope	DSP System Toolbox	Compute and display the short-time FFT of a signal. It has internal buffering that can slow your process without adding value.
To File	Simulink	Send data to a file on your host machine.
From File	Simulink	Get data from a file on your host machine.
Triggered to Workspace	DSP System Toolbox	Send data to your MATLAB workspace.
Signal To Workspace	DSP System Toolbox	Send a signal to your MATLAB workspace.
Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
Triggered Signal From Workspace	DSP System Toolbox	Get a signal from your MATLAB workspace.
To Wave device	DSP System Toolbox	Send data to a `.wav` device.
From Wave device	DSP System Toolbox	Get data from a `.wav` device.

Target Preferences

This topic contains the following subtopics:

- "Supported IDEs" on page 43-4
- "What is a Target Preferences Block?" on page 43-4
- "Adding a Target Preferences Block to Your Model" on page 43-5
- "Creating a Library of Customized Target Preferences Blocks" on page 43-7

Supported IDEs

This "Target Preferences" on page 43-4 section applies to the following IDEs:

- Analog Devices VisualDSP++®
- Eclipse™ IDE
- Green Hills MULTI®
- Texas Instruments Code Composer Studio™
- Texas Instruments Code Composer Studio v4 (makefile generation only)
- Wind River Diab/GCC (makefile generation only)

For information about using Target Preferences with Altium TASKING, see "Setting Target Preferences for Altium TASKING" on page 46-8.

What is a Target Preferences Block?

To prepare a model for code generation, add a Target Preferences block to the model. This block describes the target environment for which you are generating code. The block includes information about the processor, hardware settings, operating system, memory mapping, and code generation features. Simulink Coder, Embedded Coder, and Simulink products use this information to generate code from your model.

For detailed information about the Target Preference block parameters and options, consult the Target Preferences block reference topic.

Key Points

- To generate code, the model must contain a Target Preferences block.
- Use one Target Preferences block per model. Exceptions to this rule are noted in the documentation for specific features, such as “Model Block PIL” on page 44-4.
- Changing Target Preferences block settings can change tabs, panes, parameters, and options visible when you open the block.
- Adding a Target Preferences block to a model changes the values of that model’s Configuration Parameters. If you need to preserve the Configuration Parameters of a specific model, make a backup copy of the model before adding a Target Preferences block to it.

Adding a Target Preferences Block to Your Model

To generate code, your model must contain only one Target Preferences block.

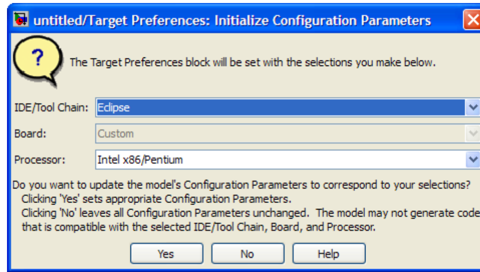
- When you are generating code for a model, place the Target Preferences block at the top level of your model.
- When you are generating code for a subsystem in your model, place the Target Preferences block at the subsystem level of your model.

Note Adding a Target Preferences block to a model changes the values of that model’s Configuration Parameters. If you need to preserve the Configuration Parameters of a specific model, make a backup copy of the model before adding a Target Preferences block to it.

To add a Target Preferences block to your model:

1. Open the Simulink library browser.
2. Copy the Target Preferences block from the **Embedded Coder** > **Embedded Targets** library to your model

3 The software dislays the **Initialize Configuration Parameters** dialog box. For example.



Set the following parameters, and click **Yes**:

- **IDE/Tool Chain**
- **Board**
- **Processor**

When you click **Yes**, the software automatically sets the model Configuration Parameters to the proper values for the IDE/Tool Chain, Board, and Processor you selected. In this case, you have completed the process of adding a Target Preferences block to your model.

If you click **No**, the software leaves the values of the Configuration Parameters unchanged. The model cannot simulate or generate code correctly unless you configure the Configuration Parameters with the proper values. Setting these values manually can be difficult.

Note The following actions update the appropriate model Configuration Parameters with new values:

- Adding a Target Preferences block to your model and clicking Yes in the **Initialize Configuration Parameters** dialog box.
- Opening the Target Preferences block in your model and selecting a new **IDE/Tool Chain**.
- Opening the Target Preferences block in your model and applying changes to the **Board** and **Processor** parameters.

Note The **Initialize Configuration Parameters** dialog box uses your previous selections as default values the next time you copy a Target Preference block to a model.

Creating a Library of Customized Target Preferences Blocks

If you work regularly with a variety of IDEs, tool chains, boards and processors, you can save time by creating a library of customized Target Preferences blocks. Later, you reuse these preconfigured Target Preferences blocks instead repeating the customization process on a new Target Preferences block.

To create a library of customized Target Preferences blocks:

1. In Simulink, select **File** > **New** > **Library**. This action creates a new untitled library.

2. Save the library to a folder included in your MATLAB search paths.

Note Click **File** > **Set Path** to see a list of MATLAB search paths and add new ones.

---

3. Copy or drag configured Target Preferences blocks from your models to the library.

4. Edit the label of each block to describe that block's configuration.

5. After the new blocks are added and labeled, save the library.

To copy a Target Preferences block from your library to a model, type the library name at the MATLAB command line. When the library appears, copy the block to your model.

Configuration Parameters

- "What are Configuration Parameters?" on page 43-8

- "Setting Model Configuration Parameters" on page 43-8

What are Configuration Parameters?

To see the model configuration parameters, open the **Configuration Parameters** dialog box. You can do this in the model editor by selecting **Simulation > Configuration Parameters** , or by pressing **Ctrl+E** on your keyboard.

The **Configuration Parameters** dialog box specifies the values for a model's active *configuration set*. These parameters determine the type of solver used, the import and export settings, and other values that determine how the model runs.

To set the configuration parameters to the correct values for you to generate code from your model, add a Target Preferences block to your model. This action initializes the model Configuration Parameters to the appropriate default values for you to generate code. You can then use the Configuration Parameters dialog box to make further modifications to the values.

For more information, see "Configuration Parameters Dialog Box" and "Managing Configuration Sets".

Setting Model Configuration Parameters

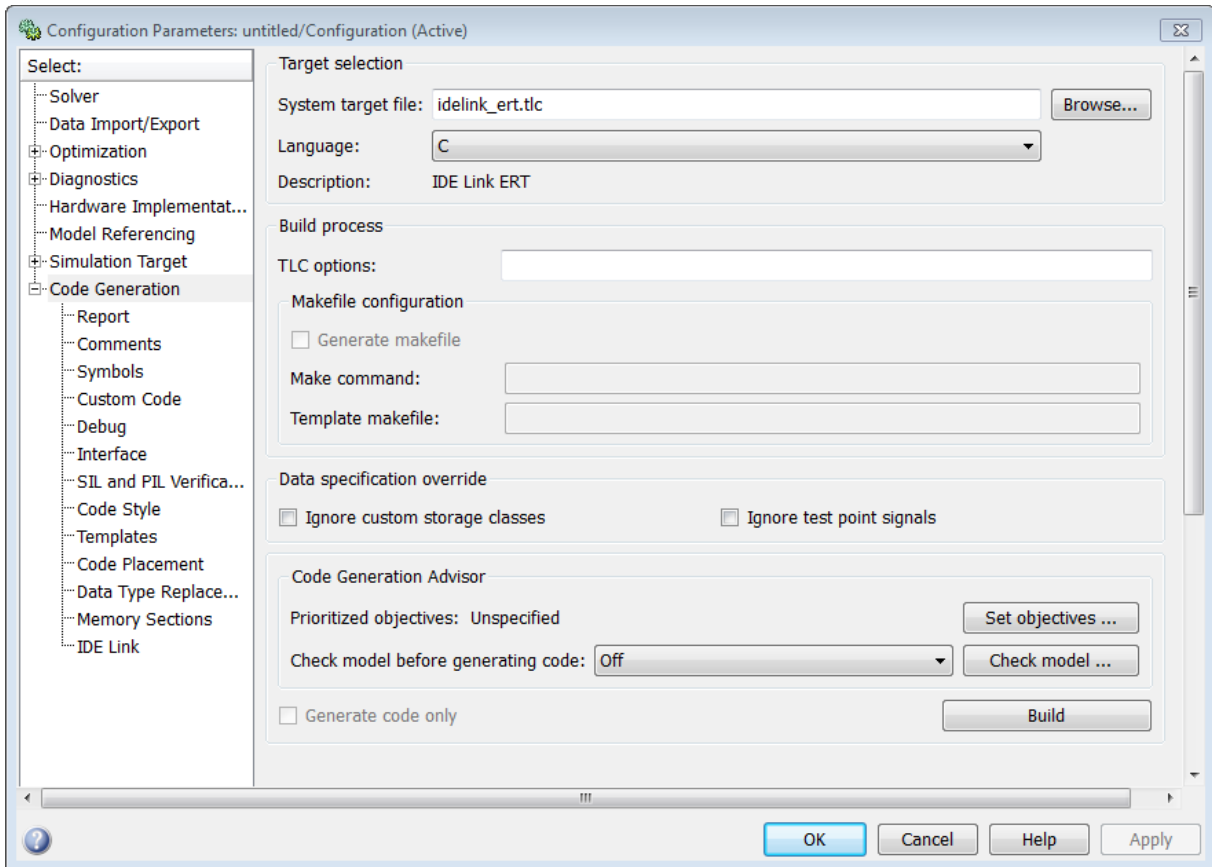
To apply the appropriate default values in Configuration Parameters, add a Target Preferences block to your model and select the **Initialize Configuration Parameters**, as described in "Adding a Target Preferences Block to Your Model" on page 43-5. You can generate buildable code using these default values.

To make further changes, select **Simulation > Configuration Parameters** in the Model Editor, or press **Ctrl+E**. This action opens the **Configuration Parameters** dialog box.

Note You can generate buildable code after adding a Target Preferences block to your model and initializing the Configuration Parameters.

The following subsections provide a quick overview of the panes and parameters with which you are most likely to interact.

Code Generation Pane. When you set **System target file** to idelink_ert.tlc or idelink_grt.tlc, the dialog box displays a **IDE Link** at the bottom of the select tree.



Setting the **System target file** to idelink_ert.tlc requires an Embedded Coder license.

Leave **Language** set to C. The idelink_ert.tlc and idelink_grt.tlc system target files do not support C++ code generation.

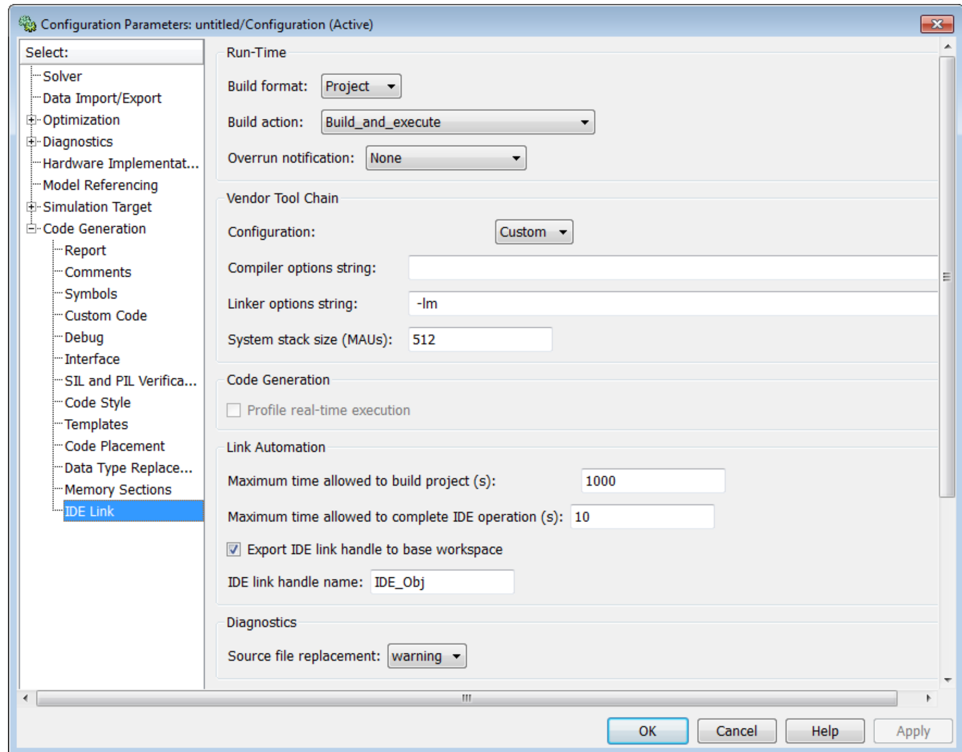
Disregard the **Makefile configuration** options. To generate makefiles, use a separate feature to automatically generate makefiles. For more information, see "Makefiles" on page 43-24.

If you generate code from a model that uses custom storage classes (CSC), leave **Ignore custom storage classes** unselected.

To use a system target file that does not support CSCs, without reconfiguring your parameter and signal objects, select **Ignore custom storage classes**. (For example, `idelink_grt.tlc` does not support CSCs.) When you select **Ignore custom storage classes**:

- The software treats objects with CSCs as if you set their storage class attribute to `Auto`.
- The storage class of signals that have CSCs does not appear on the signal line, even when you select `Storage class` from **Format > Port/Signals Display** in your Simulink menus.

IDE Link Pane Parameters.



On the select tree, the IDE Link entry provides options in these areas:

- **Run-Time** — Set options for run-time operations, like the build action
- **Vendor Tool Chain** — Set compiler, linker, and system stack size options
- **Code Generation** — Configure your code generation requirements
- **Link Automation** — Export an IDE handle object, such as `IDE_Obj`, to your MATLAB workspace
- **Diagnostics** — Determine how the code generation process responds when you use source code replacement in the **Custom Code** pane.

For comprehensive detailed information, see IDE Link Pane.

Build format

Select `Project` to create an IDE project, or select `Makefile` to create a makefile build script.

Build action

Your selection for **Build action** determines what happens when you click **Build** or press **Ctrl+B**. Your selection tells Simulink Coder software when to stop the code generation and build process.

To run your model on the processor, select `Build_and_execute`. This selection is the default build action; Simulink Coder software automatically downloads and runs the model on your board.

The actions are cumulative—each action performs an additional step relative to the preceding action on the list.

If you set **Build format** to `Project`, select one of the following options:

- `Create_Project` — Directs Simulink Coder software to start the IDE and populate a new project with the files from the build process. This option offers a convenient way to build projects in the IDE.
- `Archive_library` — Directs Simulink Coder software to create an archive library for this model. Use this option when you plan to use the model in a model reference application. Model reference requires that you archive your the IDE projects for models that you use in model referencing.
- `Build` — Builds the executable file, but does not download the file to the target.
- `Build_and_execute` — Directs Simulink Coder software to build, download, and run your generated code as an executable on your target.
- `Create_processor_in_the_loop_project` — Directs code generation process to create PIL algorithm object code as part of the project build. This option requires an Embedded Coder license.

If you set **Build format** to `Makefile`, select one of the following options:

- `Create_makefile` — Creates a makefile.
- `Archive_library` — Creates a makefile and the generated output will be an archive library.
- `Build` — Creates a makefile and an executable.
- `Build_and_execute` — Creates a makefile and an executable. Then it evaluates the execute instruction in the current configuration.

Overrun notification

To enable the overrun indicator, choose one of three ways for the target to respond to an overrun condition in your model:

- `None` — Ignore overruns encountered while running the model.
- `Print_message` — When the target encounters an overrun condition, it prints a message to the standard output device, `stdout`.
- `Call_custom_function` — Respond to overrun conditions by calling the custom function you identify in **Function name**.

Function name

When you select `Call_custom_function` from the **Overrun notification** list, you enable this option. Enter the name of the function the target should use to notify you that an overrun condition occurred. The function must exist in your code on the target.

Configuration

The **Configuration** parameter defines sets of build options that apply to all of the files generated from your model.

The `Release` and `Debug` option apply build settings that are defined by your compiler. For more information, refer to your compiler documentation.

`Custom` has the same default values as `Release`, but:

- Leaves **Compiler options string** empty.

Compiler options string

To determine the degree of optimization provided by the optimizing compiler, enter the optimization level to apply to files in your project. For details about the compiler options, refer to your IDE documentation. When you create new projects, the coder software does not set any optimization flags.

For certain IDEs, if you have an active project open in the IDE, you can click **Get From IDE** to import the compiler option setting from the current project in the IDE. To reset the compiler option to the default value, click **Reset**.

Linker options string

To specify the options provided by the linker during link time, you enter the linker options as a string. For details about the linker options, refer to your IDE documentation. When you create new projects, the coder software does not set any linker options.

If you have an active project open in the IDE, you can click **Get From IDE** to import the linker options string from the current project in the IDE. To reset the linker options to the default value of no options, click **Reset**.

System stack size (MAUs)

Enter the number of minimum addressable units (MAU) to use for the stack. For more information, refer to **Enable local block outputs** on the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box. Block output buffers are placed on the stack until the stack memory is fully allocated. After that, the output buffers go in global memory. An MAU is typically 1 byte, but its size can vary by target processor.

System heap size (MAUs)

Enter the amount of minimum addressable units (MAU) to use for the heap. An MAU is typically 1 byte, but its size can vary by target processor.

Profile real-time execution

To enable the real-time execution profile capability, select **Profile real-time execution**. With this selected, the build process instruments your code to provide performance profiling at the task level or for atomic subsystems. When you run your code, the executed code reports the profiling information in an HTML report.

Link Automation

When you build a model for a target, the coder software automatically creates or uses an existing *IDE handle object* (named `IDE_Obj`, by default) to connect to your IDE.

Although `IDE_Obj` is a handle for a specific instance of the IDE, it also contains information about the IDE instance to which it refers, such as the board and target the IDE accesses. In this pane, the **Export IDE link handle to base workspace** option lets you instruct the coder software to export the object to your MATLAB workspace, giving it the name you assign in **IDE link handle name**.

You can also use the IDE handle object to interact with the IDE using Automation Interface commands.

Maximum time allowed to build project (s)

Specifies how long the software waits for the IDE to build the software.

Maximum time allowed to complete IDE operations (s)

Specifies how long the software waits for IDE functions, such as `read` or `write`, to return completion messages. If you do not specify a timeout

Export IDE link handle to base workspace

Directs the software to export the IDE_Obj object to your MATLAB workspace.

IDE link handle name

Specifies the name of the IDE_Obj object that the build process creates.

Source file replacement

Selects the diagnostic action to take if the software detects conflicts when you replace source code with custom code. The diagnostic message responds to both source file replacement in the Configuration Parameters under Code Generation > IDE link parameters and under Code Generation > Custom Code.

The following settings define the messages you see and how the code generation process responds:

- `none` — Does not generate warnings or errors when it finds conflicts.
- `warning` — Displays a warning. `warn` is the default value.
- `error` — Terminates the build process and displays an error message that identifies which file has the problem and suggests how to resolve it.

The build operation continues if you select `warning` and the software detects custom code replacement problems. You see warning messages as the build progresses.

Select `error` the first time you build your project after you specify custom code to use. The error messages can help you diagnose problems with your

custom code replacement files. Use `none` when the replacement process is correct and you do not want to see multiple messages during your build.

Model Reference

The `idelink_ert.tlc` and `idelink_grt.tlc` system target files provide support for generating code from models that use Model Reference. A referenced model will generate an archive library.

To enable Model Reference builds:

1. Open your referenced model.
2. Select Simulation > Configuration Parameters from the model menus.
3. From the Select tree, choose **IDE Link**.
4. In the right pane, under Runtime, select `Archive_library` from the **Build action** list.

If your top model uses a reference model that does not have the **Build action** set to `Archive_library`, the build process automatically changes the **Build action** to `Archive_library` and issues a warning about the change.

Target Preferences Blocks in Reference Models

Each referenced model and the top model must include a Target Preferences block for the correct processor. You must configure all the Target Preferences blocks for the same processor. This allows the code generation be consistent across the model hierarchy. This also ensures that consistent tools – compiler, linker and archivers, are used during the build process.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information. Target Preferences provides all the necessary information for the referenced models.

IDE Projects

In this section...

"Third Party Product Setup" on page 43-18

"Installation of MathWorks Products on 64-bit Host Computers" on page 43-20

"IDE Link Configuration" on page 43-20

"Code Generation and Build" on page 43-21

"Automation of IDE Tasks and Processes" on page 43-22

Third Party Product Setup

Install your third party IDE or software build tool chain according to the vendors instructions.

For more information about Embedded Coder support for third-party IDEs and targets, see the following links, organized by vendor:

- Analog Devices™ VisualDSP++ IDE and Blackfin® processors — http://www.mathworks.com/products/embedded-coder/adi-adaptor.html
- Altium® TASKING IDE — http://www.mathworks.com/products/embedded-coder/altium-adaptor.html
- Eclipse IDE — http://www.mathworks.com/products/embedded-coder/eclipse-adaptor.html
- Green Hills® MULTI® — http://www.mathworks.com/products/embedded-coder/ghs-adaptor.html
- Texas Instruments' Code Composer Studio™ IDE, and C2000™, C5000™, C6000™ processors — http://www.mathworks.com/products/embedded-coder/ti-adaptor.html
- ARM® — http://www.mathworks.com/products/embedded-coder/arm-adaptor/
- Freescale MPC5xx processors — http://www.mathworks.com/products/embedded-coder/freescale-adaptor/

- Infineon® C166® processors — http://www.mathworks.com/products/embedded-coder/infineon-adaptor/
- Wind River VxWorks — http://www.mathworks.com/products/embedded-coder/windriver-adaptor/

For Code Composer Studio

Before you use Embedded Coder with Code Composer Studio (CCS IDE) for the first time, use the checkEnvSetup function to check for third-party tools and set environment variables. Run checkEnvSetup again whenever you configure CCS IDE to interact with a new board or processor, or upgrade any of the related third-party tools.

To verify that CCS is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

at the MATLAB software command line. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

```
Board Board                        Proc Processor  Processor
 Num  Name                         Num  Name        Type
 ---  ------------------------------  ---  -------------
1  C6xxx Simulator (Texas Instrum .0   6701     TMS320C6701
0  C6x13 DSK (Texas Instruments)   0   CPU      TMS320C6x1x
```

If MATLAB software does not return information about any boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to ensure that it starts up successfully. For Embedded Coder to operate with CCS, the CCS IDE must be able to run on its own.

For Eclipse

To install Eclipse, complete the instructions in “Installing Third-Party Software for Eclipse” on page 48-4.

Installation of MathWorks Products on 64-bit Host Computers

For 64-bit host computers, install the 32-bit version of your MathWorks products. The 64-bit version of your MathWorks products does not support the Embedded IDEs and Embedded Targets-related features.

IDE Link Configuration

After completing "Third Party Product Setup" on page 43-18, if you are using the one of the following IDEs, perform additional configuration of your MathWorks software:

Analog Devices VisualDSP++ IDE

Use the `adivdspsetup` command to complete the configuration and install a plug-in in VisualDSP++. The reference page provides a brief example. Also see Chapter 47, "Working with Analog Devices™ VisualDSP++ IDE".

Eclipse IDE

Complete the process in "Configuring Your MathWorks Software to Work with Eclipse" on page 48-10.

Green Hills MULTI IDE

Use the `ghsmulticonfig` command to complete the configuration for MULTI. Also see Chapter 50, "Working with Green Hills® MULTI IDE".

Texas Instruments Code Composer Studio IDE

Use the `checkEnvSetup` command to complete the configuration for Code Composer Studio. Also see Chapter 54, "Working with Texas Instruments Code Composer Studio IDE".

Code Generation and Build

Building Your Model

In your model, click the build button or enter **Ctrl+B**. The software performs the actions you selected for **Build action** in the model Configuration Parameters, under Code Generation > IDE Link.

Green Hills MULTI Output Folder

With Green Hills MULTI, Embedded Coder outputs the derived files in the `<builddir>` folder. For example, in `model_ghsmulti`.

Project Generator Features

The *Project Generator* component provides or supports the following features for developing IDE projects and generating code:

- Automatically create IDE projects for your generated code during the code generation process.
- Customize code generation using model "Configuration Parameters" on page 43-7 and "Target Preferences" on page 43-4 block options
- Configure the automatic project build process
- Automatically download and run your generated projects on your target

IDE Handle Objects

Project Generator uses an IDE handle object to connect to the IDE you selected in the Target Preferences block. If a handle doesn't exist when you generate code, Project Generator automatically creates a handle object for your IDE. Project Generator uses a constructor function to create a handle object for the your IDE:

- `altiumtasking` for Altium TASKING
- `adivdsp` for Analog Devices™ VisualDSP++®
- `eclipseide` for Eclipse IDE
- `ghsmulti` for Green Hills MULTI

- `ticcs` for Texas Instruments' Code Composer Studio

Automation of IDE Tasks and Processes

The *Automation Interface* component provides a powerful API for automating IDE tasks via MATLAB scripts. For example, with Automation Interface, your script can automatically:

- Automate project creation, including adding source files, include paths, and preprocessor defines
- Configure batch building of projects
- Launch a debugging session

Getting Started with Automation Interface

For your reference, consult the list of the supported functions and methods for your IDE, located in the Embedded Coder reference under "IDE Automation Interface".

The implementation of Automation Interface for TASKING is different from the implementation for other IDEs. For TASKING, use the "Automation Interface" on page 46-37 topic in the TASKING appendix instead of this section.

Introducing the Automation Interface Tutorial. To help you become familiar with Automation Interface, you can use the "Automation Interface Tutorial" demo for the following IDEs:

- Altium TASKING
- Green Hills MULTI
- Eclipse IDE
- Analog Devices VisualDSP++
- Texas Instruments Code Composer Studio

The tutorial shows you how to:

1 Configure and create an IDE handle object.

2. Create and query objects in an IDE.
3. Use MATLAB software to load files into your IDE.
4. Work with your IDE project from MATLAB software.
5. Close connections you the IDE.

Makefiles

In this section...

"Using XMakefile to Generate and Build Software" on page 43-24

"Making an XMakefile Configuration Operational" on page 43-31

"Example: Creating a New XMakefile Configuration" on page 43-31

"XMakefile User Configuration Dialog Box" on page 43-38

Using XMakefile to Generate and Build Software

- "Overview" on page 43-24
- "Feature Support" on page 43-25
- "Configuring Your Model to Use Makefiles" on page 43-26
- "Choosing an XMakefile Configuration" on page 43-27
- "Building Your Model" on page 43-30

Overview

You can use the XMakefile feature to generate and build your software using makefiles.

Scenarios for using makefiles include:

- Building software without opening an IDE
- Automating the build process for testing and continuous build environments
- Fine-tuning and customizing the build process

In addition to this chapter, see the Makefile Generator Tutorial demo for more information about using makefiles to generate code.

You can use makefiles with the following toolchain and target combinations.

Tool Chain	Processor Family/Target Operating System	Host Operating System
Analog Devices™ VisualDSP++®	Blackfin™, SHARC™, and TigerSHARC™	Windows
Green Hills® MULTI®	ARM®, Analog Devices™ Blackfin®, PowerPC®, and NEC® v850	Linux, Windows
GNU development tools	Linux	Linux
GNU development tools	ARM	MontaVista Linux
Texas Instruments™Code Composer Studio v3 and v4	Texas InstrumentsC2000™, C5500™, and C6000™	Windows
Wind River Diab/GCC	ARM9, Host Simulator, VxWorks, RTP and RTP_SO	Windows

Feature Support

With XMakefile, you cannot use features that rely on direct communications between your MathWorks software and third-party IDEs.

You cannot use the following features with makefiles:

- Automation Interface
- Processor-in-the-loop (PIL) communications via an IDE debugger

If your IDE supports them, you can use makefiles with the following features and Embedded Coder:

- Real-Time executable
- Free-running executables
- Interrupt Block

- Idle Block
- Linker File
- TFL Operators
- TCP/IP PIL

If your target supports them, you can use makefiles with the following features:

- Real-Time Operating Systems such as DSP/BIOS, VxWorks, or Real-Time Linux
- Windows and Linux
- Device Drivers
- External mode
- Optimization

Note Using XMakefile with demos: You can only complete those demos or portions of demos that rely on features that XMakefile supports.

Configuring Your Model to Use Makefiles

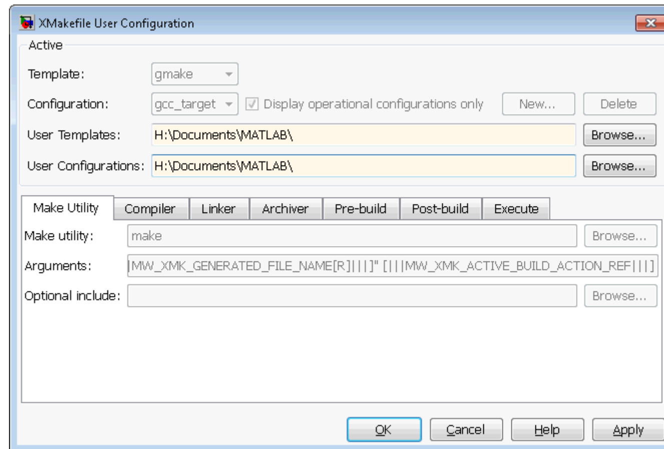
Update your model configuration parameters to use a makefile instead of an IDE when you build software from the model:

1. Add a Target Preferences block to your model and configure it for your target. For more information, see "Target Preferences" on page 43-4.
2. In your model window, select **Simulation > Configuration Parameters**.
3. Under **Code Generation**, select **IDE Link**.
4. Set **Build format** to `Makefile`. For more information, see Build format on page 12.
5. Set **Build action** to `Build_and_execute`. For more information, see Build action on page 12.

Choosing an XMakefile Configuration

Configure how to generate makefiles:

1. Enter `xmakefilesetup` on the MATLAB command line. The software opens an XMakefile User Configuration dialog box.



2. Leave **Template** set to `gmake`.

3. For **Configuration**, select the option that describes your software build toolchain and target platform.

Note With the XMakefile User Configuration dialog, if you have an Embedded Coder license but no Simulink Coder license, the **Configuration** list includes two unsupported options: `gcc_target` or `msvs_host`. Disregard those two configurations. Choose one of the other configurations.

Note If you set **Configuration** to `msvs_host`, restart MATLAB as described in before building your model software.

Things to consider while setting **Configuration**:

- Selecting **Display operational configurations only** hides configurations that do not work. For a configuration to be operational, the vendor tool chain must be installed, and the configuration must have the correct path for the vendor tool chain. For more information, see "Making an XMakefile Configuration Operational" on page 43-31.
- To display all of the configurations, including non-operational configurations, deselect **Display operational configurations only**.
- The list of configurations can include non-editable configurations defined in the software and editable configurations defined by you.
- To create a new editable configuration, use the **New** button.
- For more information, see "XMakefile User Configuration Dialog Box" on page 43-38.

Available XMakefile Configurations. The following list provides a description each configuration available in the XMakefile dialog box:

- `adivdsp_blackfin`: Analog Devices VisualDSP++ & Analog Devices Blackfin
- `adivdsp_sharc`: Analog Devices VisualDSP++ & Analog Devices SHARC
- `adivdsp_tigersharc`: Analog Devices VisualDSP++ & Analog Devices TigerSHARC
- `gcc_target`: GNU Compiler Collection & Host Operating System or Embedded Operating System
- `ghsmulti_arm`: Green Hills MULTI & ARM
- `ghsmulti_blackfin`: Green Hills MULTI & Analog Devices Blackfin
- `ghsmulti_ppc`: Green Hills MULTI & PowerPC
- `ghsmulti_v850`: Green Hills MULTI & NEC V850
- `mingw_host`: Minimalist GNU for Windows & Host Operating System
- `msvs_host`: Microsoft Visual Studio & Host Operating System
- `ticcs_c2000_ccsv3`: Texas Instruments Code Composer Studio v3 & Texas Instruments C2000

- `ticcs_c2000_ccsv4`: Texas Instruments Code Composer Studio v4 & Texas Instruments C2000
- `ticcs_c5500_ccsv3`: Texas Instruments Code Composer Studio v3 & Texas Instruments C5500
- `ticcs_c5500_ccsv4`: Texas Instruments Code Composer Studio v4 & Texas Instruments C5500
- `ticcs_c6000_ccsv3`: Texas Instruments Code Composer Studio v3 & Texas Instruments C6000
- `ticcs_c6000_ccsv4`: Texas Instruments Code Composer Studio v4 & Texas Instruments C6000
- `ticcs_c6000_dspbios_ccsv3`: Texas Instruments Code Composer Studio v3 & Texas Instruments DSP/BIOS on C6000
- `ticcs_c6000_dspbios_ccsv4`: Texas Instruments Code Composer Studio v4 & Texas Instruments DSP/BIOS on C6000
- `wrsdiab_arm9_rtp`: Wind River Systems DIAB Compiler & ARM 9 & real-time process applications
- `wrsdiab_arm9_rtp_so`: Wind River Systems DIAB Compiler & ARM 9 & real-time process applications with shared object
- `wrsdiab_hostsim_rtp`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & real-time process applications
- `wrsdiab_hostsim_rtp_so`: Wind River Systems DIAB Compiler & VxWorks Host Simulator & real-time process applications with shared object
- `wrsgnu_arm9_vxworks_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & real-time process applications
- `wrsgnu_hostsim_vxworks_rtp`: Wind River Systems GNU Compiler & VxWorks Host Simulator & real-time process applications with shared object

Understanding Naming Conventions for the XMakefile Configurations. The names of XMakefile configurations consist of abbreviations and underscores in the following sequence:

Vendor + Tool Chain + Underscore + Processor Family + Underscore + Relevant Modifier

For example:

Name	Vendor	Tool Chain	Processor Family	Modifier
adivdsp_tigersharc	Analog Devices (adi)	VisualDSP++ (vdsp)	TigerSharc (tigersharc)	
ghsmulti_arm	Green Hills (ghs)	MULTI (multi)	ARM (arm)	
ticcs_c6000_ccsv3	Texas Instruments (ti)	Code Composer Studio (ccs)	C6000 (c6000)	CCS version 3 (ccsv3)
mingw_host		Minimalist GNU for Windows (mingw)	Host Operating System (host)	
msvs_host	Microsoft (ms)	Visual Studio (vs)	Host Operating System (host)	
wrsdiab_arm9_rtp_so	Wind River Systems (wrs)	DIAB Compiler (diab)	ARM9 (arm 9)	Real-time process applications with shared object (rtp_so)
wrsgnu_hostsim_rtp	Wind River Systems (wrs)	GNU Compiler (gnu)	VxWorks Host Simulator (hostsim)	Real-time process applications (rtp)

Building Your Model

In your model, click the build button or enter **Ctrl+B**. This action creates a makefile and performs the other actions you specified in **Build action**.

By default, this process outputs files in the `<builddir>/<buildconfiguration>` folder. For example, in `model_name/CustomMW`.

Green Hills MULTI Output Folder

With Green Hills MULTI, Embedded Coder outputs the derived files in the `<builddir>` folder. For example, in `model_ghsmulti`.

Making an XMakefile Configuration Operational

When the XMakefile utility starts, it checks each factory default configuration to verify that the specified toolchain paths exist. If the paths are invalid, the configuration is non-operational. Typically, the cause of this problem is a difference between the path in the configuration and the actual path of the vendor toolchain.

To make a configuration operational:

1. Deselect **Display operational configurations only** to display non-operational configurations.
2. Select the non-operational configuration from the **Configuration** options.
3. When you click **Apply**, a new dialog box prompts you for the folder path of any missing resources the configuration requires.

   Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Example: Creating a New XMakefile Configuration

- "Overview" on page 43-32
- "Create a Configuration" on page 43-32
- "Modify the Configuration" on page 43-33
- "Test the Configuration" on page 43-36

Overview

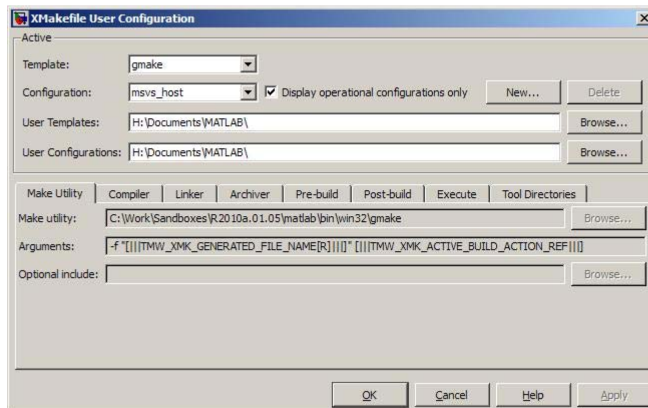
This example shows you how to add support for a software development toolchain to the XMakefile utility. This example uses the Intel Compiler and Eclipse IDE, which provides an open framework and allows for otherwise unsupported toolchains.

Note To specify directory locations, use mapped network drives instead of UNC paths. UNC paths cause build errors with compilers that do not support them.

Create a Configuration

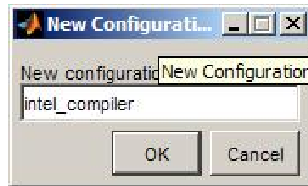
When you click **New**, the new configuration inherits values and behavior from the current configuration. To create a configuration for the Intel Compiler, clone a configuration from any of these configurations: `msvs_host`, `mingw_host`, `montavista_arm` and `gcc_target`.

Open the XMakefile User Configuration UI by typing `xmakefilesetup` at the MATLAB prompt. This action displays the following dialog box.

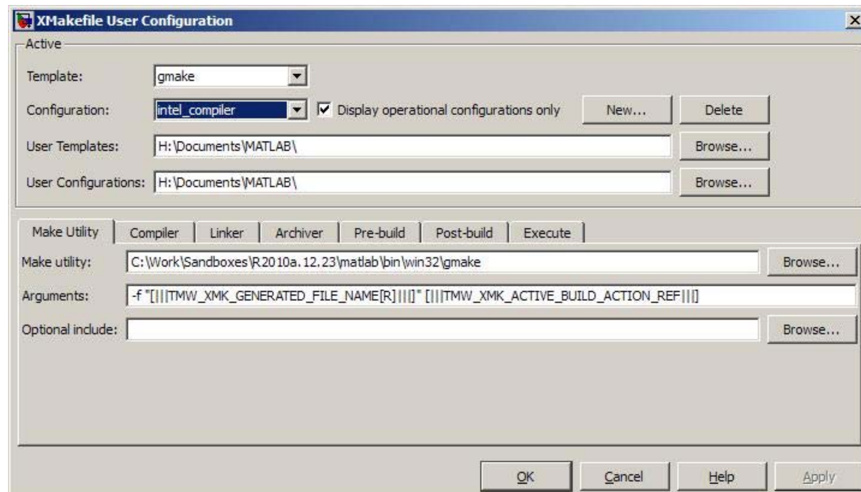


Select an existing configuration, such as `msvs_host`, `mingw_host`, `montavista_arm` or `gcc_target`. Click the **New** button.

A pop-up dialog prompts you for the name of the new configuration. Enter `intel_compiler` and click **OK**.



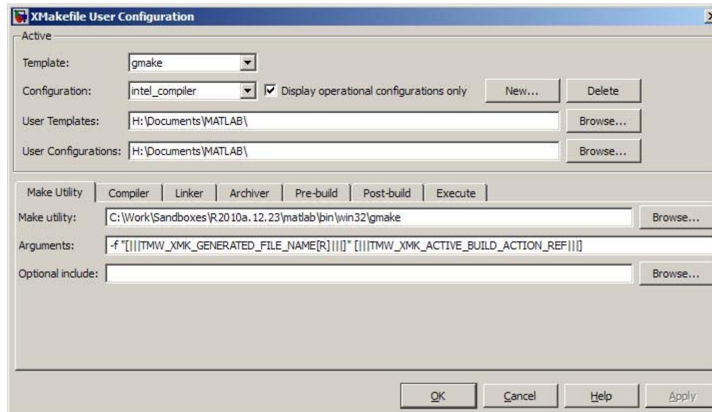
The dialog box displays a new configuration called `intel_compiler` based on `msvs_host`.



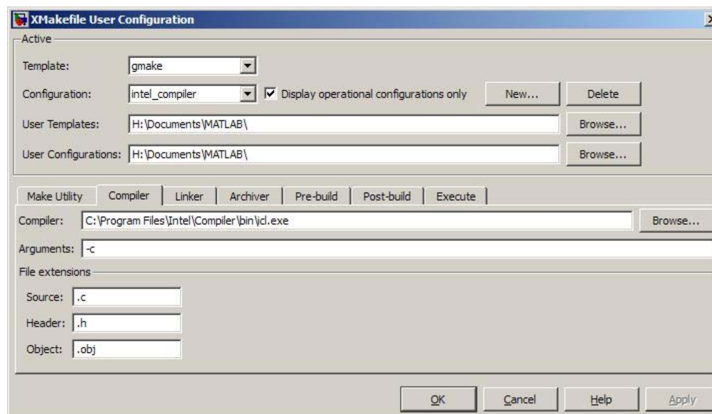
Modify the Configuration

Adjust the compiler, linker, and archiver settings of the newly created configuration. This example assumes the location of the Intel compiler is `C:\Program Files\Intel\Compiler\`.

Make Utility. You do not need to make any changes. This configuration uses the gmake tool that ships with MATLAB.

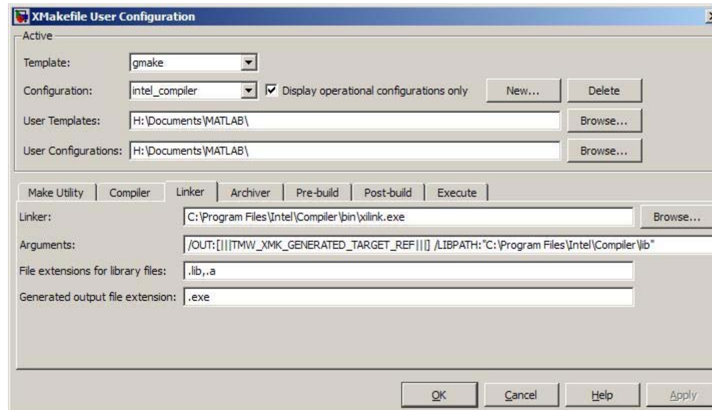


Compiler. For **Compiler**, enter the location of `icl.exe` in the Intel installation.

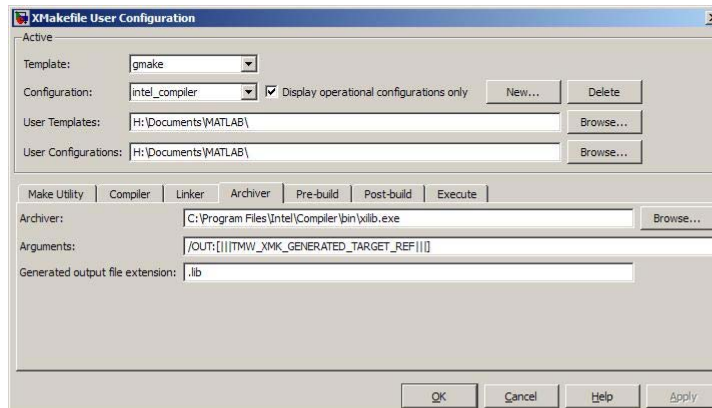


Linker. For **Linker**, enter the location of the linker executable, xilink.exe.

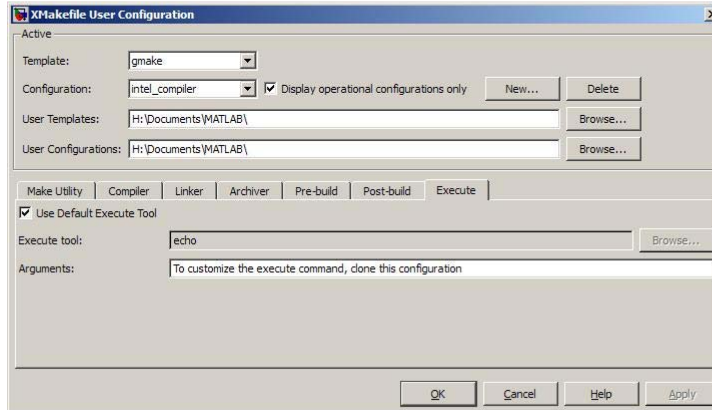
For **Arguments**, add the /LIBPATH path to the Intel libraries.



Archiver. For **Archiver**, enter the location of the archiver, `xilib.exe`. Confirm that **File extensions for library files** includes `.lib`.

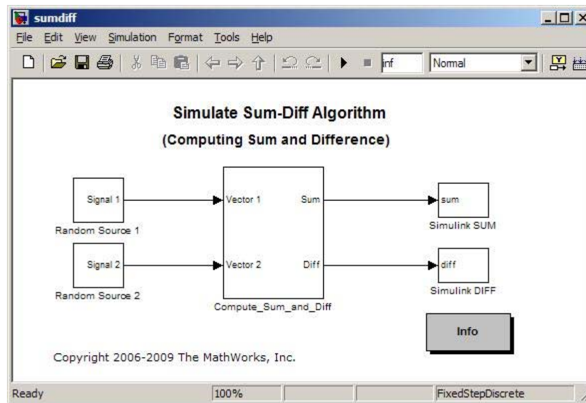


Other tabs. For this example, ignore the remaining tabs. In other circumstances, you can use them to configure additional build actions. In a later step of this example, you will configure the software to automatically build and run the generated code.

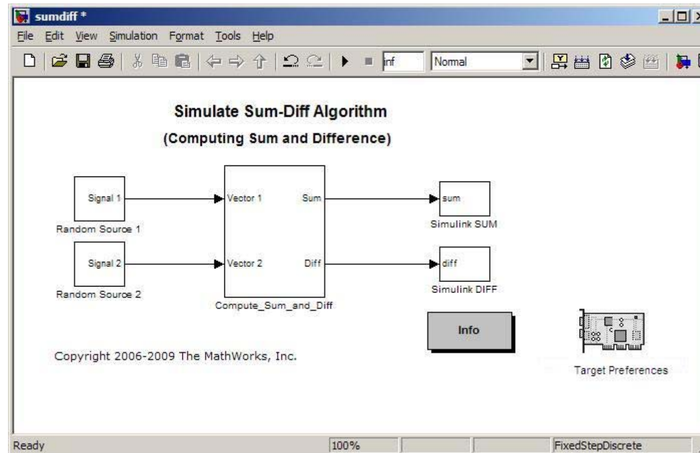


Test the Configuration

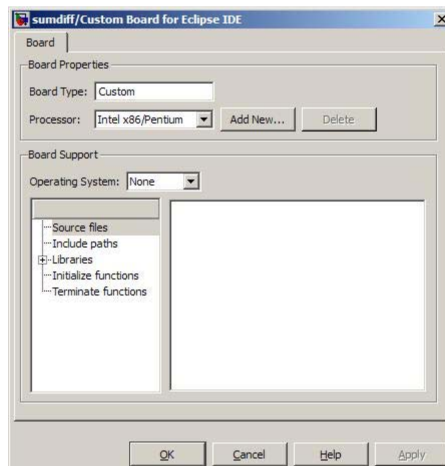
Open the "sumdiff" model by entering `sumdiff` on the MATLAB prompt.



Use a Target Preferences block to configure the model for use with the Eclipse IDE. First enter `idelinklib_common` at the MATLAB prompt. Then drag and drop the Target Preferences block onto the `summdiff` model.



Open the Target Preferences block, and set **Processor** to `Intel x86/Pentium`. Set **Operating System** to `None` or select `Windows`. Click **OK**.



Open the Configuration Parameters for the summdiff model by pressing **Ctrl+E**. Set **Build format** to `Makefile` and **Build action** to `Build_and_execute`.

Save the model to a temporary location, such as `C:\Temp\IntelTest\`.

Set that location as a Current Folder by typing cd `C:\temp\IntelTest\` at the MATLAB prompt.

Build the model by pressing **Ctrl+B**. The MATLAB Command Window displays something like:

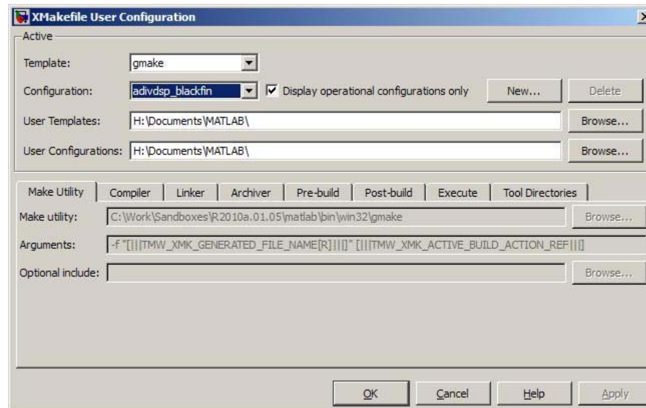
```
### TLC code generation complete.
### Creating HTML report file sumdiff_codegen_rpt.html
### Creating project: c:\temp\IntelTest\sumdiff_eclipseide\sumdiff.mk
### Project creation done.
### Building project...
### Build done.
### Downloading program: c:\temp\IntelTest\sumdiff_eclipseide\sumdiff
### Download done.
```

A command window comes up showing the running model. Terminate the generated executable by pressing **Ctrl+C**.

XMakefile User Configuration Dialog Box

- "Active" on page 43-39
- "Make Utility" on page 43-40
- "Compiler" on page 43-41
- "Linker" on page 43-42
- "Archiver" on page 43-42
- "Pre-build" on page 43-43
- "Post-build" on page 43-43
- "Execute" on page 43-44
- "Tool Directories" on page 43-44

Active



Template. Select the template that matches your toolchain and processor. The template defines the syntax rules for writing the contents of the makefile or buildfile. The factory default template is `gmake`, which works with the GNU make utility.

To add templates to this parameter, save them as .mkt files to the location specified by the **User Templates** parameter. For more information, see "User Templates" on page 43-40.

Configuration. Select the configuration that best describes your toolchain and target.

You cannot edit or delete the factory default configurations provided by MathWorks. You can, however, edit and delete the configurations that you create.

Use the **New** button to create an editable copy of the currently selected configuration.

Use the **Delete** button to delete a configuration you created.

Note You cannot edit or delete the factory default configurations provided by MathWorks.

Note Use mapped network drives instead of UNC paths to specify directory locations. Using UNC paths with compilers that do not support them causes build errors.

Display operational configurations only. When you open the XMakefile User Configuration dialog box, the software verifies that each factory default configuration contains valid paths to the executable files it uses.

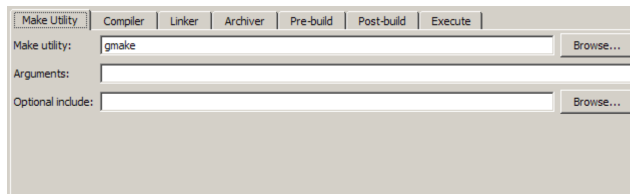
To display valid factory default configurations, select "Display operational configurations only". This option does not apply to configurations you create.

To display all of the configurations, including non-operational configurations, deselect **Display operational configurations only**. The software categorizes a configuration as non-operational if a required resource is missing. For more information, see "Making an XMakefile Configuration Operational" on page 43-31.

User Templates. Set the path of the folder to which you can add template files. Saving templates files with the .mkt extension to this folder adds them to the **Templates** options.

User Configurations. Set the location of configuration files you create with the **New** button.

Make Utility

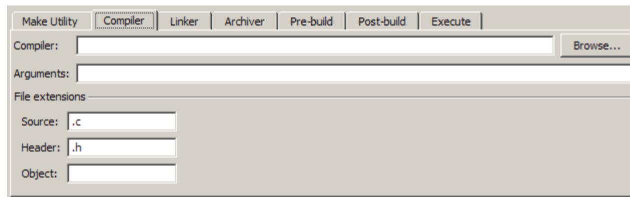


Make utility. Set the path and filename of the make utility executable.

Arguments. Define the command-line arguments to pass to the make utility. For more information, consult the third-party documentation for your make utility.

Optional include. Set the path and file name of an optional makefile to include.

Compiler



Compiler. Set the path and file name of the compiler executable.

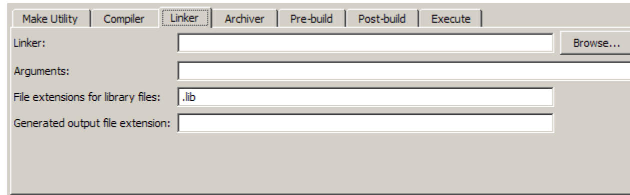
Arguments. Define the command-line arguments to pass to the compiler. For more information, consult the third-party documentation for your compiler.

Source. Define the file name extension for the source files. Use commas to separate multiple file extensions.

Header. Define the file name extension for the header files. Use commas to separate multiple file extensions.

Object. Define the file name extension for the object files.

Linker



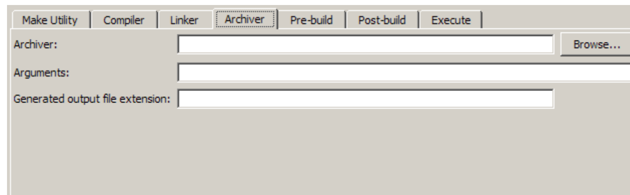
Linker. Set the path and file name of the linker executable.

Arguments. Define the command-line arguments to pass to the linker. For more information, consult the third-party documentation for your linker.

File extensions for library files. Define the file name extension for the file library files. Use commas to separate multiple file extensions.

Generated output file extension. Define the file name extension for the generated libraries or executables.

Archiver



Archiver. Set the path and file name of the archiver executable.

Arguments. Define the command-line arguments to pass to the archiver. For more information, consult the third-party documentation for your archiver.

Generated output file extension. Define the file name extension for the generated libraries.

Pre-build

Enable Prebuild Step. Select this check box to define a prebuild tool that runs before the compiler.

Prebuild tool. Set the path and file name of the prebuild tool executable.

Arguments. Define the command-line arguments to pass to the prebuild tool. For more information, consult the third-party documentation for your prebuild tool.

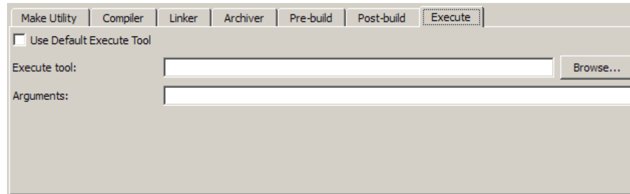
Post-build

Enable Postbuild Step. Select this check box to define a postbuild tool that runs after the compiler or linker.

Postbuild tool. Set the path and file name of the postbuild tool executable.

Arguments. Define the command-line arguments to pass to the postbuild tool. For more information, consult the third-party documentation for your postbuild tool.

Execute



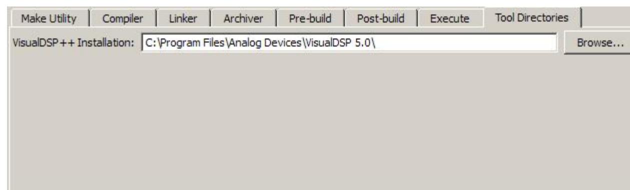
Use Default Execute Tool. Select this check box to use the generated derivative as the execute tool when the build process is complete. Uncheck it to specify a different tool. The default value, `echo`, simply displays a message that the build process is complete.

Note On Linux, multirate multitasking executables require root privileges to schedule POSIX threads with real-time priority. If you are using makefiles to build multirate multitasking executables on your Linux development system, you cannot use **Execute tool** to run the executable. Instead, use the Linux command, `sudo`, to run the executable.

Execute tool. Set the path and file name of the execute tool executable or built-in command.

Arguments. Define the command-line arguments to pass to the execute tool. For more information, consult the third-party documentation for your execute tool.

Tool Directories



Installation. Use the Tool Directories tab to change the toolchain path of an operational configuration.

For example, if you installed two versions of a vendor build tool in separate folders, you can use the **Installation** path to change which one the configuration uses.

Verification and Profiling

- "What Is Verification?" on page 44-2
- "Processor-in-the-Loop (PIL) Simulation" on page 44-3
- "Execution Profiling" on page 44-14
- "Stack Profiling" on page 44-21

Verify and profile generated code executing on processors

What Is Verification?

Verification consists broadly of running generated code on a processor and verifying that the code does what you intend. The coder software provides tools that help you verify your code during development by letting you run portions of simulations on your hardware and profiling the executing code.

Using the Automation Interface and Project Generator components, the coder software offers the following verification functions:

- Processor-in-the-Loop — A technique to help you evaluate how your process runs on your processor.
- Real-Time Task Execution Profiling — A tool that lets you see how the tasks in your process run in real-time on your processor hardware.

Processor-in-the-Loop (PIL) Simulation

In this section...

"Overview" on page 44-3
"Approaches" on page 44-4
"Communications" on page 44-9
"Definitions" on page 44-11
"PIL Issues and Limitations" on page 44-13

Overview

You can use processor-in-the-loop (PIL) simulation to verify your generated code as compiled object code. PIL helps you evaluate the behavior of a candidate algorithm on the target platform selected for the deployment of your application. In PIL simulation, the candidate algorithm runs on the target platform as a part of the Simulink simulation loop.

You can use PIL simulation to verify your generated code on a target processor or instruction set simulator. In PIL simulation, the target processor participates fully in the simulation loop — hence the term processor-in-the-loop simulation. To verify your generated code, you can compare the output of regular simulation modes, such as Normal or Accelerator, with the output of PIL simulation mode. You can easily switch between simulation and PIL modes. This flexibility allows you to verify the generated code by executing the model as compiled code in the target environment. You can model and test your embedded software component in Simulink and then reuse your regression test suites across simulation and compiled object code. This process avoids the time-consuming process of leaving the Simulink software environment to run tests again on object code compiled for the production hardware.

Embedded Coder supports the following PIL approaches:

- Top-model PIL
- PIL block
- Model Block PIL

For more information about PIL, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations"

Processor-in-the-Loop (PIL) builds and uses a MEX function to run the PIL simulation block. Before using PIL, set up a compiler for MATLAB to build the MEX files. Run the command |mex -setup| to select a compiler configuration. For more information, read "Building MEX-Files"

Approaches

Model Block PIL

Use Model block PIL to:

- Verify code generated for referenced models (model reference code interface).
- Provide a test harness model (or a system model) to generate test vector or stimulus inputs.
- Switch a Model block between normal, SIL, or PIL simulation modes.

To perform a model block PIL simulation, start with a top model that contains a Model block. The top model serves as a test harness, providing inputs and outputs for the Model block. The Model block references the model you plan to run on a target. During PIL simulation, the referenced model runs on the target platform.

For more information about using the Model block, see Model Variants and "Referencing a Model".

By default, your MathWorks software uses the IDE debugger for PIL communications with the target platform. To achieve faster communications, you can configure your MathWorks software to use a TCP/IP network connection, as described in "TCP/IP" on page 44-9.

To use model block PIL:

1 Right-click the Model block, and select **ModelReference Parameters**.

2. When the software displays the **Function Block Parameters: Model** dialog box, set **Simulation mode** to `Processor-in-the-loop (PIL)` and click **OK**.

3. Open the Model block.

4. In the referenced model (Model block) Configuration Parameters (**Ctrl+E**), under **Code Generation**>**IDE Link**, set **Build action** set to `Archive_library`. This action avoids a warning when you start the simulation.

5. Add a Target Preferences block to either model, and configure it for the target platform.

6. Copy the Target Preferences block from one model to the other. The top model and the Model block now contain identical Target Preference blocks.

7. Save the changes to both models.

8. In the top model menu bar, select **Simulation** > **Start**. This action builds the referenced model in the Model block, downloads it to your target platform, and runs the PIL simulation.

Note In the top model Configuration Parameters (**Ctrl+E**), under **Code Generation** > **IDE Link**, leave **Build action** set to `Build_and_execute`. Do not change **Build action** to `Create_Processor_In_the_Loop_Project`.

For more information, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations"

Top Model PIL

Use top-model PIL to:

- Verify code generated for a top model (standalone code interface).
- Load test vectors or stimulus inputs from the MATLAB workspace.
- Switch the entire model between normal and SIL or PIL simulation modes.

For more information, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations"

Setting Model Configuration Parameters to Generate the PIL Application. Configure your model to generate the PIL executable from your model:

1. Add a Target Preferences block in to your model. The Target Preferences block is located in the Simulink library browser under Embedded Coder > Embedded Targets.
2. Open the Target Preferences block and select your processor from the list of processors.
3. From the model window, select **Simulation > Configuration Parameters**.
4. In Configuration Parameters, select **Code Generation**.
5. Set **System Target File** to `idelink_ert.tlc`.
6. On the **Select** tree, choose **IDE Link**.
7. Set **Build format** to `Project`.
8. Set **Build action** to `Create_processor_in_the_loop_project`.
9. Click **OK** to close the Configuration Parameters dialog box.

Running the Top Model PIL Application. To create a PIL block, perform the following steps:

1. In the model window menu, select **Simulation > Processor-in-the-loop**.
2. In the model toolbar, click the Start simulation button.

   A new model window opens and the new PIL model block appears in it. The third-party IDE compiles and links the PIL executable file. Follow the progress of the build process in the MATLAB command window.

PIL Block

Use the PIL Block to:

- Use a compiler and target environment supported by the Embedded Coder product.
- Verify code generated for a top model (standalone code interface) or subsystem (right-click build standalone code interface).
- Change the model and insert a PIL block to represent a component running in SIL or PIL mode. The test harness model or a system model provides test vector or stimulus inputs.

For more information, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations".

Preparing Your Model to Generate a PIL Block

Start with a model that contains the algorithm blocks you want to verify on the processor as compiled object code. To create a PIL application and PIL block from your algorithm subsystem, follow these steps:

1. Identify the algorithm blocks to cosimulate.

2. Convert those blocks into an unmasked subsystem in your model.

   For information about how to convert your process to a subsystem, refer to Creating Subsystems in *Using Simulink* or in the online Help system.

3. Open the newly created subsystem and copy a Target Preferences block to it. The Target Preferences block is located in the Simulink library browser under Embedded Coder > Embedded Targets.

   Open the Target Preferences block and select your processor from the list of processors.

Setting Model Configuration Parameters to Generate the PIL Application

After you create your subsystem, set the configuration parameters for your model to enable the model to generate a PIL block.

Configure your model to enable it to generate PIL algorithm code and a PIL block from your subsystem:

1. From the model menu bar, select **Simulation > Configuration Parameters**. This action opens the Configuration Parameters dialog box.
2. On the **Select** tree, choose **Code Generation**.
3. Set **System Target File** to `idelink_ert.tlc`.
4. On the **Select** tree, choose **IDE Link**.
5. Set **Build format** to `Project`.
6. Set **Build action** to `Create_processor_in_the_loop_project`.
7. Click **OK** to close the Configuration Parameters dialog box.

Creating the PIL Block Application from a Model Subsystem

To create a PIL block, perform the following steps:

1. Right-click the masked subsystem in your model and select **Code Generation > Build Subsystem** from the context menu.

   A new model window opens and the new PIL block appears in it. The third-party IDE compiles and links the PIL executable file.

   This step builds the PIL algorithm object code and a PIL block that corresponds to the subsystem, with the same inputs and outputs. Follow the progress of the build process in the MATLAB command window.

2. Copy the new PIL block from the new model to your model. To simulate the subsystem processes concurrently, place it parallel to your masked subsystem. Otherwise, replace the subsystem with the PIL block.

   To see a PIL block in a parallel masked subsystem, search the product help for *Getting Started with Application Development* and select the demo that matches your IDE.

Note Models can have multiple PIL blocks for different subsystems. They cannot have more than one PIL block for the same subsystem. Including multiple PIL blocks for the same subsystem causes errors and incorrect results.

Running Your PIL Application to Perform Simulation and Verification

After you add your PIL block to your model, click **Simulation > Start** to run the PIL simulation and view the results.

Communications

Chose one of the following communication methods for transferring code and data during PIL simulations:

- TCP/IP: Faster. Requires an IP stack on the target.
- Debugger: Slower. Requires a supported IDE. Does not require an IP stack on the target.

TCP/IP

Using TCP/IP with PIL requires an implementation of the Internet Protocol (IP) stack on the target platform. Therefore, you can only use TCP/IP with one of the following targets:

- Linux
- Texas Instruments DSP/BIOS
- Wind River VxWorks

If your target supports TCP/IP, use TCP/IP instead of using an IDE debugger. Using TCP/IP for PIL communications is typically faster than using a debugger, particularly for large data sets, such as with video and audio applications.

It also works well when you build an application on a remote Linux target using the `remoteBuild` function.

You can use TCP/IP with all three PIL approaches:

- Top-model PIL
- Model block PIL
- Subsystem PIL

To enable and configure TCP/IP with PIL:

1 Set up PIL simulation according to the PIL approach you have chosen.

2 At the MATLAB command line, use `setpref` to specify the IP address of the PIL server (`servername`).

   If you are running the PIL server on a remote target, specify the IP address of the target. For example:

```
>> setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','servername','144.212.109.114');
```

   If you are running PIL server locally, on your host Windows or Linux system, enter `'localhost'` instead of an IP address:

```
>> setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','servername','localhost');
```

3 Specify the TCP/IP port number to use for PIL data communication. Use one of the free ports in your system. For example:

```
>> setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','portnum', 17025);
```

4 Enable PIL communications over TCP/IP:

```
>> setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', true);
```

   To disable PIL communications over TCP/IP, change the value to `false`. This action automatically enables PIL communications over an IDE debugger, if an IDE is available.

5 Open the Target Preferences block in your model, then set the **Operating System** parameter to an operating system.

Note You cannot use TCP/IP for PIL when the value of **Operating System** is `None`.

Additional Steps for TI C6000 Processors

Add an IP Config block to the following location in your model:

- For Top Model PIL, add it at the top level of your model.
- For Model Block PIL, add it to the referenced model to which you are pointing.
- For Subsystem PIL, place it in the subsystem.

Configure the IP Config block settings as described in C6000 IP Config.

To determine the IP address assigned to the PIL server on the C6000 target:

1. Enter an arbitrary IP address the first time you specify the IP address.
2. Build and run the code for your model.
3. In the CCS command window, observe the actual IP address assigned to the C6000 processor by the DHCP server.
4. Enter the actual IP address the second time you specify the IP address.

Debugger

If you disable PIL communications over “TCP/IP” on page 44-9, and an IDE is available, PIL uses the IDE debugger to communicate with the application running on target.

To enable PIL communications over an IDE debugger, disable PIL communications over TCP/IP by entering:

```
>> setpref('MathWorks_Embedded_IDE_Link_PIL_Preferences','enabletcpip', false);
```

Definitions

Simulation

The division of model simulation activities between a host and a target. The test harness runs on a host, such as a development workstation, and drives the inputs to the generated code running on the target.

PIL Algorithm

The algorithmic code, which corresponds to a subsystem or portion of a model, to test during the PIL simulation. The PIL algorithm is in compiled object form to enable verification at the object level.

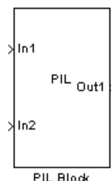
PIL Application

The executable application that runs on the processor platform. The coder software creates a PIL application by augmenting your algorithmic code with the PIL execution framework. The PIL execution framework code compiles as part of your embedded application.

The PIL execution framework code includes the string.h header file so that the PIL application can use the `memcpy` function. The PIL application uses `memcpy` to exchange data between the Simulink model and the simulation processor.

PIL Block

When you build a subsystem from a model for PIL, the process creates a PIL block optimized for PIL simulation. When you run the simulation, the PIL block acts as the interface between the model and the PIL application running on the processor. The PIL block inherits the shape and signal names from the source subsystem in your model, as shown in the following example. Inheritance is convenient for copying the PIL block into the model to replace the original subsystem for simulation.



PIL Issues and Limitations

Consider the following issues when you work with PIL blocks.

Constraints

When using PIL in your models, keep the following constraints in mind:

- A model can contain a single subsystem PIL block.
- A model can contain a single model block running PIL mode.
- A model can contain a subsystem PIL block or a model block in PIL mode, but not both.

Generic PIL Issues

Refer to the Support Table section in the Embedded Coder documentation for general information about using the PIL block with embedded link products. Refer to PIL Feature Support and Limitations.

With Texas Instruments CCS, PIL with DSP/BIOS Enabled Does Not Support System Stack Profiling

Enabling DSP/BIOS for Texas Instruments™ processors disables the stack profiling option. To use stack profiling with PIL, disable DSP/BIOS™ in the model Target Preferences block and rebuild your project.

Simulink Coder grt.tlc-Based Targets Not Supported

PIL does not support `grt.tlc` system target files.

To use PIL, set **System target file** in the Configuration Parameters > Code Generation pane to `idelink_ert.tlc`.

Execution Profiling During PIL Simulation and Standalone Execution Not Supported for Desktop Targets

Execution profiling is not supported for Desktop Targets during PIL simulations and during Standalone Exection. For more information, see the "Execution Profiling" on page 44-14 section in the Embedded Coder product documentation.

Execution Profiling

In this section...

"What Is Execution Profiling?" on page 44-14

"Execution Profiling during Standalone Execution Mode" on page 44-15

"Execution Profiling during PIL Simulation" on page 44-19

What Is Execution Profiling?

You can measure the execution performance of the generated code running on your target processor with Execution Profiler. This feature can be used to measure CPU utilization during standalone target processor execution or processor-in-the-loop (PIL) simulation.

Your software includes a set of utilities for profiling execution of generated code on a target. The utilities profile execution times for synchronous tasks, asynchronous tasks, and atomic subsystems. You can perform execution profiling during standalone execution or processor-in-the-loop (PIL) simulation. The following table summarizes execution profiler applicability.

Execution Mode	Desktop IDEs and Desktop Targets	Embedded IDEs and Embedded Targets
Standalone Execution	N/A	Tasks (Synchronous and Asynchronous) or Atomic Subsystems
PIL Simulation	N/A	Tasks (Synchronous only)

Note To profile by atomic subsystems, your model must include at least one atomic subsystem. To learn more about creating atomic subsystems, refer to "Creating Subsystems" in the online help for Simulink software.

Execution Profiling during Standalone Execution Mode

In standalone execution mode, instrumented code in the generated code collects a user-specified number of execution time samples and stores in target processor memory. Once target processor execution is halted the profile function can be called to transfer profiling data from target processor memory to the MATLAB workspace for viewing and analysis.

The following Targets do not support execution profiling in standalone mode:

- VxWorks Target
- Windows Target
- Linux Target

By Tasks

To configure a model to use task execution profiling, perform the following steps:

1. Ensure your model is configured with Target Preferences block.
2. Open the Configuration Parameters dialog box for your model.
3. Select **IDE Link** from the Select tree.
4. Set **Build format** to `Project` and set **Build action** to `Build_and_execute`.
5. Select **Profile real-time execution**.
6. In the **Profile by** list, select **Tasks** to enable profiling by tasks.
7. Set the **Number of profiling samples to collect**. This is number of execution sample times to be measured.
8. By default, the **Export IDE link handle to base workspace** is enabled, and the **IDE link handle name** is set to `IDE_Obj`.
9. Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

1 Click **Incremental build** on the model toolbar to generate, build, load, and run your code on the processor.

2 To stop the running program, select **Debug > Halt** in the IDE or use `IDE_obj.halt` from the MATLAB command prompt. Gathering profiling data from a running program may yield incorrect results.

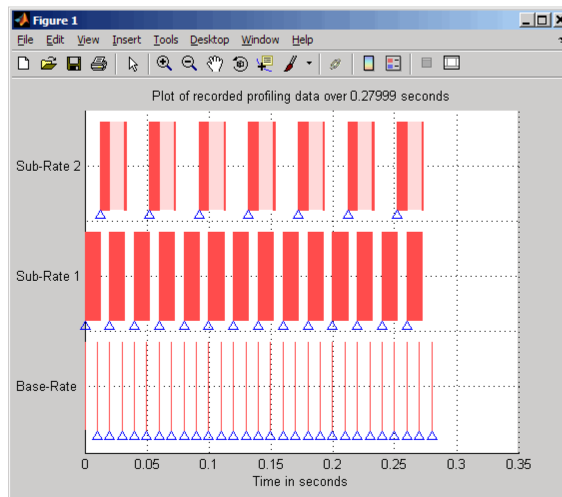
3 At the MATLAB command prompt, enter

```
profile(IDE_Obj, execution , report )
```

to view the MATLAB software graphic of the execution report and the HTML execution report.

For more information about other reporting options, see the product help for the `profile` function.

The following figure shows the profiling plot from running an application that has three rates—the base rate and two slower rates. The gaps in the Sub-Rate2 task bars indicate preempted operations.



By Subsystems

When your models use atomic subsystems, you have the option of profiling your code based on the subsystems.

To configure a model to use subsystem execution profiling, perform the following steps:

1. Ensure your model is configured with a Target Preferences block.
2. Open the Configuration Parameters dialog box for your model.
3. Select **IDE Link** from the Select tree.
4. Set **Build format** to `Project` and set **Build action** to `Build_and_execute`.
5. Select **Profile real-time execution**.
6. In the **Profile by** list, select `Atomic subsystems` to enable profiling by atomic subsystems.
7. Set the **Number of profiling samples to collect**. This is number of execution sample times to be measured.
8. By default, **Export IDE link handle to base workspace** is enabled, and the IDE link handle name is set to `IDE_Obj`.
9. Click **OK** to close the Configuration Parameters dialog box.

To view the execution profile for your model:

1. Click **Incremental build** on the model toolbar to generate, build, load, and run your code on the processor.
2. To stop the running program, select Debug > Halt in the IDE or use `IDE_obj.halt` from the MATLAB command prompt. Gathering profiling data from a running program may yield incorrect results.
3. At the MATLAB command prompt, enter:

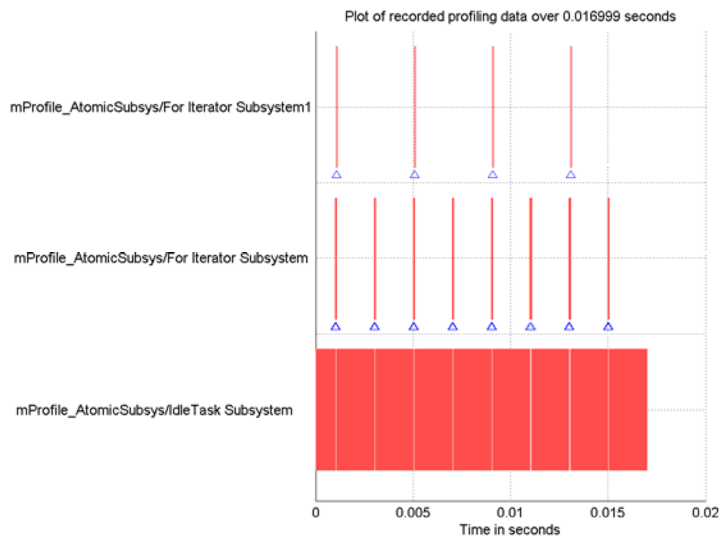
   At the MATLAB command prompt, enter:

   ```
   profile(IDE_Obj,  execution , report )
   ```

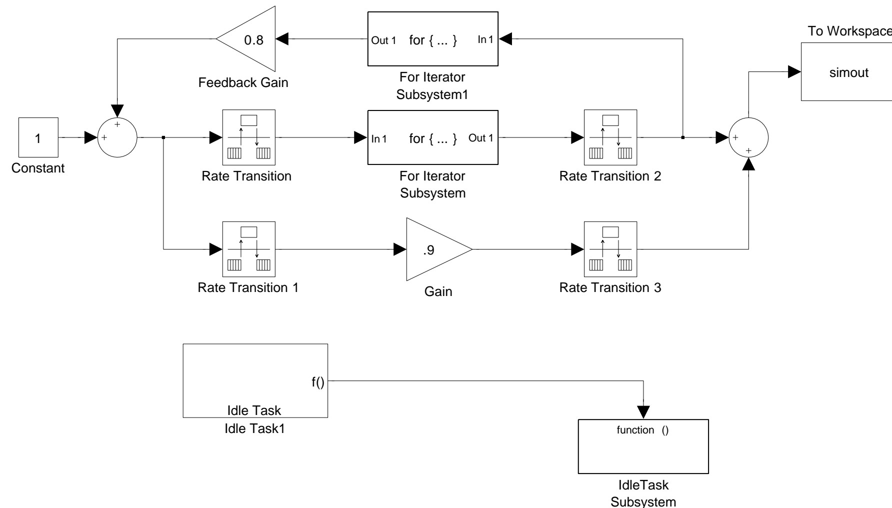
to view the MATLAB software graphic of the execution report and the HTML execution report.

For more information, see the product help for the `profile` function.

The following figure shows the profiling plot from running an application that has three subsystems—For Iterator Subsystem, For Iterator Subsystem1, and Idle Task Subsystem.



The following figure presents the model that contains the subsystems reported in the profiling plot.



Execution Profiling during PIL Simulation

You can also invoke Execution Profiler during processor-in-the-loop (PIL) simulation to measure CPU utilization of each synchronous task. Once PIL simulation is halted, the collected profiler statistics is available in the MATLAB workspace for viewing and analysis.

Configure the model for PIL simulation as described in "Processor-in-the-Loop (PIL) Simulation" on page 44-3.

Then perform the following steps:

1 Select Simulation > Configuration Parameters > Code Generation > SIL and PIL Verification.

2 Select the **Collect execution time measurements** check box.

3 In the Workspace edit box, specify a valid MATLAB variable name. When you run the simulation, the software generates a variable with this name.

The variable contains the profiler statistics, and is an object of type `rtw.pil.ExecutionProfile`.

4. Click **OK**. You can view or analyze the measurements stored in the workspace in MATLAB.

See "Viewing and Analyzing Code Execution Profiles" in Embedded Coder documentation.

Stack Profiling

In this section...

"What is Stack Profiling?" on page 44-21

"Profiling System Stack Use" on page 44-22

What is Stack Profiling?

The coder software enables you to determine how your application uses the processor system stack. This is available only on systems that do not use operating systems. Using the `profile` method, you can initialize and test the size and usage of the stack. This information can help you optimize both the size of the stack and how your code uses the stack.

To provide stack profiling, `profile` writes a known pattern to the addresses in the stack. After you run your application for a while, and then stop your application, `profile` examines the contents of the stack addresses. `profile` counts each address that no longer contains the known pattern as used. The total number of address that have been used, compared to the total number of addresses you allocated, becomes the stack usage profile. This profile process does not tell you how often any address was changed by your application.

You can profile the stack with both the manually written code in a project and the code you generate from a model.

When you use `profile` to initialize and test the stack operation, the software returns a report that contains information about stack size, usage, addresses, and direction. With this information, you can modify your code to use the stack efficiently. The following program listing shows the stack usage results from running an application on a simulator.

```
profile(IDE_Obj,'stack','report')

Maximum stack usage:

System Stack: 532/1024 (51.95%) MAUs used.
```

```
             name: System Stack
     startAddress: [512    0]
       endAddress: [1535    0]
        stackSize: 1024 MAUs
  growthDirection: ascending
```

The following table describes the entries in the report:

Report Entry	Units	Description
`System Stack`	Minimum Addressable Unit (MAU)	Maximum number of MAUs used and the total MAUs allocated for the stack.
`name`	String for the stack name	Lists the name assigned to the stack.
`startAddress`	Decimal address and page	Lists the address of the stack start and the memory page.
`endAddress`	Decimal address and page	Lists the address of the end of the stack and the memory page.
`stackSize`	Addresses	Reports number of address locations, in MAUs, allocated for the stack.
`growthDirection`	Not applicable	Reports whether the stack grows from the lower address to the higher address (`ascending`) or from higher to lower (`descending`).

Profiling System Stack Use

To profile the system stack operation, perform these tasks in order:

1. Load an application.
2. Set up the stack to enable profiling.
3. Run your application.
4. Request the stack profile information.

Follow these steps to profile the stack as your application interacts with it. This particular example uses a IDE handle object, `IDE_Obj`, for Texas Instruments' Code Composer Studio. However, you can generalize from this example to any IDE that supports profiling.

1. Load the application to profile.
2. Use the `profile` method with the **`setup`** input keyword to initialize the stack to a known state.

   ```
   profile(IDE_Obj,'stack','setup')
   ```

   With the **`setup`** input argument, `profile` writes a known pattern into the addresses that compose the stack. For example, the pattern for C6000 processors is A5, and for C2000 and C5000 processors is A5A5 (to account for their address size). As long as your target application does not write the same pattern to the system stack, `profile` can report the stack usage correctly.
3. Run your application.
4. Stop your running application. Stack use results gathered from an application that is running may be incorrect.
5. Use the `profile` method to capture and view the results of profiling the stack.

   ```
   profile(IDE_Obj,'stack','report')
   ```

The following example demonstrates setting up and profiling the stack. The IDE handle object, `IDE_Obj`, must exist in your MATLAB workspace and your application must be loaded on your processor. This example comes from a TI C6713 simulator.

```
profile(IDE_Obj,'stack','setup')  % Set up processor stack--write A5 to t

Maximum stack usage:

System Stack: 0/1024 (0%) MAUs used.


           name: System Stack
   startAddress: [512    0]
     endAddress: [1535    0]
      stackSize: 1024 MAUs
growthDirection: ascending


run(IDE_Obj)
halt(IDE_Obj)
profile(IDE_Obj,'stack','report') % Request stack use report.

Maximum stack usage:

System Stack: 356/1024 (34.77%) MAUs used.


           name: System Stack
   startAddress: [512    0]
     endAddress: [1535    0]
      stackSize: 1024 MAUs
growthDirection: ascending
```

45

Processor-Specific Optimizations

Target Function Library (TFL)

In this section...

"About Target Function Libraries and Optimization" on page 45-2

"Using a Processor-Specific Target Function Library to Optimize Code" on page 45-4

"Process of Determining Optimization Effects Using Real-Time Profiling Capability" on page 45-5

"Reviewing Processor-Specific Target Function Library Changes in Generated Code" on page 45-5

"Reviewing Target Function Library Operators and Functions" on page 45-8

"Creating Your Own Target Function Library" on page 45-8

About Target Function Libraries and Optimization

A *target function library* is a set of one or more function tables that define processor- and compiler-specific implementations of functions and arithmetic operators. The code generation process uses these tables when it generates code from your Simulink model.

The coder software registers processor-specific target function libraries during installation. To use one of the libraries, select the set of tables that correspond to functions implemented by intrinsics or assembly code for your processor from the **Target function library** list in the model configuration parameters. To do this, complete the following steps:

1. In your model, select **Simulation** > **Configuration Parameters**.
2. In the Configuration Parameters dialog box, select **Code Generation** and **Interface**.
3. Set the **Target function library** parameter to the appropriate library for your processor.

After you select the processor-specific library, the model build process uses the library contents to optimize generated code for that processor. The generated code includes processor-specific implementations for `sum`, `sub`, `mult`,

div, and various functions, such as tan or abs, instead of the default ANSI C instructions and functions. The optimized code enables your embedded application to run more efficiently and quickly, and in many cases, reduces the size of the code. For more information about target function libraries, refer to "Introduction to Target Function Libraries" on page 31-2 in the Embedded Coder documentation.

For a list of supported TFLs, see "Reviewing Target Function Library Operators and Functions" on page 45-8

Code Generation Using the Target Function Library

The build process begins by converting your model and its configuration set to an intermediate form that reflects the blocks and configurations in the model. Then the code generation phase starts.

During code generation for your model, the following process occurs:

1. Code generation encounters a call site for a function or arithmetic operator and creates and partially populates a target function library entry object.

2. The entry object queries the target function library database for an equivalent math function or operator. The information provided by the code generation process for the entry object includes the function or operator key, and the conceptual argument list.

3. The code generation process passes the target function library entry object to the target function library.

4. If there is a matching table entry in the target function library, the query returns a fully-populated target function library entry to the call site, including the implementation function name, argument list, and build information

5. The code generation process uses the returned information to generate code.

Within the target function library that you select for your model, the software searches the tables that comprise the library. The search occurs in the order in which the tables appear in either the Target Function Library Viewer or the **Target function library** tool tip. For each table searched, if the search finds multiple matches for a target function library entry object, priority

level determines the match to return. The search returns the higher-priority (lower-numbered) entry.

For more information about target function libraries in the build process, refer to "Introduction to Target Function Libraries" on page 31-2 in the Embedded Coder documentation.

Using a Processor-Specific Target Function Library to Optimize Code

As a best practice, you should select the appropriate target function library for your processor after you verify the ANSI C implementation of your project.

Perform the following steps to select the target function library for your processor:

1. Select **Simulation > Configuration Parameters** from the model menu bar. The Configuration Parameters dialog box for your model opens.

2. On the **Select** tree in the Configuration Parameters dialog box, choose **Code Generation**.

3. Use **Browse** to select `idelink_ert.tlc` as the **System target file**.

4. On the **Select** tree, choose **Interface**.

5. On the **Target function library** list, select the processor family that matches your processor. Then, click **OK** to save your changes and close the dialog box.

With the target function library selected, your generated code uses the specific functions in the library for your processor.

To stop using a processor-specific target function library, open the `Interface` pane in the model configuration parameters. Then, select the `C89/C90 (ANSI)` library from the **Target function library** list.

Process of Determining Optimization Effects Using Real-Time Profiling Capability

You can use the real-time profiling capability to examine the results of applying the processor-specific library functions and operators to your generated code. After you select a processor-specific target function library, use the real-time execution profiling capability to examine the change in program execution time.

Use the following process to evaluate the effects of applying a processor-specific target function library when you generate code:

1. Enable real-time profiling in your model. Refer to "Execution Profiling" on page 44-14.
2. Generate code for your project using the default target function library `C89/C90 ANSI`.
3. Profile the code, and save the report.
4. Rebuild your project using a processor-specific target function library instead of the `C89/C90 ANSI` library.
5. Profile the code, and save the second report.
6. Compare the profile report from running your application with the processor-specific library selected to the profile results with the ANSI library selected in the first report.

For a demonstration of verifying the code optimization, search the product help for the "Optimizing Embedded Code via Target Function Library" and select the demo that matches your IDE.

Reviewing Processor-Specific Target Function Library Changes in Generated Code

Use one of the following techniques or tools to see the target function library elements where they appear in the generated code:

- "Reviewing Code Manually" on page 45-6.

- "Using Model-to-Code Tracing" on page 45-6 to navigate from blocks in your model to the code generated from the block.
- "Using a File Differencing Scheme" on page 45-7 to compare projects that you generate before and after you select a processor-specific target function library.

Reviewing Code Manually

To see where the generated code uses target function library replacements, review the file *modelname*.c . Look for code similar to the following examples.

For example, with CCS:

```
codeopt_tfl_B.Sum6[j] = ldexp((real_T)Sum[j], -11) + ldexp((real_T)
        c62x_mul_s32_s32_s32_sr_sat(codeopt_tfl_P.Gain5_Gain, UnitDelay[j], 6),
```

For example, with MULTI:

```
j = mul_s32_s32_s32_sr6_sat(codeopt_tfl_P.Gain1_Gain, rtb_SineWave[i]);
      tmp_0 = mul_s32_s32_s32_sr6_sat(codeopt_tfl_P.Gain2_Gain, rtb_UnitDelay[i]);
      tmp_1 = j + tmp_0;
```

For example, with VisualDSP++:

```
codeopt_tfl_B.UnitDelay3[j] = sharc_mul_s32_s32_s32_sr_sat
        (codeopt_tfl_P.Gain4_Gain, codeopt_tfl_B.UnitDelay2[j], 6);
```

The functions shown are the multiply implementation functions registered in the target function library. In these examples, the function performs an optimized multiplication operation. Similar functions appear for `add`, and `sub`. For more information about the arguments in the function, refer to "Introduction to Target Function Libraries" on page 31-2 in the online Help system.

Using Model-to-Code Tracing

You can use the model-to-code report options in the configuration parameters to trace the code generated from any block with target function library. After you create your model and select a target function library, follow these steps to use the report options to trace the generated code:

1 Open the model configuration parameters.

2 Select `Report` from the **Select** tree.

3 In the `Report` pane, select **Create code generation report** and **Model-to-code**, and then save your changes.

4 Press **Ctrl+B** to generate code from your model.

   The Code Generation Report window opens on your desktop. For more information about the report, refer to the Chapter 20, "Generating Reports for Code Reviews and Traceability Analysis" topic in the Embedded Coder documentation.

5 Use model-to-code highlighting to trace the code generated for each block with target function library applied:

   - Right-click on a block in your model and select **Code Generation > Navigate to code** from the context menu.
   - Select **Navigate-to-code** to highlight the code generated from the block in the report window.

   Inspect the code to see the target function operator in the generated code. For more information, refer to "Tracing Code Generated Using Your Target Function Library" on page 31-143 in the Embedded Coder documentation in the online Help system.

If a target function library replacement did not occur as you expected, use the techniques described in "Examining and Validating Function Replacement Tables" on page 31-139 in the Embedded Coder documentation to help you determine why the build process did not use the function or operator.

Using a File Differencing Scheme

You can also review the target function library induced changes in your project by comparing projects that you generate both with and without the processor-specific target function library.

1 Generate your project with the default `C89/C90 ANSI` target function library. Use `Create Project`, `Archive Library`, or `Build` for the **Build action** in the `IDE Link` options.

2 Save the project to a new name—*newproject1*.

3 Go back to the configuration parameters for your model, and select a target function library appropriate for your processor.

4 Regenerate your project.

5 Save the project with a new name—*newproject2*

6 Compare the contents of the *modelname*.c files from newproject1 and newproject2. The differences between the files show the target function library induced code changes.

Reviewing Target Function Library Operators and Functions

Embedded Coder software provides the Target Function Library viewer to enable you to review the arithmetic operators and functions registered in target function library tables.

To open the viewer, enter the following command at the MATLAB prompt.

```
RTW.viewTfl
```

For details about using the target function library viewer, refer to “Selecting and Viewing Target Function Libraries” in the online Help system.

Creating Your Own Target Function Library

For details about creating your own library, refer to the following sections in your Embedded Coder documentation:

- “Introduction to Target Function Libraries” on page 31-2
- “Creating Function Replacement Tables” on page 31-16
- “Examining and Validating Function Replacement Tables” on page 31-139

Working with Altium TASKING IDE

- "Getting Started" on page 46-2
- "Components" on page 46-27
- "Verification" on page 46-50
- "Optimization" on page 46-69
- "Tutorials" on page 46-75
- "Code Generation Pane — IDE Link" on page 46-83
- "Limitations and Tips" on page 46-95

Note Support for the Altium TASKING integrated development environment will be removed in a future release of your MathWorks software.

Note The information in this chapter describes Embedded Coder features and user interfaces that are unique to Altium TASKING. Do not generalize this information to Embedded Coder support for other IDEs.

Getting Started

In this section...

"Overview" on page 46-2
"Supported Altium TASKING Toolsets" on page 46-6
"Using This Guide" on page 46-7
"Setting Target Preferences for Altium TASKING" on page 46-8
"Working with Configuration Sets" on page 46-13
"Accessing Utilities for TASKING" on page 46-20
"Option Sets" on page 46-24

Overview

- "Introduction" on page 46-2
- "Project Generator" on page 46-3
- "Automation Interface" on page 46-4
- "Verification" on page 46-4
- "Optimization" on page 46-5

Introduction

Note Support for the Altium TASKING integrated development environment will be removed in a future release of your MathWorks software.

Embedded Coder software lets you build, test, and verify automatically generated code using the MATLAB, Simulink, and Simulink Coder products, and the Altium TASKING integrated development environment. Embedded Coder software makes it easy to verify code executing within the TASKING environment using a test harness model in Simulink. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by the Embedded Coder product. A wide range of DSPs

and 8-, 16- and 32-bit microprocessors and microcontrollers are supported including devices from the Infineon, Renesas®, and Freescale product families. Embedded Coder software provides customizable templates for configuring hardware variants, automating MISRA C code checking, and controlling the build process.

With Embedded Coder software, you can use MATLAB and Simulink to interactively analyze, profile and debug target-specific execution behavior within TASKING software. In this way, Embedded Coder software automates deployment of the complete embedded software application and makes it easy for you to assess possible differences between the results from the model simulation and the results from code running on the target.

Embedded Coder software consists of a Project Generator component, an Automation Interface component, and features for code verification and optimization. The following sections summarize these components and features.

Project Generator

- Automated project-based build process

  Automatically create and build projects for code generated by the Simulink Coder or Embedded Coder products.

- Highly customizable code generation

  Use System Target Files (STF) to generate target-specific and optimized code.

- Highly customizable build process

  Support for multiple TASKING Toolsets provides a route to a large number of different target hardware platforms. Further customization is possible by using custom project templates, giving access to all options supported by the TASKING Toolset.

- Automated download and debugging

  Rapidly and effortlessly debug generated code in the CrossView Pro debugger, using either the instruction set simulator or real hardware.

Automation Interface

- MATLAB API for TASKING EDE (IDE)

  Automate complex tasks in the TASKING EDE by writing MATLAB scripts to communicate with the EDE.

  For example, you could

  - Automate project creation, including adding source files, include paths, and preprocessor defines.
  - Configure batch building of projects.
  - Launch a debugging session.
  - Execute CodeWright API Library commands.

- MATLAB API for TASKING CrossView Pro (Debugger)

  Automate complex tasks in the TASKING CrossView Pro debugger by writing MATLAB scripts to communicate with the CrossView Pro application, or debug and analyze interactively in a live MATLAB session.

  For example, you could

  - Automate debugging by executing commands from the powerful CrossView Pro command language.
  - Exchange data between MATLAB and the target running in the CrossView Pro application.
  - Set breakpoints, step through code, set parameters and retrieve profiling reports

Verification

- Processor-in-the-loop (PIL) simulation

  Use simulation techniques to verify generated code running in an instruction set simulator or real target environment.

- C Code Coverage

  Use C code instruction coverage metrics from the CrossView Pro instruction set simulator during PIL simulation to refine test cases.

- Execution Profiling

  Use execution profiling metrics from the CrossView Pro instruction set simulator during PIL simulation to establish the timing requirements of your algorithm.

- Stack Profiling

  Use stack profiling metrics for PIL simulation or real-time applications to verify the amount of memory allocated for stack usage is sufficient.

- Bi-Directional Traceability Between Model and Code

  Navigate to the generated code for a given Simulink block or, vice versa, to the Simulink block corresponding to a section of generated code.

- MISRA Checker

  Use the TASKING compiler generated MISRA report to check for an appropriate level of MISRA compliance for your application.

Optimization

- Compiler / Linker Optimization Settings

  Use Template Projects to fully control compiler and linker optimization settings.

- Target Memory Placement / Mapping

  Use Template Projects to fully configure the target memory map.

- Execution Profiling

  Use execution profiling metrics from the CrossView Pro instruction set simulator during PIL simulation to guide optimization of your algorithms.

- Stack Profiling

  Use stack profiling metrics for PIL simulation or real-time applications to optimize the amount of stack memory required for an application.

Supported Altium TASKING Toolsets

Supported Versions

Embedded Coder software includes at least one reference template project for each supported toolset. The reference projects were created for specific versions of each Altium TASKING toolset and were used by MathWorks for qualification testing. The supported toolset versions are:

- Infineon® TriCore®: TASKING VX-toolset for TriCore v2.5 r2

  See also "Regenerate Template Projects to Use Selected Toolset Versions" on page 46-7.

- Infineon C166: TASKING Tools for C166/ST10 v8.7 r1
- Renesas M16C: TASKING Tools for M16C v3.1 r1 patch 2
- ARM: TASKING C Compiler for ARM v2.0 r2

  Simulator only, see "On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware" on page 46-98.

- Freescale DSP563xx: TASKING Tools for DSP563xx v3.5 r3 patch 2
- 8051: TASKING Tools for 8051 v7.2 r1

The Renesas R8C family is supported by the Renesas M16C TASKING Toolset.

The Freescale DSP566xx family is supported by the Freescale DSP563xx Toolset.

Support for Other Versions

For minor release increments it may be sufficient to create new default template projects. To do this,

1. Specify the location of your TASKING toolset in the Target Preferences (see "Setting Target Preferences for Altium TASKING" on page 46-8).
2. Close all projects/project spaces in the EDE, and close the EDE.
3. Move to a clean work folder or clean out the existing one.

4. Run the `tasking_generate_templates` command. You must specify your configuration description string, e.g.:

```
tasking_generate_templates('C166', true)
```

   or

```
tasking_generate_templates('TriCore', true)
```

Note Make sure you check the Product Support webpage for the latest information about toolchains qualified with the product. You may be able to obtain patches in order to use other toolsets.

Regenerate Template Projects to Use Selected Toolset Versions. The following toolsets should work after regenerating the template projects:

- TASKING VX-toolset for TriCore and PCP v2.5 r2
- As TASKING VX-toolset for TriCore v2.5 r2 but without On-Chip Debug Support (OCDS)
- "TASKING C/C++ Compiler for ARM v2.0 r2"

Some TASKING packages do not include On-Chip Debug Support (OCDS). For example, "TASKING C/C++, CrossView Pro SIM" does not include OCDS support, but "TASKING VX-Toolset" does. To use a package without OCDS support you must regenerate the template projects as previously described.

Using This Guide

To get started with Embedded Coder software:

1. Follow the instructions in "Setting Target Preferences for Altium TASKING" on page 46-8.
2. After you set target preferences, follow the instructions in "Working with Configuration Sets" on page 46-13 to see how to set up configurations using an example model.
3. Try the demos to gain experience using Embedded Coder software.

4 See “Accessing Utilities for TASKING” on page 46-20 for a quick guide to the functionality available in the menus, with links to more information.

See the following chapters to learn about Embedded Coder software features:

- “Components” on page 46-27 explains the Embedded Coder software components: the Project Generator build process, and the Automation Interface objects.
- “Verification” on page 46-50 describes how to use PIL simulation and other product features for verification.
- “Optimization” on page 46-69 describes how to use product features for optimization.
- “Tutorials” on page 46-75 contains instructions to show you how to create new configurations and template projects, how to use Embedded Coder software with existing models, and how to use different build actions.

Setting Target Preferences for Altium TASKING

This information only applies to using Embedded Coder with the Altium TASKING IDE. To set target preferences when you are using other IDEs, see “Target Preferences” on page 43-4.

Procedure

You must configure your target preferences to use Embedded Coder software.

Note Target preferences are persistent across MATLAB sessions. If you have used a previous version of Embedded Coder software, click **Reset to Default** before setting up your new preferences, to ensure you use the latest values for all fields.

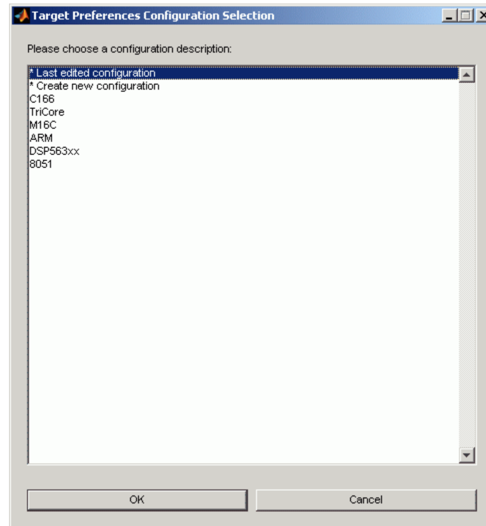
---

1 Enter `taskingutils` in the Command Window.

   The IDE Link Utilities for Use with TASKING dialog box appears.

2 Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.



3 Select or create a configuration:

- Choose a predefined configuration from the list that matches your target.
- Alternatively, select `Create new configuration` to create a new configuration, and click **OK**. For new configurations, see the tutorial section "Creating a New Configuration" on page 46-80.

The Target Preferences dialog box appears. You can use this dialog box to configure the location of your Altium TASKING toolchain executable and other files.

4 Click the plus to expand `Configuration Options`. Similarly, expand `CrossView_Pro_Configuration` and `EDE_Configuration`. This example is set up for the `Infineon C166 Simulator` configuration.

5 Replace the string `<ENTER_TASKING_PATH>` to complete the path to the `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`. See the next section, "Target Preference Fields" on page 46-10, for details on each field. The following example is set up for the `Infineon TriCore Simulator` configuration.

If you have multiple configurations, you have to set them up in your target preferences only once, and then it is simple to switch between them. See the tutorial example "Working with Configuration Sets" on page 46-13.

6 Click **OK** to dismiss the IDE Link Target Preferences dialog box.

The next section explains each target preference field.

Target Preference Fields

Enter `tasking_edit_prefs` in the Command Window.

- `Configuration`

  Select a configuration from the drop-down list. There are preconfigured configurations for

  - C166
  - TriCore
  - M16C
  - ARM
  - DSP563xx
  - 8051

  If you have multiple configurations, you have to set them up in your target preferences only once, and then it is simple to switch between them. You can switch between them using this target preference field.

  Select a free configuration number to set up a new configuration from scratch. See "Creating a New Configuration" on page 46-80.

- `Configuration_Description`

  The title of the configuration. After it is created, this title is the name that appears in the **Target Preferences Configuration** drop-down list in the Configuration Parameters dialog box. Edit this field to change the name of the configuration. These names are predefined for the preconfigured configurations. For a new configuration enter a descriptive name (do not include spaces).

- `CrossView_Pro_Executable`

Enter the full path to your TASKING CrossView Pro installation to replace the string `<ENTER_TASKING_PATH>`. For example, for `Configuration_1` for `Infineon C166 Simulator`:

`D:\Applications\TASKING\c166\bin\xfw166.exe`

- `Initialization`

  This setting determines what the CrossView Pro Debugger executes when it first starts. There are three options.

  - `Use .st Initialization_File` This option is the default setting. "`.st`" files are in an internal file format used by MathWorks to provide initialization options to CrossView Pro software during debugger start up. For example, a `.st` file may specify a CrossView Pro configuration file (`.cfg`) and target type for CrossView Pro to use. Each of the option sets shipped with Embedded Coder software specifies a corresponding `.st` file. For example, the `c166_sim` option set specifies the `c166_default.st` file, which includes basic initialization commands for the C166 CrossView Pro Simulator. See "Option Sets" on page 46-24 for related information. To customize your CrossView Pro configuration, you should use one of the `.ini` initialization options.

  - `Use .ini Initialization_File` Use this option if you have a custom `.ini` initialization file. The file should be a valid CrossView Pro initialization file for your custom configuration. Refer to your CrossView Pro application documentation for details.

  - `Use CrossView Pro Default .ini File` Use this option if you want to run CrossView Pro Default `.ini` file when launching the CrossView Pro Debugger. When launching CrossView Pro software you may be prompted to make configuration selections. Refer to your CrossView Pro application documentation to find the location of this `.ini` file, and for details of CrossView Pro initialization files.

- `Initialization_File`

  Full path of the initialization file corresponding to the `Initialization` field.

- `DOL_File`

  The full path to the TASKING EDE DOL file. For example, the `Infineon_C166_Simulator` Configuration has the `<ENTER_TASKING`

PATH>_\etc\c166.dol as the dol file. You need to replace <ENTER_TASKING_PATH> with your real TASKING installation path.

- EDE_Executable

  Enter the full path to your TASKING EDE installation to replace the string <ENTER_TASKING_PATH>. For example, for Configuration_1 for Infineon C166 Simulator, enter

  D:\Applications\TASKING\c166\bin\ede.exe

- Target_Project_Space

  When you build models, new projects in the TASKING EDE will be created. These projects belong to the project space defined in this entry. The default setting is $(DEFAULT_LOCATION)\projspace.psp. The code generation process expands the $(DEFAULT_LOCATION) token based on the build folder of the model, the model name, and model configuration settings, including the name of the template application project. You should avoid changing this default setting.

- Template_Application_Project

  When you build a Simulink model with Embedded Coder software, the generated projects for your application in the TASKING EDE have the same project settings as the template application project. This template project provides a centric place to manage the project options (e.g., compiler settings, linker settings, etc.) your Simulink models use during code generation. You can modify the project settings of the default template projects or create new ones. See “Accessing Utilities for TASKING” on page 46-20 for information on creating or opening template projects, and see “Tutorial: Creating New Template Projects” on page 46-76.

- Template_Library_Project

  The same as the Template_Application_Project field, but this is applicable for Library projects.

- Use_State_File

  Opens the TASKING EDE in its last saved state. For more information, refer to your TASKING EDE documentation.

Working with Configuration Sets

- "Adding the Embedded Coder Configuration Set Component" on page 46-13
- "Configuration Set Options" on page 46-13
- "Using Configuration Sets to Specify Your Target" on page 46-16
- "Setting Build Action" on page 46-18

Adding the Embedded Coder Configuration Set Component

To add Embedded Coder configuration options to a model, select the menu item **Tools > IDE Link > Add IDE Link Configuration to Model**.

Similarly, you can use the menu item **Remove IDE Link Configuration from Model** to remove the configuration set component.

The following sections explain how to use the configuration set component.

See also "Setting Up Configuration Sets" in the Simulink documentation for more information.

Configuration Set Options

To see Embedded Coder configuration options, navigate to the configuration parameters by any of the following paths:

- **Simulation > Configuration Parameters** in a model
- **Tools > IDE Link > Options** in a model
- **View > Model Explorer** in a model

Click **IDE Link** to see the following options.

The following options are available under **Build Configuration**:

- **Build action**

  Set what action to take after the Simulink Coder build process. You can create application and library projects in the Altium TASKING EDE and

then stop, or you can also choose to build, execute, or debug. See "Setting Build Action" on page 46-18 for more details.

- **Target Preferences Configuration**

  Select target preference configurations. The names correspond to the `Configuration Description` for each configuration in the Target Preferences dialog box. Click **Edit Configuration** to open the Target Preferences dialog box for the currently selected configuration. See "Using Configuration Sets to Specify Your Target" on page 46-16.

- **Add build subdirectory suffix**

  Select the check box to specify a model-specific suffix to be added the regular Simulink Coder build folder suffix. This setting is useful to avoid shared utility function code generation errors which occur because of conflicts over Simulink Coder utility functions shared between different models.

  Clear this check box to use the default Simulink Coder build folder suffix. Not using an additional suffix may result in rebuilding shared libraries unnecessarily. See “Shared Libraries” on page 46-31 and particularly “Supporting Multiple Shared Utility Function Locations: Build Folder Suffix” on page 46-32 for details.

- **Build subdirectory suffix**

  Enter in the edit box a model-specific suffix to be added the regular Simulink Coder build folder suffix.

The following options are available under **Export Handles**:

- **Export EDE handle to MATLAB base workspace**

  Select this check box to export the EDE object handle to the workspace.

- **EDE handle name**

  Enter a MATLAB variable name for the exported handle.

- **Export CrossView Pro handle to MATLAB base workspace**

  Select this check box to export the CrossView Pro object handle to the workspace.

- **CrossView Pro handle name**

  Enter a MATLAB variable name for the exported handle.

See “Automation Interface” on page 46-37 for information on using these object handles.

The following option is available under **Processor-in-the-Loop (PIL) Verification**:

- **Configure model to build PIL algorithm object code**

  Select this box to build PIL algorithm code.

Using Configuration Sets to Specify Your Target

Follow the steps in this example to see where to find and change Embedded Coder software settings. These steps are described to help you find the settings you need to get started using the demo models. To use the demos, you need to specify your target by working with configuration sets.

This example describes how to use Embedded Coder software to build a project from a demo model using two different toolchains. The instructions refer to C166 and TriCore® TASKING toolchains; adapt the instructions to your toolchain as appropriate.

Finding the Embedded Coder Software Settings.

1 Open the model `tasking_demo_enginewc`.

2 Double-click the `Active Configuration Set` block to open the Model Explorer (or select **View > Model Explorer**).

   Under `TASKING_demo_enginewc` is a list of configuration sets you can review. The currently selected set is labeled `(Active)`.

Reviewing and Changing the Configuration Settings. Inspect the active configuration set.

1 The default active configuration set for this model is `C166`. If you want to use a different target, right-click the configuration set that matches your target, and select **Activate**.

2 Click **IDE Link** to see the configuration settings.

3 The **Target Preferences Configuration** drop-down list shows all available target preference configurations. After you have set up target preferences for particular configurations, you can switch between them here (or in the Target Preferences dialog box).

   a Click **Edit Configuration** to inspect your current target preferences.

   b Before building, you must replace the string `<ENTER TASKING PATH>` to set up the correct paths to the target preferences `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`. See "Setting Target Preferences for Altium TASKING" on page 46-8.

c Click **OK** to dismiss the Target Preferences dialog box.

In the Embedded Coder demos, when you activate a configuration (e.g., `C166`), the appropriate **Target Preferences Configuration** is automatically selected. You may want to select a different target preference configuration description, e.g., if you have set up a custom configuration (such as `C167_user_hardware`). For an example, see "Creating a New Configuration" on page 46-80.

See "Adding the Embedded Coder Configuration Set Component" on page 46-13 for information on other Embedded Coder software settings in the Configuration Parameters.

4 Click **Code Generation** to see the selected system target file.

Note You can use a configuration set specifying any system target file with Embedded Coder software.

---

5 Click **Hardware Implementation** to see the C166 settings. If you are using a different target, make sure the settings match your device. Select from the **Device type** list. There are custom configurations and preconfigured settings that include the following processors:

- Infineon C16x, XC16x
- Infineon TriCore
- ARM 7/8/9
- Renesas M16C
- 8051 Compatible
- Freescale DSP563xx (16-bit mode)

Close the Model Explorer.

6 In the model `tasking_demo_enginewc`, right-click the `t_eng_speed` subsystem, and select **Code Generation > Build Subsystem**. Click **Build** in the dialog box to continue.

Watch the output messages in the MATLAB Command Window as code is generated, your TASKING toolchain EDE is launched, and a new project created.

Switching Target Preference Configurations. If you have multiple toolchains, you only have to set up your target preferences once. After this initial setup, it is simple to switch between different configurations. For example, to switch configurations from C166 to TriCore targets:

1. In the model `tasking_demo_enginewc`, double-click the `Active Configuration Set` block to open the Model Explorer.

2. Right-click `TriCore` and select **Activate**. Close the Model Explorer.

3. To rebuild the subsystem with the new settings, right-click the `t_eng_speed` subsystem, and select **Code Generation > Build Subsystem**.

   Watch the output in the MATLAB Command Window as code is generated, the TASKING C166 EDE is closed, the TASKING TriCore EDE is launched, and the new project created.

You can follow similar steps to specify your target in the other demo models.

To switch between simulator and hardware implementations for the same target configuration, you can use option sets. See “Option Sets” on page 46-24.

The next section describes using the build action setting in this example.

Setting Build Action

In this example, the model `tasking_demo_enginewc` is set up so the project is created but not built in the TASKING EDE.

To view this setting:

1. In the model `tasking_demo_enginewc`, select **Simulation > Configuration Parameters**.

2. Click **IDE Link** to see the **Build Configuration** parameters.

3. Look at the **Build Action** drop-down list.

Using this drop-down list, you can set what action to take after the Simulink Coder build process completes. You can create application and library projects in the TASKING EDE and then stop, or you can also choose to build, execute, or debug.

If you choose to build, execute, or debug, the CrossView Pro application will be launched.

Note The first time you build this model it will take a few minutes to compile the required Simulink Coder floating point library. This library is not rebuilt on subsequent builds unless necessary.

You can use the **Build Action** setting to do the following:

- `Create Application Project`

  Generates code for the model or subsystem, creates a TASKING application project for the selected TASKING configuration, connects to the TASKING EDE, and opens the application project (in addition to the required Simulink Coder and DSP System Toolbox Library projects, if required) in the TASKING EDE. This option does not build or execute the application.

  An `EDE_Obj` object handle is exported to the MATLAB workspace (if the option **Export EDE handle to MATLAB base workspace** is selected). This object allows you to interact with the TASKING EDE from MATLAB. For more information, see the section on using object handles, "Automation Interface" on page 46-37.

  Note To manually build the generated project in the TASKING EDE, right-click on the application project (starts with the same name as the model name), and select **Build**.

- `Create Library Project`

  Performs the same actions as `Create Application Project`, but this option archives the generated code into a library in the TASKING EDE. No `main.c` file is generated.

- `Create and Build Application Project`

  Performs the same actions as `Create Application Project`, but also instructs theTASKING EDE to build the application project.

  Note To manually debug the executable from the application project, click the Debug Application icon in the TASKING EDE.

- `Create and Build Library Project`

  Performs the same actions as `Create Library Project`, but also instructs the TASKING EDE to build the Library project.

- `Create, Build and Execute Application Project`

  Performs the same actions as `Create and Build Application Project` and also downloads the executable file to your CrossView Target and runs the executable. No debugging information is downloaded into the target with this option.

  A CrossView Pro object handle is exported to the MATLAB workspace (if the option **Export CrossView Pro handle to MATLAB base workspace** is selected). This object allows you to interact with the CrossView Pro debugger from MATLAB. For more information, see the section on using object handles, "Automation Interface" on page 46-37.

- `Create, Build and Debug Application Project`

  Performs the same actions as `Create, Build and Execute Application Project` but also downloads debugging information to the target. This option behaves the same way as the Debug Application icon in the TASKING EDE.

Accessing Utilities for TASKING

- "IDE Link Utilities for Use with TASKING dialog" on page 46-21
- "Tools Menu Items" on page 46-23

IDE Link Utilities for Use with TASKING dialog

Open the IDE Link Utilities for Use with TASKING dialog box by entering `taskingutils` in the Command Window or double-clicking Launch TASKING Utilities in the Simulink block library.

You see the following options:.

- **Target Preferences**

  Opens the Target Preferences Configuration Selection dialog box, and after you choose a configuration to match your target (e.g., `TriCore`), you can edit the Target Preferences dialog box. In this dialog box, you can modify your TASKING preferences configurations. You can also open this dialog box from the MATLAB prompt by typing `tasking_edit_prefs`.

  You must set up your target preferences before you can use Embedded Coder software. See "Setting Target Preferences for Altium TASKING" on page 46-8.

- **Select Preconfigured Target Preference Settings**

  Opens the Target Preferences Configuration Selection dialog box. Choose a configuration to match your target and click **OK**. Then you can select a preconfigured option set. Your target preferences are automatically updated according to the option set you select, for example, specifying either hardware or simulator settings. See "Option Sets" on page 46-24.

- **Launch and Test Communication with TASKING EDE**

  Opens the Target Preferences Configuration Selection dialog box. Choose a configuration and click **OK**, and Embedded Coder software tests whether MATLAB can communicate successfully with the Altium TASKING EDE for the selected configuration. You see messages at the command line to confirm whether communication is successful.

- **Create a New Model (configured for use with TASKING)**

  Creates a new untitled Simulink model, with Embedded Coder configuration set options already added. You can also configure an existing model by selecting the Simulink model menu item **Tools > Utilities for Use with TASKING(R) IDE > Add IDE Link Configuration to Model**.

- **View, Modify, and Copy Configuration Sets via Model Explorer**

  Opens the Model Explorer where you can edit all configuration sets available for each currently open model.

- **Create New Template Projects**

  The Embedded Coder product ships with preconfigured application and library template projects for the default configurations in the Target Preferences dialog box. You might, however, create your own template projects (using preconfigured options as a starting point), and use them with any configuration. See "Tutorial: Creating New Template Projects" on page 46-76 for an example.

  This option opens the Target Preferences Configuration Selection dialog box. Choose a configuration and click **OK**, and Embedded Coder software launches the appropriate TASKING EDE and creates new template projects for a specific **Tasking Configuration**. When you are prompted, choose a project folder, a template name, and an option set. See "Option Sets" on page 46-24 for more details. `app_template_name.pjt` and `lib_template_name.pjt` are created for the configuration you selected.

- **Open Existing Template Projects**

  Opens existing application and library template projects in the TASKING EDE for the selected **Tasking Configuration**. You can modify these options; however, it is preferable to do this by first creating new template projects, which avoids overwriting the default template projects. If you modify the default template projects, you can use the following function to recreate the defaults: `tasking_generate_templates`. You must specify your configuration description string, e.g.: `tasking_generate_templates('C166', true)`.

Note Opening or editing template projects causes the regeneration of application and library projects. When making any changes to template projects, it is important to make sure your changes are saved. To do this, remove the project from the project space; otherwise the changes may not be applied immediately. To remove a current project from the project space, right-click on it and choose **Remove from Project Space**.

- **Demos**

  Opens the Embedded Coder Demos page in the Help browser.

Tools Menu Items

In a Simulink model, you can access Embedded Coder menu items in the **Tools** menu. Select **Tools > Utilities for Use with TASKING(R) IDE** to see the following submenu items.

- **Target Preferences**

  As it does in the **Start** menu, this menu choice opens the Target Preferences Configuration Selection dialog box. After you choose a configuration, you can edit the Target Preferences Setup dialog box. You must set up your target preferences before you can use Embedded Coder software. See "Setting Target Preferences for Altium TASKING" on page 46-8.

- **Add IDE Link Configuration to Model**

  Adds Embedded Coder configuration options to the model configuration parameters.

  To see exactly which configuration parameter settings are changed, refer to `tasking_addto_configset.m`. Enter `edit tasking_addto_configset`.

- **Remove IDE Link Configuration from Model**

  Removes Embedded Coder configuration options from the model's configuration parameters.

- **Options**

  Opens the Configuration Parameters dialog box to show Embedded Coder software options. See "Configuration Set Options" on page 46-13.

Option Sets

- "What Are Option Sets?" on page 46-24
- "Supported DAS Software" on page 46-26

What Are Option Sets?

Option sets are preconfigured settings to specify the target configuration for the Altium TASKING tools. For example, after you set up your target preferences for a TriCore configuration, you can use option sets to switch between using an instruction set simulator configuration, two hardware board configurations, or a simulator with some MISRA C rule checking.

You can use option sets either:

- To switch between default target configurations, or
- When creating new template projects, to set up an initial configuration that you can choose to modify later

See "Tutorial: Using Option Sets" on page 46-75 for instructions.

The following preconfigured option sets are available.

A notation of "*" indicates the default in the Target Preferences. The processor type for the default configurations below is defined by your TASKING toolchain.

- Infineon TriCore:
  - * `tricore_sim`: Default instruction set simulator configuration.
  - `tricore_sim_misra`: As `tricore_sim`, but with some example MISRA C rule checking enabled. See also the TriCore MISRA C demo example, `tasking_demo_misra.m`, with instructions under Embedded Coder Demos.
  - `tricore_1796b`: Infineon TriCore 1796b hardware configuration.
  - `tricore_1766b`: Infineon TriCore 1766b hardware configuration.
- Infineon C166:

- c166_sim : Default instruction set simulator configuration.
- c167cr : Phytec kitCON-C167CR serial connection to hardware (_hw) and simulator (_sim) configurations.
- *c167cs : Phytec phyCORE-C167CS serial connection to hardware (_hw) and simulator (_sim*) configurations.
- st10f252 : STMicrolectronics MB449 ST10F25x EVA Board serial connection to hardware (_hw) and simulator (_sim) configurations.
- st10f269 : Phytec phyCORE-ST10F269 serial connection to hardware (_hw) and simulator (_sim) configurations.
- xc164cm : Infineon XC164CM U CAN start kit USB connection to hardware (_hw_u_can) and simulator (_sim_u_can) configurations. See "Supported DAS Software" on page 46-26.
- xc167ci: On-board parallel port wiggler connection to the Infineon XC167CI Starter Kit hardware (_hw) and simulator (_sim) configurations.

  xc167ci_hw_usb : USB wiggler connection to the XC167CI

  Note For xc167ci targets, you must change jumper 501 when switching between USB wiggler and on-board parallel port wiggler. See your board manual for details.

- Renesas M16C
  - * m16c_sim: Default instruction set simulator configuration.
  - r8ctiny_sim: Renesas R8C Tiny instruction set simulator configuration.
- ARM:
  - * arm_sim: Default instruction set simulator configuration.
  - arm_sim_big_endian: As arm_sim, but in big-endian mode.
- Freescale DSP563xx:
  - * dsp563xx_sim: DSP563xx Family, 16-bit memory model, instruction set simulator configuration.

  - dsp566xx_sim: DSP566xx Family instruction set simulator configuration.

- 8051:
  - * i8051_sim: Default, large memory model, no language extensions, floating point, instruction set simulator configuration.

Supported DAS Software

For the XC164CM and certain TriCore hardware like TC1766 and TC1796, you need to download and install the supported DAS software. If your installation of the TASKING toolset did not come with DAS, then you can download the latest DAS software from this URL:

http://www.infineon.com/das.

At the time of writing, the latest tested DAS versions are:

- DAS Edition v2.6.2
- JTAG JDRV LPT Server v2.4.0

Make sure you restart your computer as instructed after DAS installation.

Components

In this section...

"Project Generator" on page 46-27

"Automation Interface" on page 46-37

Project Generator

- "Overview of the Project Generator Component" on page 46-27
- "Project-Based Build Process" on page 46-29
- "Template Projects" on page 46-29
- "Shared Libraries" on page 46-31
- "Build Process — Folder Structure" on page 46-33

Overview of the Project Generator Component

The Embedded Coder Project Generator Component provides a customizable build process that is designed to work with the highly customizable code generation process provided by Simulink Coder. See "Project Generator" on page 46-3 for a summary.

To explain the separation of duties between Simulink Coder and Embedded Coder, the following sections discuss the terms *code generation process* and *build process*.

Code Generation Process. The code generation process is performed by the Simulink Coder family of products and is the process of translating a Simulink model into C code.

Customized code generation, perhaps to create target-specific device drivers or target-optimized code, is often a key requirement for users who want to generate code from Simulink models.

Simulink Coder and Embedded Coder provide a variety of mechanisms for users to customize the code generation process. For example, the standard code generation process, using the regular system target files

(like `grt.tlc` and `ert.tlc`) can be customized by making changes to the model's configuration parameters. Alternatively, for an even greater level of customization, including the ability to define custom Simulink Coder options, you can use a user created system target file.

The demos that come with Embedded Coder make use of the first type of customization with regular system target files. That is, the standard code generation process has been tailored for the appropriate target platform simply by changing the model's configuration parameters.

For greater flexibility, you should use a custom system target file. For further details on customizing the code generation process, see the Simulink Coder and Embedded Coder documentation.

Build Process. The build process is performed by Embedded Coder and is the process of taking the C code produced by the code generation process and building (assembling, compiling, and linking) it for the target platform.

A customized build process, perhaps to use optimized compiler and linker settings, or perhaps to produce a MISRA compliance report, is often a key requirement for users wishing to build code produced from Simulink models.

Embedded Coder provides access to the full build process customization capabilities of the TASKING tools by allowing the user to set up the exact required configuration in TASKING. Embedded Coder then uses this configuration as a template for the build process.

Memory Placement Example. As an example, to consolidate the descriptions above of code generation and the build process, consider the common task of placing program data into a particular area of memory on a target platform.

Usually, this is achieved by using compiler-specific notations (like #pragmas) to define special memory sections and to assign data definitions to those sections. Additionally, a linker command file defines the different available memory regions on the target, and where in these regions the different memory sections should be located.

Splitting this task between the processes of code generation and building could be done as follows:

1. Customized code generation defines memory sections and assigns data.
2. Customized build process defines memory regions and assigns memory sections.

Project-Based Build Process

The Embedded Coder build process automatically creates TASKING EDE projects representing the application and libraries to be built.

A Simulink Coder application usually consists of some application code that makes references to modules that are part of libraries like the Simulink Coder library.

Embedded Coder creates separate projects for the main application code and each required library. The required libraries are included in the main application projects as subprojects.

Although the build process is project-based, underlying the projects are "makefiles" that can be used independently of the EDE. For an example of how to obtain the appropriate make command, see the demo instructions in `tasking_demo_objects` .

Target Project Space. Embedded Coder places projects in a project space known as the *target project space*. The location of the target project space is controlled by the `Target_Project_Space` setting in the Target Preferences, and usually depends on the `$(DEFAULT_LOCATION)` token, which is expanded based on the current folder at the time the build process is invoked, the model name, and model configuration settings, including the name of the template application project.

Template Projects

Template projects are regular TASKING EDE projects that are used by Embedded Coder to allow customization of the build process. Template projects are tied to particular TASKING Configurations as set up in the Target Preferences.

There are two types of template projects: application, and library template projects.

The application template project is used as the template for application projects and the library template project is used as the template for library projects.

Relocation of Template Projects. During the build process, the template project is copied to a target project location, and is then populated with the information relating to how to build the generated code.

Therefore, the project options of the template project become the project options of the target project, and hence the build process is customized according to the template project.

On subsequent build processes, Embedded Coder determines whether the template project has been updated since it was last copied to the target project location. If it has, then the target project is updated with a new copy of the template project. Otherwise, the target project is not updated from the template project.

Note Project options should be updated in the template project and not in the target project.

How the Build Process Modifies the Relocated Template Project. The Embedded Coder build process determines if any changes (preprocessor defines, include paths and source files) to the target project are required to build the code associated with a particular model, and updates the target project only if required. Thus, unnecessary project rebuilding is avoided.

Any include paths and preprocessor defines in the template project are always maintained in the target project. Maintaining this information is useful for keeping the include path to the compiler's standard header files, and setting global defines.

Additionally, the optional startup code file automatically generated by the EDE is also maintained.

Note Adding any other source files to your template project is not supported and will result in errors. Instead, you should add source files to the project by adding them to the Build Info object by using either the Code Generation > Custom Code settings in the configuration parameters, the `rtwmakecfg.m` mechanism, or by writing your own post code generation command (taking care not to overwrite any existing commands). See the Simulink Coder documentation for details.

Shared Libraries

Embedded Coder models that share the same target project space share required libraries such as the Simulink Coder library. Sharing of libraries means that a library is only built the first time a model that requires it is built.

The advantages of this shared library approach are

- No unnecessary per-model building of libraries; models with similar library requirements (e.g., integer code only) can share libraries.
- Libraries are built with the project options specified in the corresponding template project.
- Multiple sets of libraries, each with custom model, project options, or both can coexist.

Utility code generation: Shared Location. The shared library approach uses the Simulink Coder “Utility code generation” feature.

By setting Utility code generation to use a shared location, rather than the model-specific default, you can ensure that the library projects created have no dependence on model-specific generated code. This feature is the key to allowing library projects to be shared between models.

As an example, consider the generated header file, `rtwtypes.h`, that contains the set of Simulink Coder data types available for compiling code modules, including any libraries.

With the Utility code generation set to the default, individual `rtwtypes.h` files are generated into each code generation folder. Therefore, multiple

definitions of `rtwtypes.h` would exist for a library shared between these models. The problem is, how can one of these `rtwtypes.h` files be chosen to build the library?

Setting the Utility code generation to use a shared location provides a solution. In this case, a single `rtwtypes.h` file is generated into a folder shared between a set of models. This single file can be used to build the library without any dependence on the model-specific generated code.

Supporting Multiple Shared Utility Function Locations: Build Folder Suffix. The approach outlined in the previous section works well for a single set of models that have the same shared utility requirements.

However, what happens if you have two sets of models, each set with different shared utility requirements?

Normally, the Simulink Coder code generation process uses the current working folder as the location for generated files. In this location, it supports only a single shared utilities folder for each system target file. Therefore, it is possible for conflicts over the contents of the shared utility folder to occur.

Example 1

For example, conflicts would occur if the Hardware Implementation settings were different for two models using the same system target file. If the standard `grt.tlc` or `ert.tlc` code generation process is customized by changing configuration set parameters, this situation is highly likely to occur.

To work around this problem, when using a `Target_Project_Space` (specified in the Target Preferences) containing the `$(DEFAULT_LOCATION)` token, Embedded Coder automatically appends the name of the current template application project to the regular Simulink Coder build folder suffix. This creates code generation and project folders that are specific to the current template application project, and so also specific to the current Hardware Implementation settings. Different Hardware Implementation settings always have different template projects.

Example 2

Another common example of this conflict, for two models sharing the same system target file, would be if one model was configured to support floating-point numbers and the other was configured to support integer code only.

To work around this conflict, use the Embedded Coder options **Add build subdirectory suffix** and **Build subdirectory suffix**.

If you select the **Add build subdirectory suffix** check box, then the **Build subdirectory suffix** you enter is appended to the regular Simulink Coder build folder suffix (before the name of the template application project discussed earlier, see "Tutorial: Creating New Template Projects" on page 46-76). This creates code generation and project folders that are specific to both the **Build subdirectory suffix** setting and the template projects.

For example, you can add `fp` for floating point models and `int` for non-floating-point models.

Note Using the same build subfolder suffix for a similar set of models allows them to generate code into their own working folder, avoiding conflict with other models, while still allowing a shared utilities folder.

This feature of Embedded Coder removes the need for the user to manually manage changing folders to avoid shared utility folder conflicts.

For examples of using this setting, look at the models included in the product help demos for Altium TASKING.

Build Process — Folder Structure

The following table shows the typical folders that are created, relative to the current working folder, during the Simulink Coder code generation process and the Embedded Coder build process.

Folder	Contents

`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_$(MODEL)`
e.g,
`ert_rtw_int_tricore_sim\pjt_fuelsys0`
	Main project: `$(MODEL).pjt` and associated files.

`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_rtwlib`	Simulink Coder library project: `rtwlib.pjt` and associated files.
`$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)\pjt_rtwshared` (if required)	Shared utilities library project: `rtwshared.pjt` and associated files.

`$(MODEL)_$(REG_SUFFIX)_$(MODEL_SUFFIX)_$(TEMPLATE_NAME)`
e.g.,
`fuelsys0_ert_rtw_int_tricore_sim`
	Simulink Coder code generation folder.

Key	
`$(MODEL)`	Simulink Coder code generation model name (e.g., `fuelsys0`).
`$(TEMPLATE_NAME)`	Token expanded from the name of the template application project in the target preferences (e.g., `tricore_sim`). If the project name is prefixed with "`app_`" this token is removed.
`$(REG_SUFFIX)`	Regular Simulink Coder build folder suffix (e.g., `ert_rtw`).
`$(MODEL_SUFFIX)`	Model-specific build folder suffix (e.g., `int`).

See the next section, "Command Line Project Information" on page 46-35, for details about finding file names, paths, and other build information.

Command Line Project Information. When you build an application you can see information containing links at the MATLAB command line. You can use these links to get further details such as paths to projects, preprocessor defines, include paths, added files and their locations.

The following example shows a typical output:

```
### Building the PIL Application...
### Updating EDE projects according to BuildInfo object.
Please wait...
Creating project: t_shift_alg_ert_rtw_pil.pjt
Updating preprocessor defines in project:
t_shift_alg_ert_rtw_pil.pjt
Updating include paths in project:
t_shift_alg_ert_rtw_pil.pjt
Adding source files to project:
t_shift_alg_ert_rtw_pil.pjt
```

You can click the hyperlinks within these messages to get more information. The build messages are more readable with this information hidden, and the links provide access when you require more details.

Click the project file name (e.g., `t_shift_alg_ert_rtw_pil.pjt`) to see the full path to the project being built, like the following example:

```
Project path: D:\MATLAB\\work\tricore_fp\tricore_sim\
pjt_t_shift_alg_ert_rtw_pil\t_shift_alg_ert_rtw_pil.pjt
```

Click `preprocessor defines` to see a list of preprocessor defines similar to the one in the following example:

```
t_shift_alg_ert_rtw_pil.pjt preprocessor defines:

INTEGER_CODE=0
MAT_FILE=0
MODEL=t_shift_alg
MT=0
MULTI_INSTANCE_CODE=0
NCSTATES=0
NUMST=1
ONESTEPFCN=1
```

```
TERMFCN=1
TID01EQ=0
```

Click `include paths` to see a list of include paths similar to the one in the following example:

```
t_shift_alg_ert_rtw_pil.pjt include paths:

$(PRODDIR)\include
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw
D:\MATLAB\work\tricore_fp
D:\MATLAB\matlab\toolbox\rtw\targets\tasking\taskingdemos
D:\MATLAB\matlab\extern\include
D:\MATLAB\matlab\simulink\include
D:\MATLAB\matlab\rtw\c\src
D:\MATLAB\matlab\rtw\c\libsrc
D:\MATLAB\matlab\rtw\c\ert
D:\MATLAB\work\tricore_fp\slprj\ert\_sharedutils
D:\MATLAB\matlab\toolbox\rtw\targets\tasking\tasking\pil
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil
```

Click `source files` to see a list of files added and their full paths.

```
t_shift_alg_ert_rtw_pil.pjt added files:

D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_common.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_lib.c
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\pil\
pil_interface_lib.h
D:\MATLAB\toolbox\rtw\targets\tasking\tasking\
tasking_pil_main.c
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil\
pil_interface.c
D:\MATLAB\work\tricore_fp\t_shift_alg_ert_rtw_pil\
pil_interface_data.h
D:\MATLAB\work\tricore_fp\tricore_sim\
```

```
pjt_exp_t_shift_alg_ert_rtw\exp_t_shift_alg_ert_rtw.pjt
D:\MATLAB\work\tricore_fp\tricore_sim\pjt_rtwlib\rtwlib.pjt
```

Automation Interface

- “Overview of Automation Interface Component” on page 46-37
- “Classes” on page 46-38
- “Using Objects” on page 46-38
- “List of Methods” on page 46-46
- “Details of Particular Methods” on page 46-49

Overview of Automation Interface Component

The Embedded Coder Automation Interface Component provides powerful MATLAB API’s for automating interaction with the TASKING EDE and CrossView Pro Debugger. See “Automation Interface” on page 46-4 for a summary.

Objects for Embedded Coder. Embedded Coder uses object-oriented programming techniques and requires a basic knowledge of some object-oriented terminology. The following are some fundamental terms you should understand:

- Object — Something you can operate on. An object is an instance of a class, created by calling the class constructor.
- Class — A class defines the properties and methods common to all objects of the class.
- Constructor — A function that creates an object, based on the class definition, and initializes it.
- Method — An operation on an object, defined as part of the class definition.
- Property — Part of an object, treated as a variable at times, that is defined as part of the class definition.
- Handle — A mechanism to access any object that Embedded Coder creates. Used in this guide to refer to the object. Often the handle is the name you assign when you create the object.

The following sections describe how to use and get help for Embedded Coder objects. See "Objects Demo Example" on page 46-46 for an example demonstrating some basic capabilities of Embedded Coder objects.

Classes

The following table shows the different classes that are provided with Embedded Coder for use with Altium TASKING.

Class	Description
`tasking.edeapi`	Represents the TASKING EDE.
`tasking.edeprojectspace`	Represents a project space in the TASKING EDE.
`tasking.edeproject`	Represents a project in the TASKING EDE.
`tasking.xviewapi`	Represents the TASKING CrossView Pro debugger.
`tasking.Tasking_Configuration`	Property of a `tasking.edeapi` class representing TASKING configuration details.
`tasking.EDE_Configuration`	Property of a `tasking.tasking_Configuration` representing EDE configuration details.
`tasking.CrossView_Pro_Configuration`	Property of a `tasking.tasking_Configuration` representing CrossView Pro configuration details.

Using Objects

The topics in this section are:

1. "Using the altiumtasking Function" on page 46-39
2. "Creating an Object Directly" on page 46-43
3. "Determining the Available Methods for a Class" on page 46-44

4 "Obtaining Help for a Class Method" on page 46-44

5 "Calling a Method" on page 46-45

6 "Determining the Available Properties for a Class" on page 46-45

7 "Accessing a Property" on page 46-45

8 "Objects Demo Example" on page 46-46

Using the altiumtasking Function. You can use `altiumtasking` to create Embedded Coder objects for Altium TASKING. The command

```
[ede,xview,projspc,proj] = altiumtasking
```

returns the following results:

- *ede* — A handle to the TASKING EDE application.
- *xview* — A handle to the TASKING CrossView Pro application.
- *projspc* — A handle to the project space currently open in the EDE.
- *proj* — A handle to the active project in the EDE.

Note You can only run `altiumtasking` after you configure your target preferences.

When you first run `altiumtasking`, the Target Preferences Configuration Selection dialog box opens. Select your required configuration (for example, `TriCore`), and click **OK**.

- If you supplied a valid project space file when you configured the target preferences, `altiumtasking` opens that project space in the EDE window and returns a handle to it. However, if no valid project space file exists, then `altiumtasking` returns an empty project space handle.
- If one or more projects are defined in the project space, and opened, `altiumtasking` returns a handle to the active project. If no project is opened, `altiumtasking` returns an empty project handle.

You can also run `altiumtasking` in the following ways:

`[`*`ede`*`,`*`xview`*`,`*`projspc`*`] = altiumtasking` does not create and return the project handle.

`[`*`ede`*`,`*`xview`*`] = altiumtasking` does not create and return the project space and project handles.

`ede` `= altiumtasking` does not create and return the CrossView Pro application, project space and project handles.

`altiumtasking(`*`Property`*`, `*`Value`*`)` enables you to specify parameters that control the behavior of the function. For example:

```
altiumtasking('configDesc','TriCore')
```

specifies the use of the `TriCore` configuration in the function.

Create EDE and CrossView Pro Handles

This example generates EDE and CrossView Pro handles.

```
>>[ede, xview] = altiumtasking;

Registering COM object: "D:\share\apps\BuildTools\win32\TASKING\apps\tricore\v2.5r2\ctc\bin\xfwtc.exe" -
Creating COM object: xfwtc.CommandLine
Initializing COM object: -G "C:\Temp" -ini "C:\Temp\tricore_default.ini" -tcfg
"D:\share\apps\BuildTools\win32\TASKING\apps\tricore\v2.5r2\ctc\etc\tsim.cfg" -C tc1775b

Testing COM communications with CrossView Pro by sending command: "echo MATLABLinkTest"
[Test timeout is 60 seconds, to allow hardware setup - use "Ctrl-C" to terminate]
Test successful.
```

Note You must save preferences to the CrossView Pro `.ini` file. The link software overwrites the temporary file, (`C:\Temp\tricore_default.ini`, when it creates a new `xviewapi` object.):

1 In CrossView Pro, select **File > Exit**. The Options dialog box opens.

2 On the **Save** tab, select the **Save desktop and target settings** check box.

3 Click **Exit**. CrossView Pro exits after a few moments, saving your preferences in the `.ini` file.

You can view the handles CrossView Pro created.

```
>> ede
tasking.edeapi (handle)
  configuration: [1x1 tasking.Tasking_Configuration]

 "configuration" property:
  Configuration_Description: 'TriCore'
         EDE_Configuration: [1x1 tasking.EDE_Configuration]
  CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_Configuration]

  Connected to EDE: 0

>> xview
tasking.xviewapi (handle)
 configuration: [1x1 tasking.CrossView_Pro_Configuration]

  "configuration" property:
    CrossView_Pro_Executable: [1x76 char]
          Initialization: [1x29 char]
       Initialization_File: [1x107 char]

  Status:
   Current executable: None
     Current project: None
   isRunning: 0
```

```
    eventReporting: 1
```

Create an EDE handle for TriCore

This example creates an EDE handle for `TriCore`.

```
>> obj = altiumtasking('configDesc', 'TriCore')

tasking.edeapi (handle)
 configuration: [1x1 tasking.Tasking_Configuration]

 "configuration" property:
     Configuration_Description: 'TriCore'
             EDE_Configuration: [1x1 tasking.EDE_Configuration]
   CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_Configuration]

   Connected to EDE: 0
```

Creating an Object Directly. Embedded Coder allows you to create objects directly. To find out how to create an object of a particular class you can use the `help` function to find help for the constructor. At the MATLAB command prompt, enter

```
help <classname>.<constructorname>
```

For example, for the `tasking.edeapi` class, enter

```
help tasking.edeapi.edeapi
```

For the `tasking.edeprojectspace` class, enter

```
help tasking.edeprojectspace.edeprojectspace
```

Follow these steps to create example objects.

1. To create a `tasking.edeapi` object, you call the constructor as follows:

   ```
   Ede = tasking.edeapi
   ```

   The name on the left side of the “=” could be any valid MATLAB identifier and is the handle to the object.

   You must choose a configuration, then communication is tested with the TASKING EDE. At the command line you see the configuration target preferences.

2. To create a `tasking.edeprojectspace` object, you call the constructor as follows:

   ```
   tasking.edeprojectspace(projspace, edeapi)
   ```

   where `projspace` is the absolute path of the TASKING Project Space this object will relate to, and `edeapi` is a `tasking.edeapi` object.

   ```
    ps = tasking.edeprojectspace('D:\MATLAB\work\
   myprojspace.psp', Ede)
   ```

3. To create a `tasking.edeproject` object, you call the constructor as follows:

   ```
   tasking.edeproject(proj, edeprojspace)
   ```

where `proj` is the absolute path of the TASKING Project this object relates to, and `edeapiprojspace` is a `tasking.edeprojectspace` object, as shown in the following example:

```
proj = tasking.edeproject('D:\MATLAB\work\myproj.pjt', ps)
```

4 To create a `tasking.xviewapi` object, you call the constructor as follows

```
xv = tasking.xviewapi
```

You must choose a configuration, then communication is tested with CrossView Pro. At the command line, you see the configuration target preferences.

Determining the Available Methods for a Class. After you create an object, you can find the available methods by running the "methods" function.

1 For example, to find the methods available on the `tasking.edeapi` object created above (in "Creating an Object Directly" on page 46-43), enter `methods(Ede)`.

2 To find the methods available on the `tasking.edeprojectspace` object previously created, enter `methods(ps)`.

3 To find the methods available on the `tasking.edeproject` object previously created, enter `methods(proj)`.

4 To find the methods available on the `tasking.xviewapi` object previously created, enter `methods(xv)`.

To see the methods available, refer to the tables in "List of Methods" on page 46-46.

Obtaining Help for a Class Method. To get help for a class method, you can use the `help` function.

For example, to find out more about the `getProject` method of the `tasking.edeapi` class, you could enter the following command:

```
help tasking.edeapi.getProject
```

MATLAB returns the following output:

```
GETPROJECT - get the active Project in the EDE
project = getProject
project: edeproject object representing the active Project
 in the EDE
project will be empty if there is no open project
```

To see the methods available, refer to the tables in "List of Methods" on page 46-46.

Calling a Method. When you know the details of a class method, you can call it using dot (.) notation.

For example, to get a `tasking.edeproject` object representing the active project, run the following command:

```
project = Ede.getProject
```

Determining the Available Properties for a Class. After you create an object, you can find the available properties by running the `get` function.

For example, to find the properties available on the `tasking.edeapi` object created above, enter

```
get(Ede)
```

Accessing a Property. You can access a property of a class using dot (.) notation.

For example, to get the "configuration" property of the `tasking.edeapi` object created above, enter:

```
config = Ede.configuration
tasking.Tasking_Configuration (handle)
      Configuration_Description: 'C166'
              EDE_Configuration: [1x1 tasking.EDE_Configuration]
    CrossView_Pro_Configuration: [1x1 tasking.CrossView_Pro_
Configuration]
```

Objects Demo Example. For experience using objects, you can work through the demo example, `tasking_demo_objects`.

This example provides step-by-step instructions for using Embedded Coder objects to communicate with the TASKING EDE and CrossView Pro debugger from the MATLAB command line. You can use any command available in the powerful CrossView Pro command language. The demo illustrates using objects during the process of building and debugging projects.

List of Methods

See the following tables for lists of available methods:

- "Methods for Class tasking.edeapi" on page 46-46
- "Methods for Class tasking.edeprojectspace" on page 46-48
- "Methods for Class tasking.edeproject" on page 46-48
- "Methods for Class tasking.xviewapi" on page 46-48

The public methods are shown in the tables (methods beginning with "`p`" or "`p_`" are private methods and should not be used).

Methods for Class tasking.edeapi.

close	getOptionSetNames
disp	getProject
display	getProjectSpace
edeapi	getTargetProject
exec	getToolchainInfo
execApiMacro	newProject
execRetNumeric	newProjectSpace
execRetString	newProjectTemplates
getCreatedEDEProcess	newProjectTemplatesViaUI
getOptionSet	newTempProjectSpaceIfNoneOpen
openProjectTemplates	processTemplateProject

pwd	validateToolchainDirectory
hilite_system	connect
isconnected	

Methods for Class tasking.edeprojectspace.

add	deleteParentDir
getEDE	isopen
checkValid	disp
getOriginalPath	new
checkValidProject	display
getPath	open
close	edeprojectspace
isequal	remove

Methods for Class tasking.edeproject.

add	getEDE	isopen
build	getFiles	new
checkValid	getHyperlink	open
close	getIncludes	rebuild
debug	getMakeCmd	remove
disp	getOriginalPath	run
display	getPath	setCDefines
edeproject	getProjectSpace	setIncludes
getBuildOutput	getTarget	setPerformToolchainName-Check
getCDefines	hasFile	
getDir	isequal	

Methods for Class tasking.xviewapi.

addBreakpointCallback	getEventReporting
getFunctionConfiguration	debug
disp	halt

removeBreakpointCallbacks	display
isRunning	setEventReporting
downloadAndRun	execute
xviewapi	executeAndWait
getCommandResponse	getExecutable
getProject	hilite_system
readMemoryUnits	writeMemoryUnits
reset	stackProfile
stackProfileReset	

Details of Particular Methods

The following methods of the `tasking.xviewapi` object simplify reading from and writing to target memory units (the smallest addressable unit in the memory of the target).

- `readMemoryUnits`

  To see help for this function, enter

  ```
  help tasking.xviewapi.readMemoryUnits
  ```

  at the MATLAB command line.

- `writeMemoryUnits`

  To see help for this function, enter

  ```
  help tasking.xviewapi.writeMemoryUnits
  ```

  at the MATLAB command line.

Use these functions with the MATLAB functions, `typecast` and `swapbytes`, for reading and writing data of different datatypes.

To see examples of syntax, see the demo example, `tasking_demo_objects`.

Verification

In this section...

"Processor-in-the-Loop (PIL) Simulation" on page 46-50

"C Code Coverage Reports" on page 46-58

"Execution Profiling" on page 46-60

"Stack Profiling" on page 46-63

"Bidirectional Traceability Between Code and Model" on page 46-66

"MISRA C Rule Checking" on page 46-67

Processor-in-the-Loop (PIL) Simulation

- "Processor-in-the-Loop Overview" on page 46-50
- "PIL Workflow" on page 46-51
- "Creating a PIL Block" on page 46-53
- "Building, Running, and Debugging PIL Block Applications" on page 46-54
- "PIL Metrics" on page 46-57

Processor-in-the-Loop Overview

Overview of PIL Simulation

Processor-in-the-loop (PIL) simulation is a verification technique designed to help you evaluate how well a candidate algorithm (e.g., a control system) operates on the actual target processor selected for the application.

When using Embedded Coder software, you have the following options for PIL simulation:

- Top-model PIL simulation mode — you can run a complete model as a PlL simulation on your target processor or instruction set simulator.
- Model block PIL simulation mode — use PIL simulation for a model reference component.

- PIL block — you can create a PIL block from one of several Simulink components including a model, a subsystem in a model, or subsystem in a library.

For information on processor-in-the-loop and how to use these different options, seeChapter 39, "Verifying Generated Code With SIL and PIL Simulations" in the Embedded Coder documentation:

Note All options (i.e. Top-model PIL, Model block PIL, and PIL block) are available when you use Embedded Coder software with Altium TASKING tools.

To learn how to use the top-model and Model block PIL simulation modes, refer to the Embedded Coder documentation linked above.

The following sections describe demos and detailed information on using the **PIL block** with Embedded Coder software and Altium TASKING tools.

PIL Workflow

You can work through the PIL block verification workflow demo for a hands-on example illustrating using SIL and PIL for system and unit testing: `tasking_demo_system_simulation.mdl`.

By running this demo you will learn how to:

- Use Software-in-the-Loop (SIL) to verify correct behavior of source code, generated by Embedded Coder software and executing on the host processor
- Use Processor-in-the-Loop (PIL) to verify correct behavior of object code and generate metrics; the object code is cross-compiled from source code generated by Embedded Coder software and executes on a target.
- Create system and unit test models
- Work with multiple heterogeneous target processors
- Include existing / legacy algorithms for SIL and PIL verification
- Export a generated algorithm for inclusion in an existing project

- Generate a fully deployable model-based application

Using `target_block_verify`. The function `target_block_verify` is used in the PIL Verification Workflow demo, `tasking_demo_system_simulation.mdl`.

You can use `target_block_verify` to verify a generated PIL or SIL block and compare the results with the simulation or algorithm block.

```
[LOG_SIGS1, LOG_SIGS2] = target_block_verify('BLOCK1', 'BLOCK2')
```

turns on signal logging for the outports of `BLOCK1`, the model containing `BLOCK1` is simulated and the logged signals are returned in `LOG_SIGS1`.

Next, `BLOCK1` and `BLOCK2` are swapped, the same model is simulated again, and the logged signals for `BLOCK2` are returned in `LOG_SIGS2`.

To verify a SIL or PIL block, set `BLOCK1` to the simulation or algorithm block, and set `BLOCK2` to the generated PIL or SIL block for `BLOCK1`.

Use full path names of Simulink blocks for `BLOCK1` and `BLOCK2`.

`BLOCK2` may be in the same model as `BLOCK1`, or in its own model. The model(s) containing `BLOCK1` and `BLOCK2` are loaded.

`LOG_SIGS1` and `LOG_SIGS2` are ModelDataLogs objects containing all the logged signals for the outports of `BLOCK1` and `BLOCK2` respectively. The data returned for each outport is a Timeseries object that allows comparison and plotting capabilities.

If `BLOCK1` and `BLOCK2` are in the same model, then one `LOG_SIGS` output is returned containing the data for both `BLOCK1` and `BLOCK2`.

Caution `target_block_verify` makes temporary changes to the model by swapping `BLOCK1` and `BLOCK2` in addition to setting some logging options. Although `target_block_verify` restores the original settings of the model, it is recommended that you save a copy of your model first.

Creating a PIL Block

The PIL settings can be found in the Configuration Parameters dialog box under the **IDE Link** settings.

The following options are available under **Processor-in-the-Loop (PIL) Verification**

- **Configure model to build PIL algorithm object code**

  Select this box to create PIL algorithm object code as part of the Simulink Coder code generation process.

After you create and build a PIL block, you can either:

- Copy it into your model to replace the original subsystem (save the original subsystem in a different model so it can be restored), or
- Add it to your model to compare with the original subsystem during simulation.

See "Building, Running, and Debugging PIL Block Applications" on page 46-54 for more details.

Building, Running, and Debugging PIL Block Applications

This section includes the following topics:

- "Building and Downloading PIL Applications" on page 46-54
- "PIL Debugging" on page 46-56

Building and Downloading PIL Applications. After you create a PIL block, you must build and download it before you can use it for simulation.

To build and download the PIL application manually:

1. Double-click the PIL block to open the mask.
2. Click **Build**. Wait until the **Application** name in the mask is updated and you see the "build complete" message.
3. Click **Download**.
4. Wait until the output in the MATLAB command window stops and you see the "download complete" message in the PIL block, and then click **OK** to close the block mask.

The PIL Application is now ready. To cosimulate with it, you must copy the PIL block into your model, either to replace the original subsystem or in addition to it for comparison. Click **Start Simulation** to run a PIL simulation.

After the test, Embedded Coder software returns execution profiling, code coverage, and stack profiling reports to MATLAB for your review. See "PIL Metrics" on page 46-57 for more information.

Note When copying PIL blocks to be used in the same model or in different models that simulate simultaneously, you must click the **Download** button in the PIL block mask in the new block after copying.

Clicking **Download** creates new connections (handles) to the TASKING EDE and CrossView Pro debugger. Otherwise, the same debugger handle may be used by multiple PIL blocks simultaneously and simulation errors or incorrect

results may occur. This concern does not apply when copying PIL blocks created automatically as part of the build process because the untitled model and test harness are typically not simulated together.

See the Embedded Coder demos for examples with instructions to enable you to build and download PIL blocks and use them in simulation.

Note The Download button has been removed from the PIL block mask.

PIL Block Parameters

For generic PIL block information, see Chapter 39, "Verifying Generated Code With SIL and PIL Simulations" in the Embedded Coder documentation.

Embedded Coder software creates PIL blocks with both the **Simulink system path** and **Configuration** properties automatically configured.

The available **Configurations** correspond to the **TASKING Configuration** descriptions in the Target Preferences.

Some guidelines for choosing a valid configuration:

1. The configuration must generate debugging information because Embedded Coder software requires this information to communicate with the PIL application.

2. The configuration must be compatible with the **Target Preferences Configuration** that was used to build the PIL algorithm. The fact that these two configurations need not match exactly allows the flexibility for the PIL algorithm to be compiled as if for a production environment, for example, without generating debugging information. However, you must be careful to ensure that the configurations are compatible in terms of linking, otherwise build errors occur when building the PIL application. In many cases, it is appropriate to use exactly the same configuration for building both the PIL algorithm and PIL application and therefore no errors can ever occur because of incompatibilities between configurations.

PIL Debugging. Prior to PIL simulation you can use the CrossView Pro debugger to set breakpoints, so that you can step through the code and watch variables during simulation. To do this, you must set breakpoints in CrossView Pro prior to starting the simulation as follows:

1. When the build process completes, a minimized CrossView Pro window should appear on your Windows Start menu. Maximize the CrossView Pro window.

2. In CrossView Pro, select **File > Open Source**, and choose a source file to open. A typical choice would be to open the main generated file associated with the algorithm, e.g. *`model`*`.c`.

3. Choose a location in the file to set a breakpoint and click the "breakpoint" button to the left of the line. A typical location for setting a breakpoint in the *`model`*`.c` file would be one of the `step` functions.

Note You can set multiple breakpoints in multiple files if you wish.

---

4. To add a variable to the watch, double-click the variable, and then click **Add Watch** in the Expression Evaluation window. A typical variable to add to the watch would be either the external inputs or external outputs structures, which usually represent all of the inputs and outputs of the algorithm.

5. Start the PIL simulation in Simulink. When the breakpoint is hit, Simulink pauses. CrossView Pro is available for debugging, and watch variables are updated. You can step through the code, set more breakpoints, and analyze data.

6. When you are finished debugging, you can continue running by clicking the "play" button in CrossView Pro. This will allow the PIL simulation to continue. If you left the breakpoint in place then the simulation stops at that point again. To return to uninterrupted simulation, remove the breakpoints.

Caution Never remove the PIL synchronization breakpoint (set on the `pilDataBreakpoint` function). This breakpoint is used to maintain synchronization between Simulink and CrossView Pro.

As an alternative to manual configuration in CrossView Pro, you can obtain a handle to the `tasking.xviewapi` object associated with a PIL block by using the `tasking_pil_crossview_handle` command as follows:

```
crossview = tasking_pil_crossview_handle('block')
```

where `block` is the full Simulink system path to the PIL block. You can use `gcb` to obtain the system path after clicking on the PIL block.

This handle can be used prior to PIL simulation to configure breakpoints, etc., by using the CrossView Pro command language.

Caution This handle should not be used during PIL simulation as this could lead to incorrect PIL results or termination of the PIL simulation.

10-Second Pause on Termination of the CrossView Pro Debugger

When terminating an instance of the CrossView Pro debugger application that was launched by Embedded Coder software, there is a pause of about 10 seconds before the CrossView Pro window closes. This 10-second pause is the intended behavior of CrossView Pro when acting as a COM server; CrossView Pro pauses for the 10 seconds to wait for clients such as MATLAB to release their COM references.

PIL Metrics

The following metrics provide verification information to be used in conjunction with the main “signal level” simulation results:

- C Code Coverage reports
- Execution profiling
- Stack profiling

C Code Coverage Reports

After you download a PIL application and run a simulation, you can view reports in MATLAB. The reports available depend on the target configuration. For the C166 Simulator, a hyperlink is provided for each report in the MATLAB command window towards the end of the Simulink Coder build log (as shown in the following example):

```
PIL reports available from CrossView Pro for block: fuelsys
Coverage ("covinfo"):  Yes (pil_coverage_report)
Profiling ("proinfo"): Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"):  Yes
(pil_cumulative_profiling_report)
```

To view the C code coverage report, click the hyperlink `pil_coverage_report`:

```
pil_coverage_report =

Module:   temp                                   0%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim_pil...
\pil_interface.c    74%
Function: pilInitialize                         75%
Function: pilGetUDataSymbol                     75%
Function: pilStep                               71%
Function: pilGetYDataSymbol                     75%
Function: pilTerminate                          75%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_ide_data_stream.c    93%
Function: pilDataBreakpoint                    100%
Function: pilReadData                           90%
Function: pilWriteData                          94%
Function: pilDataInit                          100%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_interface_lib.c    97%
Function: getNextSymbol                        100%
Function: processData                           93%
```

```
Function: pilCommandLoop                           99%
Module:   ..\..\..\..\..\matlab\toolbox\rtw\targets\common...
\tgtcommon\pilsrc\pil_main.c    83%
Function: main                                     83%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\fuelsys1.c
    61%
Function: Sens_Failure_Counter                     13%
Function: Fueling_Mode                             24%
Function: Init_controllogic                       100%
Function: controllogic                             47%
Function: fuelsys1_step                            79%
Function: fuelsys1_initialize                     100%
Function: fuelsys1_terminate                      100%
Module:   MEMCPY_C                                100%
Module:   MEMSET_C                                100%
Module:   MUL                                     100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\binarysearch_s16.c    89%
Function: BINARYSEARCH_S16                         89%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\dotproduct_s32s16.c   100%
Function: DotProduct_s32s16                       100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\interpolate_even_s16_s16_sat.c    84%
Function: INTERPOLATE_EVEN_S16_S16_SAT             84%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\interpolate_s16_s16_sat.c    83%
Function: INTERPOLATE_S16_S16_SAT                  83%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\look2d_s16_s16_s16_sat.c   100%
Function: Look2D_S16_S16_S16_SAT                  100%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils...
\div_s32_sat_floor.c    77%
Function: div_s32_sat_floor                        77%
Module:   UDIL                                     29%
Module:   UMOL                                     24%
Module:   fuelsys1_pil                              0%
Module:   CSTART                                    0%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\fuelsys1_data.c
     0%
```

Execution Profiling

- "CrossView Pro Execution Profiling" on page 46-60
- "Task Execution Profiling Kit" on page 46-63

CrossView Pro Execution Profiling

After you download a PIL application and run a simulation, you can view reports in MATLAB. The reports available depend on the target configuration. For the C166 Simulator, a hyperlink is provided for each report in the MATLAB command window towards the end of the Simulink Coder build log (as shown in the following example):

```
PIL reports available from CrossView Pro for block: fuelsys
Coverage ("covinfo"):  Yes (pil_coverage_report)
Profiling ("proinfo"): Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"):  Yes
(pil_cumulative_profiling_report)

Maximum stack usage during PIL (including the PIL test
framework overhead):

C166 User Stack: 59/109 (54.13%) words used.
```

To view the profiling report, click the hyperlink `pil_profiling_report`:

```
pil_profiling_report =

Total Execution Time:  4447016
                                          Cycles %Cycles
Function: pilInitialize                       16 0.000%
Function: pilGetUDataSymbol                22428 0.504%
Function: pilStep                          20826 0.468%
Function: pilGetYDataSymbol                22428 0.504%
Function: pilTerminate                        16 0.000%
Function: pilDataBreakpoint                14454 0.325%
Function: pilReadData                     549878 12.37%
Function: pilWriteData                    166816 3.751%
Function: pilDataInit                          4 0.000%
Function: getNextSymbol                    80100 1.801%
```

```
Function: processData                          288360 6.484%
Function: pilCommandLoop                       137966 3.102%
Function: main                                   6432 0.145%
Function: Sens_Failure_Counter                  22400 0.504%
Function: Fueling_Mode                          54740 1.231%
Function: Init_controllogic                        58 0.001%
Function: controllogic                         121744 2.738%
Function: fuelsys1_step                        677674 15.24%
Function: fuelsys1_initialize                      48 0.001%
Function: fuelsys1_terminate                        4 0.000%
Function: BINARYSEARCH_S16                     372458 8.375%
Function: DotProduct_s32s16                     37642 0.846%
Function: INTERPOLATE_EVEN_S16_S16_SAT          51678 1.162%
Function: INTERPOLATE_S16_S16_SAT              528118 11.88%
Function: Look2D_S16_S16_S16_SAT               256320 5.764%
Function: div_s32_sat_floor                    406596 9.143%
Module:   temp                                      0 0.000%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim_pil\pil_interface.c
    65714 1.478%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\pilsrc\...
pil_ide_data_stream.c   731152 16.44%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\pilsrc\...
pil_interface_lib.c   506426 11.39%
Module:   ..\..\..\..\..\sandbox\targets with spaces\matlab...
\toolbox\rtw\targets\common\tgtcommon\...
pilsrc\pil_main.c     6432 0.145%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\...
fuelsys1.c   876668 19.71%
Module:   MEMCPY_C                             357522 8.040%
Module:   MEMSET_C                                792 0.018%
Module:   MUL                                   13398 0.301%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
binarysearch_s16.c   372458 8.375%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
dotproduct_s32s16.c    37642 0.846%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
interpolate_even_s16_s16_sat.c    51678 1.162%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
```

```
interpolate_s16_s16_sat.c   528118 11.88%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
look2d_s16_s16_s16_sat.c   256320 5.764%
Module:   ..\..\slprj\ert_int_c167cs_sim\_sharedutils\...
div_s32_sat_floor.c   406596 9.143%
Module:   UDIL                              147648 3.320%
Module:   UMOL                               85064 1.913%
Module:   fuelsys1_pil                           0 0.000%
Module:   CSTART                                 0 0.000%
Module:   ..\..\fuelsys1_ert_rtw_int_c167cs_sim\...
fuelsys1_data.c        0 0.000%
27:     readDataPtr = & pil_ide_data_buffer[0];
```

For cumulative profiling, command line messages like the following inform you that you must configure CrossView Pro to specify which functions to collect data for. Select **Tools > Cumulative Profiling Setup**, specify functions, and then run the simulation again to get the report.

```
NOTE: Cumulative profiling requires manual setup in
CrossView Pro.
See Tools->Cumulative Profiling Setup
DO NOT add the function pilDataBreakpoint to the list of
functions to profile.

You must then run the PIL simulation again
to generate the report.

pil_cumulative_profiling_report =

CrossView Cumulative Profiling Report
-------------------------------------
Total Execution Time:  4447016
Function                            Calls    Recursive
Min.Time   Max.Time   Avg.Time   Total Time %Time
```

For information on build messages containing links at the command line, see “Command Line Project Information” on page 46-35.

Task Execution Profiling Kit

This kit, available on MATLAB Central, provides instructions and examples on how to implement real-time task based execution profiling on a custom target. A graphical representation of on-target execution and a HTML report are provided for analysis. You can implement this for your own custom system target file that uses the project generator.

For details, see http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12731

Stack Profiling

- "What Is Stack Profiling?" on page 46-63
- "PIL Applications" on page 46-63
- "Non-PIL Applications" on page 46-64
- "Infineon® TriCore Stack Depth Analyzer" on page 46-65

What Is Stack Profiling?

Stack profiling gives you a maximum bound on the stack usage of an application. The stack profiling feature works by first writing a signature to the stack memory region, then when the application executes normally, the signature pattern is overwritten by the application stack data. Finally the stack memory is read into MATLAB and analyzed to determine how much of the stack memory was used during execution.

PIL Applications

Stack profiling is automatically reported after PIL simulation. The report gives you a maximum bound on the stack usage of the algorithm under test.

Output at end of PIL (bold indicates hyperlinks):

```
Maximum stack usage during PIL (including the PIL test framework overhead):
TriCore User Stack: 24/2048 (1%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.
```

The hyperlinks for the individual stacks expand to more information about that stack, as shown in the following example.

```
PIL reports available from CrossView Pro for block: fuelsys

Coverage ("covinfo"):              No
Profiling ("proinfo"):             Yes (pil_profiling_report)
Cumulative profiling ("cproinfo"): Yes (pil_cumulative_profiling_report)

Maximum stack usage during PIL (including the PIL test framework overhead):

TriCore User Stack: 24/2048 (1%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.

         name: TriCore User Stack
  baseAddress: 0xA0080130 (2684879152 decimal)
   endAddress: 0xA008212F (2684887343 decimal)
      memSize: 0x2000 (8192 decimal) memory units
growDirection: down
  memorySpace: N/A
```

The hyperlink for "including the PIL test framework overhead" expands to show this explanation:

PIL Test Framework Overhead: The maximum stack usage reported after PIL is the stack usage of the entire PIL application, which includes a small amount of stack used by the PIL test framework. The stack usage reported is therefore a maximum bound on the stack usage of the algorithm under test.

To more accurately determine the stack usage of the algorithm it is possible to use the Embedded Coder CrossView Pro stack profiling feature on an application that is not configured for PIL. This will allow the stack usage to be determined without the stack overhead of the PIL test framework.

Non-PIL Applications

Non-PIL applications (perhaps with stimulus signals coming from target I/O drivers) can be profiled using the CrossView Pro API commands `stackProfileReset` and `stackProfile`.

1 Call `stackProfileReset` to reset the application you are debugging, and write a signature pattern to the stack memory region. Use the following syntax:

   ```
   xview.stackProfileReset
   ```

where *xview* is a `tasking.xviewapi` object. See “Methods for Class tasking.xviewapi” on page 46-48.

2 Call `stackProfile` immediately after resetting to view 0% stack usage profiling results.

3 Execute the application (e.g., `xview.execute('C')`).

4 After the amount of time you want to profile for, stop the application using `xview.halt`

5 Call `stackProfile` to get the profiling results for the execution period.

An example of this procedure is shown following.

```
>> XView_Obj.stackProfileReset
CrossView Pro: A hardware reset occurred.
CrossView Pro: A software reset of the program occurred.
>> XView_Obj.stackProfile
Maximum stack usage since last profile reset:

TriCore User Stack: 0/2048 (0%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.

>> XView_Obj.execute('C');
CrossView Pro: The target is running.
>> XView_Obj.halt
CrossView Pro: The target stopped executing with cause: "UNKNOWN"
CrossView Pro: Command with sequence number 71 completed.
>> XView_Obj.stackProfile
Maximum stack usage since last profile reset:

TriCore User Stack: 4/2048 (0%) words used.
TriCore Interrupt Stack: 0/256 (0%) words used.
```

Infineon TriCore Stack Depth Analyzer

The Infineon TriCoreStack Depth Analyzer (SDA) tool is a static stack depth analyzer for the TASKING TriCore toolset. This is an alternative to the dynamic stack profiling provided with Embedded Coder software.

It can be found at the Infineon TriCore Software Downloads page. Navigate there from this URL:

```
http://www.infineon.com/tricore
```

Click the link on the right: “Development Tools, Software and Training”, then click “Software Downloads”.

Bidirectional Traceability Between Code and Model

- "Using Traceability" on page 46-66
- "Enabling Traceability" on page 46-67

Using Traceability

Context menu items and command-line methods allow you to navigate bidirectionally between Simulink blocks and the corresponding generated source files in the TASKING EDE or the CrossView Pro debugger.

See the demo, tasking_demo_objects, to try this feature. This is a command line demo that you can run from the Help browser.

To find the generated code for any block in the model, right click on the block and select: **IDE Link > See Generated Code in EDE** or **IDE Link > See Generated Code in CrossView Pro**.

This opens the source file which contains the generated code for the block, and highlights the Simulink Coder tag for that block. The Simulink Coder tag is usually found in the block's generated comments preceding the block's code.

There are command-line alternatives to the right-click context menu items — see tasking_demo_objects for an example.

To find the block which corresponds to some generated code in the EDE or CrossView Pro:

1. Click to place the cursor at the line of code containing the Simulink Coder Tag for the given block. Here is an example:

   ```
   /* Outputs for atomic SubSystem: '<Root>/SS2'
   ```

2. Enter at the MATLAB command prompt:

   ```
   EDE_Obj.hilite_system
   ```

   or

   ```
   XView_Obj.hilite_system
   ```

Enabling Traceability

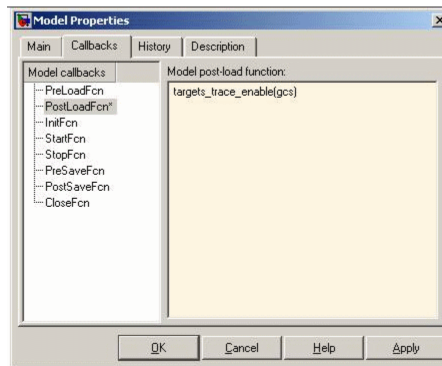
To use the Traceability feature, you must configure your model as follows:

1. Enable the generation of traceability information by adding `targets_trace_enable(gcs)` to the `PostLoadFcn` callback of the model.

   a. Select **File > Model Properties > Callbacks**.'

   b. Click `PostLoadFcn`.

   c. Enter `targets_trace_enable(gcs)`, as shown.

   Click **OK**.

   Note `targets_trace_enable` also selects the check box options **Create Code Generation report** and **Code-to-model** under **Report** in the Configuration Parameters dialog box.

2. The model must use an ERT based Target.

MISRA C Rule Checking

The TASKING C compiler supports MISRA C rule checking and can be easily configured to check the code generated by Simulink Coder software.

You can switch on MISRA C rule checking in your application and/or library template projects. When you build using these template projects,

the TASKING compiler will provide warnings about MISRA C violations. Embedded Coder software returns these warnings to the MATLAB command line for your review.

Embedded Coder software provides an example application project template, pre-configured for MISRA C rule checking, for the TASKING TriCore Toolset. For instructions, see the MISRA C Rule Checking demo, `tasking_demo_misra.m`.

Optimization

In this section...

"Compiler / Linker Optimization Settings" on page 46-69

"Target Memory Placement / Mapping" on page 46-69

"Execution and Stack Profiling" on page 46-70

"Target Specific Optimizations" on page 46-70

"Model Advisor" on page 46-74

Compiler / Linker Optimization Settings

Template projects allow you to fully control the optimization settings used by the compiler and linker.

- See "Tutorial: Creating New Template Projects" on page 46-76 for details of using template projects.
- See PIL Block Parameters on page 55 for information about optimization setting requirements for Processor-in-the-Loop.
- See the TASKING documentation for details of available optimization settings.

Target Memory Placement / Mapping

Template projects allow you to fully control the target memory map used for your application.

- See "Overview of the Project Generator Component" on page 46-27 for a general discussion of how the code generation process and subsequent build process work together, including a memory placement example.
- See "Tutorial: Creating New Template Projects" on page 46-76 for details of using template projects. See the TASKING documentation for details of memory map settings.

Execution and Stack Profiling

- “Execution Profiling” on page 46-70
- “Stack Profiling” on page 46-70

Execution Profiling

Execution profiling metrics from the CrossView Pro instruction set simulator during PIL simulation can be used to identify areas of your algorithms that can be further optimized.

See “Execution Profiling” on page 46-60 for details.

Stack Profiling

Stack profiling metrics for PIL simulation or real-time applications can be used to optimize the amount of stack memory required for an application.

See “Stack Profiling” on page 46-63 for details.

Target Specific Optimizations

- “C Language Extensions / Intrinsics” on page 46-70
- “Target Optimized Libraries for Infineon XC166 and Infineon® TriCore” on page 46-72

C Language Extensions / Intrinsics

Infineon TriCore.

Support	C89/C90 ANSI Target Function Library	Infineon TriCore ISO Target Function Library	Infineon TriCore Target Function Library (ERT Only)
ANSI Support	Yes	Yes	Yes

Support	C89/C90 ANSI Target Function Library	Infineon TriCore ISO Target Function Library	Infineon TriCore Target Function Library (ERT Only)
ISO Support		Yes	Yes
Saturated Arithmetic Support			Yes

ISO/IEC 9899:1999 Math Library

The target function library `Infineon TriCore ISO`uses the TASKING ISO/IEC 9899:1999 Math Library to implement floating-point mathematical function blocks (e.g. trigonometric functions, log functions). Using these target optimizations improves the performance of applications performing floating-point mathematical operations.

When using these target optimizations, the regular Simulink Coder implementation for many ANSI floating-point mathematical operations is replaced by the ISO equivalent. These functions behave identically to the regular Simulink Coder implementation and can be verified using processor-in-the-loop simulation.

You can use the `Infineon TriCore ISO` target function library with ERT or GRT system target files.

To enable the math library for the optimization of floating-point mathematical operations, select `Infineon TriCore ISO` for the Simulink Coder option **Target function library** (on the **Interface** pane of the Configuration Parameters dialog box).

Saturated Arithmetic

The target function library `Infineon TriCore` includes all ISO optimizations and also saturated arithmetic optimizations. The target function library `Infineon TriCore` is only available for ERT system target files.

You can use TASKING compiler extensions and intrinsic functions for saturated arithmetic. These target optimizations can increase execution speed up to 18 times for saturated arithmetic operations. The use of these target optimizations will improve the performance of most applications performing saturated arithmetic operations. It is therefore recommended to enable the optimizations.

When using these target optimizations, the regular Simulink Coder implementation for many saturated arithmetic operations are replaced by calls to target optimized inlined functions. The behavior of these functions is identical to the regular Simulink Coder implementation and can be verified using processor-in-the-loop simulation (see “Processor-in-the-Loop (PIL) Simulation” on page 46-50).

To enable TASKING compiler extensions and intrinsic functions for the optimization of saturated arithmetic, select `Infineon TriCore` for the Simulink Coder option **Target function library** (on the **Interface** pane of the Configuration Parameters dialog box).

General. Depending on your toolset, your the TASKING compiler may support C language extensions or intrinsics to help optimize in some of the following areas:

- Data Types (eg. Fractional Arithmetic, Bit Addressable Memory)
- Memory Qualifiers (eg. Near, far address space)
- Data Type Qualifiers (eg. Circular Buffers, Saturated arithmetic)

Please see your TASKING documentation for details. You can use these language extensions in your own Simulink blocks and / or custom code.

Target Optimized Libraries for Infineon XC166 and Infineon TriCore

The following optimized libraries are available for the processors supported by Embedded Coder software, and can be used to create optimized Simulink blocks:

- Infineon XC166 DSP Library for TASKING compiler

  This library is described by Infineon as follows:

XC166 DSP library is a DSP function library, is C-callable, manually coded assembly, general purpose signal processing routines:

- Arithmetic Functions
- Filters (FIR–, IIR–, Adaptive Filters)
- Transforms (FFT, IFFT)
- Matrix Operations
- Mathematical Operations
- Statistical Functions

See the Infineon C166 Software Downloads Web page to get the XC166 DSP Library. Navigate there from this URL:

```
http://www.infineon.com/c166
```

Click the link on the right: "Development Tools, Software and Training", then click "Software Downloads".

• Infineon TriCore DSP Library (TriLib)

This library is described by Infineon as follows:

TriLib is a DSP Library for TriCore, containing more than 60 commonly used DSP routines for

- Complex & Vector Arithmetic
- FIR, IIR, Adaptive Filters
- Fast Fourier, Discrete Cosine Transform
- Mathematical, Matrix, Statistical functions

See the Infineon TriCore Software Downloads page to get the TriLib DSP Library. Navigate there from this URL:

```
http://www.infineon.com/tricore
```

Click the link on the right: "Development Tools, Software and Training", then click "Software Downloads".

Model Advisor

Following the suggestions in the Model Advisor report may result in faster on-target execution. See "Consulting the Model Advisor" in the Simulink documentation.

Tutorials

In this section...

"Tutorial: Using Option Sets" on page 46-75

"Tutorial: Creating New Template Projects" on page 46-76

"Tutorial: Configuring an Existing Model for Embedded Coder Software" on page 46-81

Tutorial: Using Option Sets

Option sets are preconfigured settings to specify the target configuration for the TASKING tools. You use option sets to apply EDE project settings (e.g., compiler and linker settings, hardware or simulator) that you can then modify if you choose. For example, once you have set up your target preferences for a TriCore configuration, you can use option sets to switch between using an instruction set simulator configuration, two hardware board configurations, or a simulator with some MISRA C rule checking.

To choose an option set:

1. Enter `taskingutils` in the Command Window.

   The IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select a target configuration (e.g., `C166`, `TriCore`) from the list in the dialog box, and click **OK**.

   The Option Set Selection dialog box appears.

4. Select an option set. The list items are specific to the configuration you selected; the available option sets are listed in "Option Sets" on page 46-24. Click **OK**.

Your target preferences are automatically updated according to the option set you select, and command line messages inform you the following target preferences have changed:

- `EDE_Configuration`

  `Template_Application_Project`: Set to default template application project relating to the option set.

  `Template_Library_Project`: Set to default template library project relating to the option set.

- `CrossView_Pro_Configuration`

  `Initialization_File`: Set to CrossView Pro (`.st`) initialization file relating to the option set.

Now, when you build any model configured for the same target (e.g., `TriCore`), these project settings are used. To switch to a different option set, repeat the steps.

You can also use option sets to set up an initial configuration when creating new template projects. See "Tutorial: Creating New Template Projects" on page 46-76.

Tutorial: Creating New Template Projects

- "Creating New Template Projects" on page 46-76
- "Creating a New Configuration" on page 46-80

Creating New Template Projects

In this tutorial, you create new template projects for a target configuration and set up options such as simulator or hardware implementation, compiler and linker settings, MISRA C rule checking, or any other project options. Every time you build a model for the selected target configuration, the project options you have set up in the new template projects are used.

Note You may want to create a new configuration to use with new template projects. See the next section, “Creating a New Configuration” on page 46-80 for details.

To create custom application and library template projects:

1. Enter `taskingutils` in the Command Window.

   The IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Create New Template Projects` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select your target (e.g., `TriCore`), and click **OK**.

   Your target preferences for the location of your TASKING installation must be set up for the target configuration you choose (see “Setting Target Preferences for Altium TASKING” on page 46-8).

   a. Make sure the fields are filled in for this configuration (except the Application and Library Template Projects fields, and CrossView Initialization field, which are autopopulated during the following steps).

   b. If your target preferences are set up correctly, your TASKING EDE launches when you click **OK**.

4. When you are prompted, choose a location for the template projects, and enter the template name.

5. When you are prompted, choose an option set. An *option set* delineates options specific to your target, such as whether you want to use the simulator or hardware. You can use these to set up an initial configuration to modify later. See “Option Sets” on page 46-24 for more information and a list of available option sets.

   You now have custom template projects for this new configuration. The EDE project settings associated with the option set are applied to the new

template projects. Messages at the command line inform you the following target preferences have been automatically updated:

- EDE_Configuration

  `Template_Application_Project`: Set to new template application project configured by the option set.

  `Template_Library_Project`: Set to new template library project configured by the option set.

- CrossView_Pro_Configuration

  `Initialization_File`: Set to CrossView Pro (`.st`) initialization file configured by the option set.

Note You can always choose a preconfigured option set (from the drop-down list of the **Configuration** field) to return to the default settings.

Next, modify the compiler settings for these new template projects.

6 To modify the template projects, you need to open them in the TASKING EDE:

   a Enter `taskingutils` in the Command Window.

   The IDE Link Utilities for Use with Tasking dialog box appears.

   b Select `Open Existing Template Projects` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

   c Select the same target for which you created new template projects, and click **OK**.

   The template projects should now be open in the EDE.

Note Opening or making changes to template projects causes the regeneration of application and library projects.

---

d Right-click the project in the TASKING EDE, and select **Project Options**. You can now modify the project options (compiler settings, linker settings, etc.).

Note When making any changes to template projects, it is important to remove the project from the project space, to make sure your changes are written to disk. Otherwise the changes may not be applied immediately. To remove a current project from the project space, right-click on it and choose **Remove from Project Space**.

---

e When done, close the template projects in the TASKING EDE.

7 To modify your CrossView Pro configuration (optional) you need to specify a `.ini` file in the `Initialization_File` Target Preference field. See `Initialization` in the section "Target Preference Fields" on page 46-10.

You are now ready to use the configuration.

8 Open any Simulink model that is configured with the Embedded Coder component (`tasking_demo_fuelsys`, for example).

9 Select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.

10 Select **IDE Link** on the left-side panel. When you select your target in the **Target Preference Configuration** menu, the template projects you have set up are used.

See "Tutorial: Creating New Template Projects" on page 46-76 for details about how Embedded Coder software uses template projects during the build process.

Creating a New Configuration

You can customize the default Target Preference configurations by choosing from the preconfigured options sets, or by creating new template projects.

However, it may be useful to create a new Target Preference configuration if you want to switch between them in the **Target Preference Configuration** menu. For example, if your target is a TriCore processor, you could set up a new configuration called `TriCore_user` to specify hardware settings for your target; then you can easily switch between `TriCore` (the default instruction set simulator configuration) and `TriCore_user` using the **Target Preference Configuration** menu in your model's Configuration Parameters dialog box.

In this tutorial, you create a new TASKING configuration and save it in the target preferences. You can then use your new configuration in any Simulink model that is configured with Embedded Coder software by selecting it in the **Target Preference Configuration** menu.

To create a new configuration:

1. Enter `taskingutils` in the Command Window.

   The IDE Link Utilities for Use with Tasking dialog box appears.

2. Select `Target Preferences` from the list in the dialog box, and click **OK**.

   The Target Preferences Configuration Selection dialog box appears.

3. Select `Create new Configuration`, and click **OK**.

4. Expand `Configuration_Options`.

5. Type `Tutorial` in the **Configuration_Description** field.

6. Fill in the rest of the fields for this configuration. See "Setting Target Preferences for Altium TASKING" on page 46-8 to set these fields properly.

   a. You must specify the location of your toolset, by filling in the path to the `CrossView_Pro_Executable`, the `DOL_File`, and the `EDE_Executable`.

   b. You can set up the template projects and CrossView initialization fields automatically in one of two ways:

- You can use the **Start** menu option **Select Preconfigured Target Preference Settings**. See "Tutorial: Using Option Sets" on page 46-75 for instructions.
- You can create new template projects for this configuration. See "Tutorial: Creating New Template Projects" on page 46-76.

  If you are going to use either of these options you can leave the template projects and CrossView initialization fields blank, because they will be filled in automatically when you follow the steps in using option sets or creating new template projects.

  Click **OK** to close and save your target preferences.

7. After you save your target preferences, you can use the new `Tutorial` configuration in any model that is configured with Embedded Coder software. For example, open any of the Embedded Coder demo models (such as `tasking_demo_fuelsys`).
8. Select **Simulation > Configuration Parameters**. The Configuration Parameters dialog box opens.
9. Select **IDE Link** on the left-side panel. Click the **Target Preference Configuration** menu, and notice that the `Tutorial` configuration now appears in the list.

Tutorial: Configuring an Existing Model for Embedded Coder Software

In this tutorial, you configure an existing fixed-point model and build it with Embedded Coder software.

1. At the MATLAB command prompt, type `rtwdemo_fixptdiv` to open a fixed-point demo model.
2. Switch the model to use Embedded Coder software as follows:
   a. Select **Simulation > Configuration Parameters**, and click **Code Generation**.
   b. Click **Browse** and select `ert.tlc` (first item in the list). Click **OK**.

3 Select **Tools > IDE Link > Add IDE Link Configuration to Model** to add the Embedded Coder configuration set to the model.

4 Open the Configuration Parameters dialog box again from the **Simulation** menu, and verify that the Embedded Coder configuration set is now added to the model. Select **IDE Link** from the left panel:

   a Set the **Build Action** to `Create and Build Application Project`.

   b Select the **Target Preference Configuration** to match your target.

   c Select the check box option to **Add Build Directory Suffix**, and type `int` in the **Build Directory Suffix** field.

   d Under the **Code Generation** options, select **Interface** and clear the check box for **floating-point numbers** support under **Software environment**, because this model is fixed point. Clearing this option instructs Simulink Coder software to avoid building the floating-point version of the `rtwlib` library.

   e Under **Code Generation**, select **Hardware Implementation**, and select your device type. For example:

   - For C166 platforms, select `Infineon C16x, XC16x`.
   - For TriCore platforms, select `Infineon TriCore`.
   - For ARM platforms, select `ARM 7/8/9`.
   - For Renesas M16C, 8051 Compatible, or Freescale DSP563xx (16-bit mode) platforms, select those options.

   You are now ready to build the model. Press **Ctrl+B** or select **Tools > Code Generation > Build Model**.

Code Generation Pane — IDE Link

In this section...

"Overview" on page 46-84
"Build Action" on page 46-85
"Target Preference Configuration" on page 46-87
"Add build directory suffix" on page 46-88
"Build directory suffix" on page 46-89
"Export EDE handle to MATLAB base workspace" on page 46-90
"EDE handle name" on page 46-90
"Export CrossView Pro handle to MATLAB base workspace" on page 46-92
"CrossView Pro handle name" on page 46-92
"Configure model to build PIL algorithm object code" on page 46-94

Overview

Parameters for controlling Embedded Coder build configuration, export handles, and processor-in-the-loop verification.

Configuration

This pane appears if you add the Embedded Coder configuration options to a model with any system target file. To do this, select the menu item **Tools > Utilities for use with TASKING(R) IDE > Add IDE Link Configuration to Model**.

See Also

Working with Configuration Sets

Build Action

Set what action to take after the Simulink Coder build process completes. You can create application and library projects in the TASKING EDE and then stop, or you can also choose to build, execute, or debug.

Settings

Default: `Create Application Project`

`Create Application Project`
: Generate code for the model or subsystem, create a TASKING application project for the selected TASKING configuration, connect to the TASKING EDE, and open the application project (in addition to the required Simulink Coder and DSP System Toolbox Library projects, if required) in the TASKING EDE. This option does not build or execute the application.

`Create Library Project`
: Performs the same actions as Create Application Project, but this option archives the generated code into a library in TASKING. No `main.c` file is generated.

`Create and Build Application Project`
: Performs the same actions as Create Application Project, but also instructs TASKING to build the application project.

`Create and Build Library Project`
: Performs the same actions as Create Library Project, but also instructs TASKING to build the Library project.

`Create, Build and Execute Application Project`
: Performs the same actions as Create and Build Application Project and also downloads the executable file to your CrossView Target and runs the executable. No debugging information is downloaded into the target with this option.

`Create, Build and Debug Application Project`
: Performs the same actions as Create, Build and Execute Application Project but also downloads debugging information to the target. This option behaves the same way as the Debug Application icon in the TASKING EDE.

Tip

To manually debug the executable from the application project, use the Create and Build Application Project option, then click the Debug Application icon in the TASKING EDE

Dependency

This parameter is disabled by **Configure model to build PIL algorithm object code**.

Command-Line Information

Parameter: `TaskingBuildAction`
Type: string
Value: `'Create Application Project'` | `'Create Library Project'` | `'Create and Build Application Project'` | `'Create and Build Library Project'` | `'Create, Build and Execute Application Project'` | `'Create, Build and Debug Application Project'`
Default: `'Create Application Project'`

Recommended Settings

Application	Setting
Debugging	'Create, Build and Debug Application Project'
Traceability	No impact
Efficiency	No impact
Safety precaution	No impact

See Also

Setting Build Action

Target Preference Configuration

Select a configuration description, as defined in the Target Preferences, to be used by the build action.

Settings

Default: `'Target Preference Configuration Not Set'`

After you have set up target preferences for particular configurations, you can select them here (e.g., `'c166'`). The names in the list correspond to the Configuration Description for each configuration in the IDE Link Target Preferences dialog box. Click **Edit Configuration** to open the IDE Link Target Preferences dialog box for the currently selected configuration. For instructions, see Using Configuration Sets to Specify Your Target.

Command-Line Information

Parameter: `TaskingConfiguration`
Type: string
Value: `'Target Preference Configuration Not Set' |` Any "Configuration_Description" name defined in the IDE Link Target Preferences (e.g. `'TriCore'`, `'C166'`, etc.)
Default: `'Target Preference Configuration Not Set'`

See Also

- Using Configuration Sets to Specify Your Target

Add build directory suffix

Specify whether to add a model-specific suffix to the regular Simulink Coder build folder suffix.

Settings

Default: Off

On

Specify a model-specific suffix to be added the regular Simulink Coder build folder suffix. This setting is useful to avoid "shared utility function" code generation errors which occur because of conflicts over Simulink Coder utility functions shared between different models. A typical conflict is between with models with floating-point number support and those without. To resolve this conflict, you can add an 'fp' suffix for floating-point models, and an 'int' suffix for non-floating-point models.

Off

Use the default Simulink Coder build folder suffix — not using an additional suffix may result in rebuilding shared libraries unnecessarily.

Dependencies

This parameter enables **Build directory suffix**.

Command-Line Information

Parameter: `TaskingSpecifyBuildSubDirName`
Type: logical
Value: `0 | 1`
Default: `0`

Recommended Settings

Application	Setting
Debugging	No impact
Traceability	No impact

Application	Setting
Efficiency	On
Safety precaution	No impact

See Also

Shared Libraries

Build directory suffix

Specify a model-specific suffix to be added the regular Simulink Coder build folder suffix.

Settings

No Default

Enter a model-specific suffix to be added the build folder name. This setting is useful to avoid "shared utility function" code generation errors which occur because of conflicts over Simulink Coder utility functions shared between different models.

Dependencies

This parameter is enabled by **Add build directory suffix**.

Command-Line Information

Parameter: `TaskingBuildSubDirName`
Type: string
Value: Any string composed of the following characters: [a-z_A-Z0-9]
Default: `''`

See Also

Shared Libraries

Export EDE handle to MATLAB base workspace

Specify whether to export the EDE object handle to the workspace.

Settings

Default: On

☑ On

Export a TASKING EDE object handle to the MATLAB base workspace after the build process completes.

☐ Off

Do not export the EDE object handle to the workspace.

Dependencies

This parameter enables **EDE handle name**.

Command-Line Information

Parameter: `TaskingExportEDEHandle`
Type: logical
Value: `0 | 1`
Default: `1`

See Also

Automation Interface

EDE handle name

Specify a name for the exported handle.

Settings

Default: 'EDE_Obj'

Specify the MATLAB base workspace variable name to export the handle to.

Dependencies

This parameter is enabled by **Export EDE handle to MATLAB base workspace**.

Command-Line Information

Parameter: `TaskingExportEDEHandleName`
Type: string
Value: Any valid MATLAB variable name (see MATLAB function: `isvarname`)
Default: `'EDE_Obj'`

See Also

Automation Interface

Export CrossView Pro handle to MATLAB base workspace

Specify whether to export the CrossView Pro object handle to the workspace.

Settings

Default: On

On

Export the TASKING CrossView Pro object handle to the MATLAB base workspace after the build process completes.

The handle is only exported if the build action launches CrossView Pro.

Off

Do not export the CrossView Pro object handle to the workspace.

Dependencies

This parameter enables **CrossView Pro handle name**.

Command-Line Information

Parameter: `TaskingExportCrossViewHandle`
Type: logical
Value: `0 | 1`
Default: `1`

See Also

Automation Interface

CrossView Pro handle name

Specify a name for the exported handle.

Settings

Default: ’XView_Obj’

Specify the MATLAB base workspace variable name to export the handle to.

Dependency

This parameter is enabled by **Export CrossView Pro handle to MATLAB base workspace**.

Command-Line Information

Parameter: `TaskingExportCrossViewHandleName`
Type: string
Value: Any valid MATLAB variable name (see MATLAB function: `isvarname`)
Default: `'XView_Obj'`

See Also

Automation Interface

Configure model to build PIL algorithm object code

Specify whether to build Processor-in-the-Loop (PIL) algorithm code.

Settings

Default: Off

On

Configure the model to build PIL algorithm code that is suitable for use with the PIL block.

Off

Do not build PIL algorithm code.

Dependency

This parameter disables **Build action**.

See Also

Processor-in-the-Loop (PIL) Simulation

Limitations and Tips

In this section...

"General Issues" on page 46-95

"Debugger Issues" on page 46-97

"Build Process Issues" on page 46-98

"Processor-in-the-Loop Issues" on page 46-107

"Issues Using Simulink® Coder Software Without Embedded Coder Software" on page 46-110

General Issues

- "IDE Link TASKING requires User Account Control (UAC) to be disabled when using Windows 7" on page 46-95
- "Problems with Installations in Read-Only Locations" on page 46-95
- "Simulink Configuration Set Reference Not Supported" on page 46-96
- "Serialization of Embedded Coder Objects Not Supported" on page 46-96
- "Avoid Spaces in Environment Variables Used by `tempname` and `tempdir` Functions" on page 46-97

IDE Link TASKING requires User Account Control (UAC) to be disabled when using Windows 7

Communication between MATLAB and the TASKING CrossView Pro debugger requires Windows 7 UAC to be disabled. You can disable UAC by adjusting the settings in the Windows 7 User Account Control Settings dialog and restarting your computer. Alternatively, you can disable UAC for a single Command Prompt session (and any MATLAB processes started from that session) by right-clicking on the Command Prompt icon and selecting "Run as administrator". Note that only Windows 7 32-bit is supported.

Problems with Installations in Read-Only Locations

The process to build models works correctly when Embedded Coder software is installed in a read-only location because the template projects are copied

to the working folder during the process. However, installing Embedded Coder software in a read-only location (e.g. read-only network) causes the following problems:

- Template project generation fails because the function tasking_generate_templates attempts to write to the installation location.
- Opening existing template projects may fail because the TASKING EDE attempts to write to the installation location.

To resolve this issue:

- Do not install Embedded Coder software in a read-only location
- Avoid updating or opening the template projects or temporarily allow write access to the read-only installation location while doing so.
- Create new template projects in a writable location rather than attempting to update the default template projects.

Simulink Configuration Set Reference Not Supported

The Simulink Configuration Set Reference feature is not supported by Embedded Coder software.

For Embedded Coder software, make sure your model's configuration set objects are "Simulink.ConfigSet" objects and not "Simulink.ConfigSetRef" objects.

Serialization of Embedded Coder Objects Not Supported

Serialization (saving and loading to MATLAB .mat file) of the objects provided with Embedded Coder software (e.g., tasking.edeapi, tasking.xviewapi) is not possible. If you attempt to load a serialized object from a .mat file you may see the Target Preferences Configuration Selection GUI, warning or error messages, or both.

In some circumstances, a product (for example the SystemTest™ product) or a user script may automatically save all contents of the MATLAB base workspace to a .mat file. In this case, it may be useful to turn off the "Export Handles" settings in the Embedded Coder configuration set component. Doing

so stops EDE and CrossView Pro objects from being exported to the base workspace at the end of a Embedded Coder build process and thus avoids potential serialization problems.

Avoid Spaces in Environment Variables Used by `tempname` and `tempdir` Functions

The MATLAB functions `tempname` and `tempdir` use the TEMP or TMP environment variables. If the TEMP or TMP environment variables contain spaces, in Embedded Coder with TASKING, the software does not support spaces in the TEMP or TMP environment variables. The `tempname` and `tempdir` functions use the TEMP or TMP environment variables.

For example, if you use `tempdir` and the TEMP environment variable contains a space, the software creates a corrupt filename.

Debugger Issues

- "ARM CrossView Pro Debugger Fails with File | Open Source Content" on page 46-97
- "On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware" on page 46-98

ARM CrossView Pro Debugger Fails with File | Open Source Content

Due to a CrossView Pro bug, the File | Open Source menu item of the ARM CrossView Pro debugger may fail to open the specified source file. Instead, you may see a blank window or the wrong source file may be opened.

This limitation can affect the Traceability feature from the model to the code in CrossView. If you right-click on a block in the Simulink model and select **IDE Link > See Code in CrossView Pro**, this operation might not work as expected because the source file cannot be opened.

To work around this issue, you can set a breakpoint in the source file that is initially visible during debugging and step into other source files from there.

On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware

For ARM processors the CrossView Pro instruction set simulator can be used for debugging and processor-in-the-loop (PIL) simulation, but there is currently no on-chip debugging or PIL support.

To resolve this problem:

- You can contact TASKING for the latest information on CrossView Pro on-chip debugging support for ARM hardware.
- You can contact Hitex for a solution to debug an application generated by Embedded Coder software on ARM hardware, however this solution cannot provide PIL support.

Build Process Issues

- “Linker Errors Due to Limited Memory” on page 46-99
- “EDE Is Slow, Unresponsive, or Crashes” on page 46-99
- “DSP System Toolbox Library Build Failures” on page 46-100
- “Memory Block Freed Twice Error” on page 46-100
- “8051 EDE Cannot Compile Files with Long Names” on page 46-100
- “DSP563xx Toolset Support Limitations” on page 46-101
- ““Create, Build and Execute Application Project” Build Action Fails” on page 46-101
- “C166 Toolset Warnings” on page 46-102
- “Build Error From Root Drive Location” on page 46-102
- “Limited Support for Nonfinite Values” on page 46-103
- “Memory Warning/Error Messages in the CrossView Pro Command Window When Using the Instruction Set Simulator” on page 46-105
- “C++ Code Generation Not Supported” on page 46-105

- “Computer Vision System Toolbox Library Not Supported” on page 46-106
- “Noninlined S-functions Calling rt_matrx.c Not Supported” on page 46-106
- ““Compiler optimization level” Configuration Parameter Has No Effect” on page 46-106
- “Configuration Changes Cause Build Errors With Referenced Models” on page 46-107

Linker Errors Due to Limited Memory

The Embedded Coder software supports a variety of targets (instruction set simulators and embedded hardware) with a range of capabilities. Some demo models and user-created models may fail to build for certain targets owing to a lack of available target memory. In such cases you see linker errors like the following:

```
Linking and locating to t_pil_lib_alg_pil.out
E 268: relative linear element 'section T_PIL_LIB_ALG_4_NB class
CNEAR' cannot be located within 4 pages
total errors: 1, warnings: 0
wmk: *** action exited with value 1.
```

To work around such errors you must do one of the following:

1. Modify the model to reduce memory requirements (for example, by optimizing the algorithm, or by using smaller datatypes).

2. Alternatively, modify the target configuration to make more memory available (for example, by using a hardware board with more memory, or changing the memory map to allow extra memory to be used).

In some cases it may not be possible to resolve the problem, because the algorithm represented by the model is too complex for the target.

EDE Is Slow, Unresponsive, or Crashes

Under certain circumstances the TASKING EDE may become slow, unresponsive, or even terminate with virtual memory problems. This limitation is an open issue with the TASKING EDE (for all supported tool suites).

To resolve this issue, take one or both of the following actions:

- Close the EDE and try building the model again
- Try deleting the symbol database file, `cwright.sbl`, which can be found in the EDE_Executable folder ($TASKINGRootDir\bin)

DSP System Toolbox Library Build Failures

The following problem has been found with DSP System Toolbox product (“`DSP lib`”) library builds.

With Renesas M16C, building the DSP System Toolbox library with floating point support enabled results in the following error:

```
TASKING program builder v3.1r1 Build 076 SN 00100552
Assembling qrdc_z_rt.src asm16c E219:
["qrdc_z_rt.src" 1692] expression out of range
(0 and FF hexadecimal)wmk:
*** action exited with value 1.
```

This limitation is a known issue with the Renesas 16C compiler. To resolve this issue, disable floating point support in the model.

Memory Block Freed Twice Error

Occasionally, when Embedded Coder software is creating projects in the TASKING EDE, the following error appears: `Memory block freed twice`. This limitation is a known issue with the TASKING EDE.

To work around the problem, click **OK** in the error dialog box, and the code generation process continues as normal.

8051 EDE Cannot Compile Files with Long Names

If you encounter this problem, you receive an error message similar to the following:

```
Assembling tasking_fuel_controller_ert_rtw_pil_cstart.src
asm51 E001: tasking_fuel_controller_ert_rtw_pil_cstart.src: line 1:
syntax error
wmk: *** action exited with value 1.
```

This message indicates that the full path of the model or subsystem you are trying to build is too long.

To resolve this issue, consider moving the model to a shorter folder name, or renaming the model, subsystem, or both to use shorter names.

DSP563xx Toolset Support Limitations

The following limitations affect use of the DSP563xx Toolset:

- Only 16-bit mode for the DSP563xx Family is supported. Simulink Coder `grt.tlc`-based targets and the "GRT Compatible Call interface" option in the **Code Generation > Interface** settings are not supported. This limitation is because of the non-standard size of single- and double-precision floating-point datatypes on this architecture (`tmwtypes.h` will not compile)
- The DSP5600x Toolset is NOT supported because none of the processors supported by this toolset have 16-bit memory models.
- Both 16-bit memory models of the DSP563xx Family produce watch errors (wrong values displayed) in CrossView Pro because of an issue with the TASKING toolset. CrossView Pro does not know that the datatype sizes should be different according to the selected memory model. This issue does not affect the DSP566xx Family.

There are no resolutions for this issues.

"Create, Build and Execute Application Project" Build Action Fails

Tool Suites: Renesas M16C

With the Renesas M16C tool suite, if you are executing the application project, rather than debugging (via "Create, Build and Debug Application Project"), this does not work correctly. The application does not execute. This issue occurs because the CrossView Pro Simulator does not know the start address when debugging information is not loaded.

To resolve this issue, perform the following steps after CrossView Pro launches:

1 Stop execution by clicking the Halt button.

2 Execute the following command in the CrossView Pro command window to determine the application entry point stored at location 0xfffffc:

```
*((unsigned long *)0xfffffc)/x
```

Example output for this command is:

```
0xfffffc = 0x000d0000
```

3 Change the execution position to the application entry point by executing the "gi" command, using the output of the previous command. For example, `0xd0000 gi`

4 Resume execution by clicking the Run/Continue button.

Alternatively, use the “Create, Build and Debug Application Project” build action.

C166 Toolset Warnings

When using the C166 toolset you may see warnings similar to the following:

```
Warning: missing "sdc_lia" or "sdc_lip" lifetime record
```

This warning is caused by a problem with the TASKING toolset and has been registered with Altium as PR35043. It is related to debug life time information.

The warning can be ignored safely.

Build Error From Root Drive Location

On the C166 and 8051 platforms, a limitation of the TASKING toolset may cause build errors if you build from a root drive location such as `c:\` or `d:\`.

Following is an example error with the C166 toolset:

```
cc166: E 014: invalid control:
Files\MATLAB\R2007a_nortwec\toolbox\rtw\targets\c166\c166demos" -Wcp"-IC:\Program
wmk: *** action exited with value 1.
```

Workaround: Always build from a sub-folder location such as `c:\work` or `d:\MATLAB\work`.

Limited Support for Nonfinite Values

Nonfinite values in your model may cause wrong results, linking errors or compilation errors. See below for a possible workaround if your target is a TriCore or ARM platform.

Linking Errors. If you encounter similar linking errors when building your model:

```
undeclared identifier "rtMinusInf"
undeclared identifier "rtNaN"
undeclared identifier "rtInf"
```

then this means that:

- Your model uses nonfinite values, and
- You are using a stubbed version of `rt_nonfinite.c` which does not define `rtMinusInf`, `rtNan`, or the other nonfinite identifiers required by Simulink Coder software.

To resolve this issue:

- Do not use nonfinite values in the model. Such values are not desirable for embedded applications. Nonfinite elements on targets other than TriCore or ARM are not supported with Embedded Coder software.
- If you want to use nonfinite values and your target is a TriCore or ARM platform, then you can use the following workaround. You do not need to use a stubbed version of `rt_nonfinite.c` because the default one should compile correctly on this 32-bit target. In the configuration set, under **Code Generation** in **TLC Options**,

  1. Remove `-aCustomNonFinites="genrtnonfinite_stub.tlc"`.
  2. Delete the generated `rt_nonfinite.c` file in the build area before attempting to build the model again. This procedure should generate a new `rt_nonfinite.c` file that correctly defines the undeclared identifiers.

Compilation Errors. If you encounter compilation errors in `rt_nonfinite.c` similar to the following:

```
Compiling and assembling rt_nonfinite.c
..\..\slprj\ert_c167cs_sim\_sharedutils\rt_nonfinite.c:
    47:         uint32_T fraction : 23;
E 134: bitfield size out of range - set to 1
    57:           uint32_T fraction1 : 20;
E 134: bitfield size out of range - set to 1
    69:       (*(LittleEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
    78:         (*(LittleEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
    89:         (*(LittleEndianIEEEDouble*)&rtNaN).wordL.fraction1 = 0xFFFFF;
W 195: constant expression out of range -- truncated
    90:         (*(LittleEndianIEEEDouble*)&rtNaN).wordH.fraction2 = 0xFFFFFFFF;
W 196: constant expression out of range due to signed/unsigned type mismatch
    98:         uint32_T fraction : 23;
E 134: bitfield size out of range - set to 1
   105:           uint32_T fraction1 : 20;
E 134: bitfield size out of range - set to 1
   118:       (*(BigEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
   127:         (*(BigEndianIEEESingle*)&rtNaN).fraction = 0x7FFFFF;
W 195: constant expression out of range -- truncated
   138:         (*(BigEndianIEEEDouble*)&rtNaN).wordL.fraction1 = 0xFFFFF;
W 195: constant expression out of range -- truncated
   139:         (*(BigEndianIEEEDouble*)&rtNaN).wordH.fraction2 = 0xFFFFFFFF;
W 196: constant expression out of range due to signed/unsigned type mismatch
total errors: 4, warnings: 8
wmk: *** action exited with value 1.
wmk: *** action exited with value 1.
```

then this issue indicates that you are compiling the default Simulink Coder `rt_nonfinite.c` on a target that does not support it. The only targets which can compile the default `rt_nonfinite.c` are the TriCore and ARM platforms. Nonfinite elements on targets other than TriCore or ARM platforms are not supported with Embedded Coder software.

To resolve this issue, follow these steps:

1 Make sure you are using the stubbed out version of this file. In the configuration set, under **Code Generation** in **TLC Options**, add the following: `-aCustomNonFinites="genrtnonfinite_stub.tlc"`

2 Delete the `rt_nonfinite.c` file from the build area before attempting to rebuild the model in the same build area.

Memory Warning/Error Messages in the CrossView Pro Command Window When Using the Instruction Set Simulator

Due to a limitation in the TASKING C166 toolset you may see messages similar to the following in the CrossView Pro command window during execution of an application in the instruction set simulator:

```
GPR registers could not be scheduled to 0xF200
GPR registers could not be scheduled to 0xF220
```

and

```
Reading register "R0" (0) failed: memory failure at
memory space 0 range 0x00FC00-0x00FC01
```

These messages occur because the CrossView Pro feature "Use map file for memory map" does not work correctly.

The workaround suggested by Altium is to not use this feature, in which case the debugger assumes that the entire memory range that the processor can address is available to the application.

You can create custom Embedded Coder template projects and a custom CrossView Pro initialization file to disable this feature. For example, in the custom template application project, uncheck the project option, **CrossView Pro > Initialization > Use map file** for memory mapping.

C++ Code Generation Not Supported

C++ code generation is not supported. If you try to use this option, you see an error message like the following:

```
IDE Link does not support the Simulink Coder C++ Target
Language option. Please set the "Language" setting to
"C" in the configuration parameters of
```

```
the model.
```

There is no resolution for this issue.

Computer Vision System Toolbox Library Not Supported

The Computer Vision System Toolbox library is not supported by Embedded Coder software. If you include blocks from the Computer Vision System Toolbox library in your model then you may see compilation or link errors.

There is no resolution for this issue.

Noninlined S-functions Calling rt_matrx.c Not Supported

Noninlined S-functions that use routines in `rt_matrx.c` are not supported because `rt_matrx.c` contains functions that can allocate memory dynamically. Embedded Coder software does not support dynamic memory allocation. You may see errors like the following:

```
Linking and locating to rt_matrx_test.out
E 222: module _nmalloc.obj (_NMALLOC_C):
symbol '?C166_NHEAP_TOP': unresolved
E 222: module _nmalloc.obj (_NMALLOC_C):
symbol '?C166_NHEAP_BOTTOM':
       unresolved
total errors: 2, warnings: 0
```

There is no resolution for this issue.

"Compiler optimization level" Configuration Parameter Has No Effect

When using Embedded Coder software, the Simulink Coder Configuration Parameter **Compiler optimization level** has no effect on the building of generated code in the TASKING EDE.

The Embedded Coder template projects specify the compiler and linker settings used for building the generated code. See "Tutorial: Creating New Template Projects" on page 46-76 for more information, and "Tutorial: Creating New Template Projects" on page 46-76 for instructions on customizing settings.

Configuration Changes Cause Build Errors With Referenced Models

If you build a model hierarchy, and then change your option set or template application project before rebuilding, you see build errors like the following:

```
Simulink Configuration Parameter settings for the model
'rtwdemo_pil_link_ts' and model
'rtwdemo_pil_component_mid1_link_ts' are incompatible.
The Link or Target product settings in the configuration set
for the two models result in different build folders
for the model reference code:

rtwdemo_pil_link_ts : slprj\ert_c167cs_sim\...
rtwdemo_pil_component_mid1_link_ts : slprj\ert_c167cs_hw\...

Please check that the two models have compatible Link and
Target settings.
```

To work around this problem, change to a clean work folder after changing your target preferences and before rebuilding. Alternatively, update all models and then they will rebuild correctly with the new settings.

Processor-in-the-Loop Issues

- “Generic PIL Issues” on page 46-108
- “On-Chip PIL Not Supported on ARM Hardware” on page 46-108
- “10-Second Pause on Termination of the CrossView Pro Debugger” on page 46-108
- “DSP563xx Link-Order Issue Can Cause PIL Application Failure” on page 46-108
- “No Support for TASKING Feature “Treat double as float”” on page 46-109
- “TASKING Optimization Settings May Cause Incorrect Simulation Results” on page 46-110

Generic PIL Issues

See the Support Table section in the Embedded Coder documentation for general PIL feature support information affecting the PIL block with Link products. See "SIL and PIL Simulation Support and Limitations" on page 39-60.

On-Chip PIL Not Supported on ARM Hardware

For ARM processors the CrossView Pro instruction set simulator can be used for debugging and processor-in-the-loop (PIL) simulation, but there is currently no on-chip debugging or PIL support.

See "On-Chip Debugging/On-Chip PIL Not Supported on ARM Hardware" on page 46-98 for solutions for this issue.

10-Second Pause on Termination of the CrossView Pro Debugger

When you terminate an instance of the CrossView Pro debugger application that was launched by Embedded Coder software, there is a pause of about 10 seconds before the CrossView Pro window closes. This 10-second pause is the intended behavior of CrossView Pro when acting as a COM server. CrossView Pro pauses for the 10 seconds to wait for clients such as MATLAB to release their COM references.

DSP563xx Link-Order Issue Can Cause PIL Application Failure

When building PIL applications for DSP563xx you may see linker errors similar to the following example:

```
lk563 E208 (0): Found unresolved external(s):
            FDotProduct_s32s16            - (fuelsys0.a:fuelsys0.obj)
            FLook2D_S16_S16_S16_SAT       - (fuelsys0.a:fuelsys0.obj)
            FBINARYSEARCH_S16             - (fuelsys0.a:fuelsys0.obj)
            FINTERPOLATE_S16_S16_SAT      - (fuelsys0.a:fuelsys0.obj)
            FINTERPOLATE_EVEN_S16_S16_SAT - (fuelsys0.a:fuelsys0.obj)
wmk: *** action exited with value 1.
```

To resolve this issue, contact TASKING for a patch to make it possible to use the multipass option to rescan multiple libraries.

No Support for TASKING Feature "Treat double as float"

You can enable the feature in a TASKING project to treat the double-precision floating point datatype "double" as the single-precision floating point datatype "float". Usually, this means that double-precision floating point datatypes are represented in 4 bytes rather than 8 bytes.

PIL always assumes that the "double" datatype is represented normally. If you enable the "Treat double as float" override, PIL does not correctly transfer "double" datatypes between host and target, and unexpected data transfer errors occur during simulation. The default templates that ship with Embedded Coder software do not enable the override "Treat double as float" project option.

To resolve this issue:

- Do not use the option to treat “double” as “float”. In this case, double precision floating point values are represented normally.
- Use the “single” datatype in Simulink rather than “double”. In this case, the option to treat “double” as “float” will have no effect on PIL, because no “double” datatypes are used.

This is an example of the wider issue of problems caused by mismatching datatypes on host and target. For more details, see “Data Type Size Mismatch Issues (Embedded Targets)” on page 39-86 in the Embedded Coder documentation.

TASKING Optimization Settings May Cause Incorrect Simulation Results

Sometimes, you may observe differences between simulation and PIL simulation results. The code compiled and running in the TASKING environment may not always behave correctly, even when the generated code is correct. One cause of this issue, particularly with the TriCore toolset, is the compiler optimization configuration used to build the generated code.

If you see differences between simulation and PIL simulation results, to resolve this issue try setting the compiler optimization settings in the template projects to either `No optimization`, `Debug purpose`, or a similar equivalent for your TASKING toolset. Then, build the PIL algorithm and PIL application again and try repeating the simulation.

To create new template projects and modify their project settings see “Tutorial: Creating New Template Projects” on page 46-76.

Issues Using Simulink Coder Software Without Embedded Coder Software

- “Simulink® Coder grt.tlc-Based Targets Not Supported for PIL” on page 46-111
- “DSP563xx Toolset Support Limitations” on page 46-111
- “Use ERT Target for Memory-Constrained Targets” on page 46-111

- "8051 GRT Limitations" on page 46-112

Simulink Coder grt.tlc-Based Targets Not Supported for PIL

Simulink Coder "grt.tlc"-based targets are not supported for PIL.

To resolve this issue, use a Simulink Coder "ert.tlc"-based target.

DSP563xx Toolset Support Limitations

Only 16-bit mode for the DSP563xx Family is supported. Simulink Coder `grt.tlc`-based targets and the "GRT Compatible Call interface" option in the Simulink Coder Interface settings are not supported. This limitation is because of the nonstandard size of single- and double-precision floating-point datatypes on this architecture (`tmwtypes.h` does not compile).

You must use 16–bit mode.

Use ERT Target for Memory-Constrained Targets

Some targets such as the TASKING TriCore 1766B have memory constraints that can cause errors if you use the GRT target.

The 1766b has no external memory. You should use ERT rather than GRT when targeting this board, due to memory resource constraints. If you use the GRT target you may see compilation errors similar to the following example:

```
ltc E117: conflicting restriction for sections ".text.libc" and
".text.trapvec.000": absolute restrictions overlap
```

This problem occurs because the ERT (embedded real time) target is optimized for size and speed, while the GRT (generic real time) target is designed for ease of prototyping which incurs extra memory usage.

Use the ERT target for memory-constrained targets such as the TASKING TriCore 1766B.

See also "Linker Errors Due to Limited Memory" on page 46-99.

8051 GRT Limitations

Working with the 8051 has some limitations when using GRT.

CrossView Pro Parameters. GRT application builds link against an example main (`grt_main.c`) file which includes a main function with `argc` and `argv` parameters for handling command-line arguments. When executing the application in CrossView Pro, these parameters are uninitialized and application execution terminates early. This behavior differs from that of other toolsets, where these parameters are initialized to 0 (`argc`) and the null pointer (`argv`).

To work around this issue on 8051, you can manually set `argc` to `0` in CrossView Pro before beginning execution.

Alternatively, you can create a library project for algorithm export that does not link against `grt_main.c` — see “Setting Build Action” on page 46-18 for more detail.

DSP System Toolbox Software. The DSP System Toolbox library fails to build for GRT models with the 8051 toolset. Certain datatypes required by the DSP System Toolbox software, for example, real64_T, are not defined by Simulink Coder software for this configuration.

Use a Embedded Coder, ERT-based target, rather than a GRT-based target.

Working with Analog Devices VisualDSP++ IDE

- "Getting Started" on page 47-2
- "Automation Interface" on page 47-7
- "Project Generator" on page 47-30
- "Reported Limitations and Tips" on page 47-40

Getting Started

In this section...

"Overview" on page 47-2

"Software Structure and Components" on page 47-3

"Software Requirements" on page 47-5

"Installation and Configuration" on page 47-6

Overview

Embedded Coder software provides a connection between MATLAB and the VisualDSP++ IDE to enable you to access the processor from MATLAB. You can, manipulate data on the processor, and manage projects within the IDE, while simultaneously utilizing the MATLAB tools of numerical analysis and simulation. Using Embedded Coder software, you can perform the following tasks, and others related to Model-Based Design:

- Function calls — Write scripts in MATLAB software to execute any function in the VisualDSP++ IDE
- Automation — Write automated tests in MATLAB software to be executed on your processor, including control and verification operations
- Host-Processor Communication — Communicate with the processor directly from MATLAB software, without going to the IDE
- Verification and Validation
  - Load and execute projects into the VisualDSP++ IDE from the MATLAB command line
  - Build and compile code, and then use vectors of test data and parameters to test the code
  - Build and compile your code, and then download the code to the processor and execute it
- Design models — Design models and algorithms in MATLAB and Simulink software and run them on the processor

- Generate code— Generate executable code for your processor directly from the models designed in Simulink software, and execute it

Embedded Coder software connects MATLAB software and Simulink software with Analog Devices VisualDSP++ integrated development and debugging environment from Analog Devices. Embedded Coder software enables you to use MATLAB and Simulink software to debug and verify embedded code running on all Analog Devices DSPs that VisualDSP++ software supports, such as the Analog Devices™ Blackfin®, Analog Devices™ SHARC® and Analog Devices™ TigerSHARC® processor families.

Embedded Coder software includes a project generator component. With the project generator component, you can generate a complete project for the VisualDSP++ IDE from your Simulink software models, including ANSI C code generated with Simulink Coder software. Thus, you use the Simulink Coder and Embedded Coder software to generate generic ANSI C code projects for VisualDSP++ software from models. You can then build and run these projects on Blackfin, SHARC®, and TigerSHARC® processors.

The following list suggests some of the uses for the capabilities of the software:

- Create test benches in MATLAB and Simulink software for testing your manually written or automatically generated code running on ADI DSPs
- Generate code and project files for VisualDSP++ software from Simulink models for rapid prototyping or deployment of a system or application
- Build, debug, and verify embedded code on ADI DSPs
- Perform processor-in-the-loop (PIL) testing of embedded code

Software Structure and Components

- “Automation Interface” on page 47-4
- “Project Generator” on page 47-4
- “Verification” on page 47-5

Embedded Coder software comprises components—the Automation Interface component, the Project Generation component, and the Verification component. The Automation Interface component enables communication

between MATLAB software and Embedded Coder software. The Project Generation component leverages Simulink software and lets you build models, simulate them, and generate code from the models directly to the processor.

The Verification component offers capabilities that help you use Model-Based Design to validate and verify your projects. With the Verification component, you can simulate algorithms and processes in Simulink models and concurrently on your processor. Comparing the results helps verify the fidelity of you model or algorithm code.

Automation Interface

The Automation Interface component allows you to use Embedded Coder functions and methods to communicate with the VisualDSP++ IDE to perform the following tasks:

- Automate project management
- Debug programs
- Manipulate the data in the processor internal and external memory, and in the registers
- Communicate between the host and processor applications

The Debug Component of automation interface includes methods and functions for project automation, debugging, and data manipulation.

Project Generator

The Project Generator component comprises methods that utilize the VisualDSP++ API to create projects in VisualDSP++ software and generate code with Simulink Coder and Embedded Coder software. With the interface, you can do the following:

- Automatic project-based build process — Automatically create and build projects for code generated by Simulink Coder or Embedded Coder software.
- Custom code generation — Use System Target Files (STF) to generate processor-specific and optimized code.

- Automatic downloading and debugging — Debug generated code in the VisualDSP++ debugger, using either the instruction set simulator or real hardware.
- Create and build projects for VisualDSP++ software from Simulink models — Project Generator uses Simulink Coder or Embedded Coder software to build projects that work with Analog Devices processors.
- Generate custom code using the Configuration Parameters in your model with the system target files `vdsplink_ert.tlc` and `vdsplink_grt.tlc`.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded Coder software combine to provide the following verification tools for you to apply as you develop your code:

Processor-in-the-Loop Simulation. Use simulation techniques to verify generated code running in an instruction set simulator or real hardware environment.

Task Execution and Stack Usage Profiling. Gather execution profiling measurements with VisualDSP++ instruction set simulator to establish the timing requirements of your algorithm. Also, verify the stack usage is appropriate and as expected.

Software Requirements

For detailed information about the software and hardware required to use Embedded Coder software, refer to the Embedded Coder system requirements areas on the MathWorks Web site:

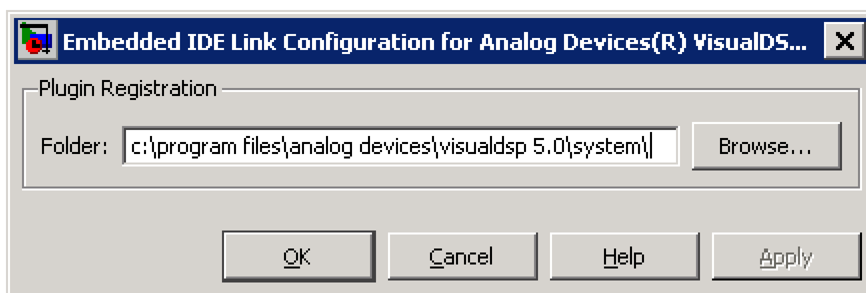
- Requirements for Embedded Coder: www.mathworks.com/products/ide-link/requirements.html
- Requirements for use with VisualDSP++: www.mathworks.com/products/ide-link/adi-adaptor.html

Installation and Configuration

1. Install VisualDSP++ according to the instructions provided with that software.

2. Enter `adivdspsetup` on the MATLAB command line.

3. Use **Browse** to locate the system folder for Analog Devices VisualDSP++. This action registers the Embedded Coder with that IDE.



4. Confirm that the installation works by entering `IDE_Obj = adivdsp` on the MATLAB command line. This action creates an IDE handle object for VisualDSP++ in MATLAB, and starts VisualDSP++.

Automation Interface

In this section...
"Getting Started with Automation Interface" on page 47-7
"Constructing Objects" on page 47-22
"Properties and Property Values" on page 47-23
"adivdsp Object Properties" on page 47-27

Getting Started with Automation Interface

- "Introducing the Automation Interface Tutorial" on page 47-7
- "Running the Interactive Tutorial" on page 47-10
- "Selecting Your Session and Processor" on page 47-11
- "Querying Objects for VisualDSP++ IDE" on page 47-12
- "Loading Files into VisualDSP++ IDE" on page 47-14
- "Running the Project" on page 47-15
- "Working with Global Variables and Memory" on page 47-16
- "Working with Local Variables and Memory" on page 47-18
- "Closing Files and Projects" on page 47-20
- "Closing the Connections or Cleaning Up VisualDSP++ Software" on page 47-21
- "Tutorial Summary" on page 47-22

Introducing the Automation Interface Tutorial

Embedded Coder software provides a connection between MATLAB software and a processor in VisualDSP++ software. You can use objects as a mechanism to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you while you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Note Before using the functions available with the objects, you must select a session in the VisualDSP++ IDE. The object you create is specific to a designated session in VisualDSP++ IDE.

To get you started using objects for VisualDSP++ software, Embedded Coder software includes an example script `vdspautointtutorial.m`. As you work through this tutorial, you perform the following tasks that step you through creating and using objects for VisualDSP++ IDE.

1. Select your session.
2. Create and query objects to VisualDSP++ IDE.
3. Use MATLAB software to load files into VisualDSP++ software IDE.
4. Work with your VisualDSP++ IDE project from MATLAB software.
5. Close the connections you opened to VisualDSP++ IDE.

You use these tasks in any development work you do with signal processing applications. Thus, the tutorial provided here gives you a working process and best practice for using Embedded Coder software and your signal processing programs to develop programs for a range of Analog Devices processors.

The tutorial covers some methods and functions for Embedded Coder software. The functions listed first do not require an `adivdsp` object. The methods listed require an existing `adivdsp` object before you can use the function syntax.

Functions for Working with VisualDSP++ Software. The following table shows functions that do not require an object.

Function	Description
`listsessions`	Return information about the boards that VisualDSP++ IDE recognizes as installed on your PC.
`adivdsp`	Construct an object that refers to a VisualDSP++ IDE session. When you construct the object you specify the session by processor.

Methods for Working with adivdsp Objects in VisualDSP++ Software. The following table presents some of the methods that require an `adivdsp` object.

Methods	Description
`add`	Add a file to a project
`address`	Return the address and page for an entry in the symbol table in VisualDSP++ IDE
`build`	Build the project in VisualDSP++ software
`cd`	Change the working folder
`display`	Display the properties of an object that references a VisualDSP++ software session
`halt`	Terminate execution of a process running on the processor
`info`	Return information about the object or session
`isrunning`	Test whether the processor is executing a process
`load`	Load a built project to the processor
`open`	Open a file in the project
`read`	Retrieve data from memory on the processor
`reset`	Restore the program counter (PC) to the entry point for the current program
`run`	Execute the program loaded on the processor
`save`	Save files or projects
`visible`	Set whether VisualDSP++ IDE window is visible on the desktop while VisualDSP++ IDE is running
`write`	Write data to memory on the processor

Running VisualDSP++ Software on Your Desktop — Visibility. When you create an `adivdsp` object in the tutorial in the next section, Embedded Coder starts VisualDSP++ software in the background.

If VisualDSP++ software is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `idde.exe`, on the **Processes** tab in Task Manager.

You can make the VisualDSP++ IDE visible with the function `visible`. The function `isvisible` returns the status of the IDE—is it visible on your desktop. To close the IDE when it is not visible and MATLAB is not running, use the **Processes** tab in WindowsWindows Task Manager and look for `idde.exe`.

If an object that refers to VisualDSP++ software exists when you close VisualDSP++ software, the application does not close. Windows software moves it to the background (it becomes invisible). Only after you clear all objects that access VisualDSP++ IDE, or close MATLAB, does closing VisualDSP++ unload the application. You can see if VisualDSP++ IDE is running in the background by checking in the Windows Task Manager. When VisualDSP++ IDE is running, the entry `idde.exe` appears in the **Image Name** list on the **Processes** tab.

Running the Interactive Tutorial

You have the option of running this tutorial from the MATLAB command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB, click `run vdspautointtutorial`. This command launches the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next section. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial MATLAB file used here by clicking `vdspautointtutorial.m`.

Note To run the interactive tutorial, you must have at least one session configured in VisualDSP++ software. If you do not yet have a session, use the Analog Devices VisualDSP++ Configurator to create a session to use for this tutorial.

Selecting Your Session and Processor

Embedded Coder IDE requires that you have at least one session available for VisualDSP++ software. To help you select the session to use for this tutorial, and for any development work, Embedded Coder software provides a command line tool, called `listsessions`, which prints a list of the available sessions. So that you can use this function in a script, `listsessions` can return a MATLAB structure that you use when you want your script to select a session in the IDE without your help.

Note The session you select is used throughout the tutorial.

---

1. To see a list of the sessions that you can use, enter the following command at the MATLAB prompt:

   ```
   session_list  =  listsessions
   ```

   MATLAB returns a list that shows all the sessions that Embedded Coder IDE recognizes as available in your installation.

   ```
   session_list =

       'ADSP-21060 ADSP-2106x Simulator'
       'ADSP-21362 ADSP-2136x Simulator'
   ```

2. `listsessions` has a verbose mode that provides further details about the sessions in a cell array. The array contains structures that describe each session—the target type, the platform, and the processor.

   ```
   sessionsinfo = listsessions('verbose');

       echo off
        sessionname: 'ADSP-21362 ADSP-2136x Simulator'
         targettype: 'ADSP-2136x Family Simulator'
       platformname: 'ADSP-2136x Simulator'
         processors: 'ADSP-21362'
   ```

3. Use `adivdsp` to create an object that accesses a session in VisualDSP++ IDE.

```
IDE_Obj = adivdsp('sessionname','ADSP-21362 ADSP-2136x Simulator','procnum',0)
```

Sessionname and procnum are property names that specify the property to set. ADSP-21362 ADSP-2136x Simulator is the session to access, and 0 is the number of the processor to refer to in the session.

When you use adivdsp, you create an object, in this case IDE_Obj, that refers to the session you specify in sessionname.

Querying Objects for VisualDSP++ IDE

In this tutorial section you create the connection between MATLAB and VisualDSP++ IDE. This connection, or object, is a MATLAB object, which for this session you save as variable IDE_Obj. You use function adivdsp to create objects. When you create objects, adivdsp input arguments let you define other object properties, such as the global time-out. Refer to the adivdsp reference information for more about the input arguments.

Use the generated object IDE_Obj to direct actions to your session processor. In the following tasks, IDE_Obj appears in all function syntax that interact with IDE session and the processor: The object IDE_Obj identifies and refers to a specific session. You need to include the object in any method syntax you use to access and manipulate a project or files in a session in VisualDSP++ IDE.

1 Create an object that refers to your selected session and processor. Enter the following command at the prompt.

   ```
   IDE_Obj = adivdsp('sessionname','ADSP-21362 ADSP-2136x Simulator','procnum',0)
   ```

   If you watch closely, and your machine is not too fast, you see VisualDSP++ software appear briefly when you call adivdsp. If VisualDSP++ was not running before you created the new object, VisualDSP++ software starts and runs in the background.

   Usually, you need to interact with VisualDSP++ while you develop your application. The function visible, controls the state of VisualDSP++ software on your desktop. visible accepts Boolean inputs that make VisualDSP++ software either visible on your desktop (input to visible ≥ 1) or invisible on your desktop (input to visible = 0). For this tutorial, you

need to interact with the development environment, so use `visible` to set the IDE visibility to `1`.

2 To make VisualDSP++ IDE show on your desktop, enter the following command at the prompt:

```
visible(IDE_Obj,1)
```

3 Next, enter `display(IDE_Obj)` at the prompt to see the status information.

```
ADIVDSP Object:
  Session name     : ADSP-21362 ADSP-2136x Simulator
  Processor name   : ADSP-21362
  Processor type   : ADSP-21362
  Processor number : 0
  Default timeout  : 10.00 secs
```

Embedded Coder software provides three methods to read the status of a processor:

- `info` — Return a structure of testable session conditions.
- `display` — Print information about the session and processor.
- `isrunning` — Return the state (running or halted) of the processor.

4 Type `procinfo = info(IDE_Obj)`.

The `IDE_Obj` link status information provides data about the hardware, as follows:

```
procinfo =

    procname: 'ADSP-21362'
    proctype: 'ADSP-21362'
    revision: ''
```

5 Verify that the processor is running by entering

```
runstatus = isrunning(IDE_Obj)
```

MATLAB responds, indicating that the processor is stopped, as follows:

```
runstatus =

     0
```

Loading Files into VisualDSP++ IDE

In this part of the tutorial, you load the executable code for the CPU in the IDE. Embedded Coder software includes a tutorial project file for VisualDSP++ IDE. Through the next commands in the tutorial, you locate the tutorial project file and load it into VisualDSP++ IDE. The `open` method directs VisualDSP++ software to load a project file or workspace file.

Note To continue the tutorial, you must identify or create a folder to which you have write access. Embedded Coder software cannot create a folder for you. If you do not have a writable folder, create one in Windows software before you proceed with the rest of this tutorial.

VisualDSP++ software has its own workspace and workspace files that are quite different from MATLAB workspace files and the MATLAB workspace. Remember to monitor both workspaces. The next steps change the working folder to your new writable folder.

1. Use `cd` to switch to the writable folder

   ```
   prj_dir = cd('C:\vdsp_demo')
   ```

   where the name and path to the writable folder is a string, such as `C:\vdsp_demo` as used in the example. Replace `C:\vdsp_demo` with the full path to your folder.

2. Change your working folder to the new folder by entering the following command:

   ```
   cd(IDE_Obj,prj_dir)
   ```

3. Next, use the following command to create a new VisualDSP++ software project named `dot_product_c.dpj` in the new folder:

```
new(IDE_Obj,'debug_demo.dpj')
```

Look in the IDE to verify that your new project exists. Next you need to add source files to your project.

4 Add the provided source file—`scalarprod.c` to the project `debug_demo.dpj` using the following command:

```
add(IDE_Obj, [matlabroot '\toolbox\vdsplink\vdspdemos\src\scalarprod.c'])
```

The variable *`matlabroot`* indicates the root folder of your MATLAB installation. Replace *`matlabroot`* with the path to MATLAB on your machine. For more information about the MATLAB root folder, refer to `matlabroot` in the MATLAB documentation.

5 Open the file in the IDE from MATLAB by issuing the following command to open the file:

```
open(IDE_Obj,[matlabroot '\toolbox\vdsplink\vdspdemos\src\scalarprod.c'])
```

Switch to the IDE to verify that the files are in your project and open.

6 Save your project.

```
save(IDE_Obj,'debug_demo.dpj','project')
```

Your IDE project is saved with the name `debug_demo.dpj` in your writable folder. The input string '`project`' specifies that you are saving a project file.

Running the Project

After you create `dot_project_c.dpj` in the IDE, you can use Embedded Coder functions to create executable code from the project and load the code to the processor.

The next steps in this tutorial build the executable and download and run it on your processor.

1 Use the following build command to build an executable module from the project `dot_product_c.dpj`.

```
build(IDE_Obj,30) % The optional input argument 30 sets the time out period to 30 seconds.
```

At the end of the build process, Embedded Coder software returns a value of 1 to indicate that the build succeeded. If the build process returns a 0, the build failed.

```
ans  =

       1
```

2 To load the new executable to the processor, use load with the project file name and the object name. The name of the executable is debug_demo.dxe, and it is stored with the project in your writable folder, in a subfolder named debug.

```
load(IDE_Obj,'c:\vdsp_demo\debug\debug_demo.dxe',30);
```

Embedded Coder software provides methods to control processor execution—run, halt, and reset. To demonstrate these methods, use run to start the program you loaded on the processor, and then use halt to stop the processor.

Try the following methods at the command prompt.

```
run(IDE_Obj)      % Start the program running on the processor.
halt(IDE_Obj)     % Halt the processor.
reset(IDE_Obj)    % Reset the program counter to start of program.
```

Working with Global Variables and Memory

After you load your program on the processor, you can access memory locations and variables. You can then read variables either from the program symbol table or directly from addresses in memory. Three methods—address, read, and write, let you get, read, and write to and from your project and processor.

Start by getting the address of the global variable v1 from the debug_demo project symbol table.

1 Enter the following command to retrieve the address for v1.

```
address_v1 = address(IDE_Obj, 'v1')
```

```
address_v1 =

       753666            1
```

2 Convert the address from decimal format to hexadecimal.

```
dec2hex(address_v1(1))

ans =

B8002
```

The address of global data array `v1` is 0xB8002, which is stored in type 1 memory on the processor

3 With the address of `v1` saved as `address_v1`, use `read` to return the data from that location. To specify the data type and the number of values to read, add the `datatype` ('`int32`') and count (`32`) input arguments.

```
value_v1 = read(IDE_Obj, address_v1, 'int32', 32) % Interpret the data as 32-bit integers.

value_v1 =

  Columns 1 through 10

     -37    -133      31    -104      32      66    -123      19     140     -28

  Columns 11 through 20

      16      80      -2      83    -243     148      56     163      46      45

  Columns 21 through 30

    -217     -11    -164      49      -3      99      21     -61     -26     101

  Columns 31 through 32

    -101    -151
```

4 Repeat the read process for another global variable in the project—`v2`. Nest the `address` method inside the `read` method to reduce typing.

```
value_v2 = read(IDE_Obj,address(IDE_Obj,'v2'),'int32',32) % Read and address methods in one call.

value_v2  =

  Columns 1 through 10

   -50      5    -17     28      5     31    -23   -156     68     -5

  Columns 11 through 20

  -220      5    -14     57    214    183    213     40    175    144

  Columns 21 through 30

   -12    -77    -18     77    130    -39    132    107     52    -59

  Columns 31 through 32

   127   -117
```

Working with Local Variables and Memory

If you examine the source files for `debug_demo` in the IDE, you can verify the values for `v1` and `v2` in the source file `scalarprod.c`. You can also use the `address` method to get the addresses of local variables on the stack, after the variable is in scope.

To get the variables in scope (on the stack), you run the program. Adding a breakpoint to the program allows you to read the stack contents when the program stops at the breakpoint. Without the breakpoint, the program runs to completion, and you cannot read the contents of the stack because it no longer exists.

Begin the process by adding a breakpoint to the project file `scalarprod.c`:

1. Insert a breakpoint on line 100 of program `scalarprod.c` with the following command:

   ```
   insert(IDE_Obj, 'scalarprod.c', 100)
   ```

2 Run the program to add the variable to the stack, and move the program counter to the breakpoint. Add the optional input argument `timeout` sets the time out value to 30s instead of the default 20s value:

```
run(IDE_Obj,'runtohalt',30)
```

The program stops at the breakpoint on line 100.

3 Read the address of the local variable `result`, and convert it to its hexadecimal equivalent value.

```
address_result = address(IDE_Obj,'result','local') % address_result is a 'local' variable.

address_result  =

     933884          1

dec2hex(address_result(1))

ans  =

E3FFC
```

`address` returns `933884` as the location of result in memory, in type 1 memory on the processor, stored in the MATLAB variable `address_result`.

4 Use the variable `address_result` to get the value stored at that address by issuing the following `read` command:

```
actual_value_result = read(IDE_Obj, address_result, 'int32')

actual_value_result  =

         18875
```

Verify in the IDE Output Window that 18875 is the correct value for the dot product.

5 Use the following command to remove the breakpoint set on line 100.

```
remove(IDE_Obj, 'scalarprod.c', 100)
```

MATLAB includes a dot product function to use to verify the value in `actual_value_result`. Called `dot`, the function calculates the dot product of two input vectors. In this case, the inputs are vectors `value_v1` and `value_v2`.

Comparing the two results—`expected_value_result` in MATLAB with `actual_value_result` from the processor implementation validates your simulation and implementation. With Automation Interface methods, you can create MATLAB file scripts to test and verify algorithms in their implementation on a processor.

1. Calculate the expected result by performing the dot function with two input vectors.

   ```
   expected_value_result = dot(value_v1, value_v2)

   expected_value_result  =

          18875
   ```

2. Test to see if the actual and expected results match.

   ```
   isequal(expected_value_result, actual_value_result)

   ans  =

        1
   ```

3. After verifying the result and removing the breakpoint, run the program to completion, and then halt and reset the processor.

   ```
   run(IDE_Obj)
   halt(IDE_Obj)
   reset(IDE_Obj)
   ```

Closing Files and Projects

You can close files in your projects from the MATLAB command line. The method `close` works at the command line to close programs or projects in the IDE through the `adivdsp` object and input keywords that describe the kind of file to close.

To finish this tutorial, close the open documents or files in the IDE, and then close the project `debug_demo.dpj`.

1 Close all of the open files and documents in the IDE. All of the open files are text files, so use the `text` input argument.

```
close(IDE_Obj, 'all', 'text')
```

2 Now, close the project.

```
close(IDE_Obj, 'debug_demo.dpj', 'project')
```

Note If you close the VisualDSP++ IDE manually outside of MATLAB, clear the IDE handle object in MATLAB. For example, at the MATLAB command line enter:

```
clear IDE_Obj
```

Closing the Connections or Cleaning Up VisualDSP++ Software

Objects that you create in Embedded Coder software have connections to VisualDSP++ software. Until you delete these handles, the VisualDSP++ process (`idde.exe` in the Windows Task Manager) remains in memory. Closing MATLAB removes these objects automatically, but there may be times when it helps to delete the handles manually, without quitting MATLAB.

Note When you clear the last `adivdsp` IDE handle object, Embedded Coder software closes VisualDSP++ software. When it closes the IDE, the link software does not save current projects or files in the IDE, and it does not prompt you to save them. A best practice is to save all of your projects and files before you clear `adivdsp` objects from your MATLAB workspace.

---

1 Use the following command to make the IDE invisible if it is visible on your desktop.

```
visible(IDE_Obj,0)
```

2. To delete your connection to VisualDSP++ IDE, use `clear IDE_Obj`.

Tutorial Summary

During the tutorial you performed the following tasks:

1. Selected your session.
2. Created and queried objects that refer to a session in Embedded Coder to get information about the session and processor.
3. Used MATLAB to load files into VisualDSP++ IDE, and used methods in MATLAB to run that file.
4. Accessed variables in the program symbol table and on the processor.
5. Used the Automation Interface methods to compare the results of a simulation in MATLAB with the same algorithm running on a processor.
6. Closed the files, projects, and connections you opened to VisualDSP++ IDE.

Constructing Objects

When you create a connection to a session in VisualDSP++ software using the `adivdsp` function, you create an object. The object implementation relies on MATLAB object-oriented programming capabilities similar to the objects you find in MATLAB or Filter Design Toolbox.

The discussions in this section apply to the objects in Embedded Coder software. Because `adivdsp` objects use the MATLAB programming techniques, the information about working with the objects, such as how you get or set properties, or use methods, apply to the objects you create in Embedded Coder software.

Like other MATLAB structures, objects in Embedded Coder software have predefined fields referred to as *object properties*.

You specify object property values by one of the following methods:

- Specifying the property values when you create the object

- Creating an object with default property values, and changing some or all of these property values later

For examples of setting link properties, refer to "Setting Property Values with set."

Example — Constructor for adivdsp Objects

The easiest way to create an object is to use the function `adivdsp` to create an object with the default properties. Create an object named `IDE_Obj` referring to a session in VisualDSP++ software by entering the following syntax:

```
IDE_Obj = adivdsp
```

MATLAB responds with a list of the properties of the object `IDE_Obj` you created along with the associated default property values.

```
ADIVDSP Object:
  Session name     : ADSP-21362 ADSP-2136x Simulator
  Processor name   : ADSP-21362
  Processor type   : ADSP-21362
  Processor number : 0
  Default timeout  : 10.00 secs
```

The object properties are described in the `adivdsp` documentation.

Note These properties are set to default values when you construct links.

Properties and Property Values

- "Setting and Retrieving Property Values" on page 47-24
- "Setting Property Values Directly at Construction" on page 47-24
- "Setting Property Values with set" on page 47-25
- "Retrieving Properties with get" on page 47-25
- "Direct Property Referencing to Set and Get Values" on page 47-26
- "Overloaded Functions for adivdsp Objects" on page 47-26

Objects in this software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. Also, a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

Setting and Retrieving Property Values

You can set `adivdsp` object property values by either of the following methods:

- Directly when you create the link — see "Setting Property Values Directly at Construction"
- By using the `set` function with an existing link — see "Setting Property Values with set"

Retrieve Embedded Coder software object property values with the `get` function.

Direct property referencing lets you either set or retrieve property values for `adivdsp` objects.

Setting Property Values Directly at Construction

To set property values directly when you construct an object, include the following entries in the input argument list for the constructor method `adivdsp`:

- A string for the property name to set followed by a comma. Enclose the string in single quotation marks as you do any string in MATLAB.
- The associated property value. Sometimes this value is also a string.

Include as many property names in the argument list for the object construction command as there are properties to set directly.

Example — Setting Object Property Values at Construction. Suppose that you want to create a link to a session in VisualDSP++ software and set the following object properties:

- Refer to the specified session.
- Connect to the first processor.
- Set the global time-out to 5 s. The default is 10 s.

Set these properties by entering

```
IDE_Obj = adivdsp('sessionname','ADSP-21060 ADSP-2106x Simulator','procnum',0,'timeout',5);
```

The `sessionname`, `procnum`, and `timeout` properties are described in Link Properties, as are the other properties for links.

Setting Property Values with set

After you construct an object, the `set` function lets you modify its property values.

Using the `set` function, you can change the value of any writable property of an object.

Example — Setting Object Property Values Using set. To set the time-out specification for the link `IDE_Obj` from the previous section, enter the following syntax:

```
set(IDE_Obj,'timeout',8);

get(IDE_Obj,'timeout');
ans =

     8
```

The display reflects the changes in the property values.

Retrieving Properties with get

You can use the `get` command to retrieve the value of an object property.

Example — Retrieving Object Property Values Using get. To retrieve the value of the `sessionname` property for `vd2`, and assign it to a variable, enter the following syntax:

```
session = get(vd2,'sessionname')

session =

ADSP-21060 ADSP-2106x Simulator
```

Direct Property Referencing to Set and Get Values

You can directly set or get property values using MATLAB structure-like referencing. Do this by using a period to access an object property by name, as shown in the following example.

Example — Direct Property Referencing in Links. To reference an object property value directly, perform the following steps:

1. Create a link with default values.

2. Change its time-out and number of open channels.

```
IDE_Obj = adivdsp;
IDE_Obj.time = 6;
```

Overloaded Functions for adivdsp Objects

Several methods and functions in Embedded Coder software have the same name as functions in other MathWorks products. These functions behave similarly to their original counterparts, but you apply them to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for objects. After you specify your object by assigning values to its properties, you can apply the methods in this toolbox (such as `address` for reading an address in memory) directly to the variable name you assign to your object. You do not have to specify your object parameters again.

For a complete list of the methods that act on `adivdsp` objects, refer to Analog Devices VisualDSP++ in the function reference pages.

adivdsp Object Properties

- "Quick Reference to adivdsp Properties" on page 47-27
- "Details About adivdsp Object Properties" on page 47-28

Embedded Coder software provides links to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Because Embedded Coder software uses objects to create the links, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the objects for VisualDSP++ software. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB users may find much of this handling of objects familiar. Objects in Embedded Coder software behave like objects in MATLAB and the other object-oriented toolbox products. C++ programmers may already understand the concepts described in this section.

Quick Reference to adivdsp Properties

The following table lists the properties for the links in Embedded Coder software. The second column indicates the object to which the property belongs. Knowing which property belongs to each object tells you how to access the property.

Property Name	User Settable?	Description
`sessionname`	At construction only	Reports the name of the session in VisualDSP++ IDE that the object references.
`procnum`	At construction only	Stores the number of the processor in the session. If you have more than one processor, this number identifies the specific processor.
`timeout`	Yes/default	Contains the global time-out setting for the link.

Some properties are read only. Thus, you cannot set the property value. Other properties you can change at any time. If the entry in the User Settable column is "At construction only", you can set the property value only when you create the object. Thereafter it is read only.

Details About adivdsp Object Properties

To use the objects for VisualDSP++ interface, set values for the following:

- `sessionname` — Specify the session with which the object interacts.
- `procnum` — Specify the processor in the session. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — Specify the global time-out value. (Optional. Default is 10 s.)

Details of the properties associated with `adivdsp` objects appear in the following sections, listed in alphabetical order by property name.

procnum. Property `procnum` identifies the processor referenced by an object for Embedded Coder IDE. Use `procnum` to specify the processor you are working with in the session specified by `sessionname`. The VisualDSP++ Configurator assigns a number to each processor installed in each session. To determine the value of `procnum` for a processor, use `listsessions` or the Configurator.

To identify a processor, you need the `sessionname` and `procnum` values. For sessions with one processor, `procnum` equals 0. VisualDSP++ IDE numbers the processors on multiprocessor boards sequentially from 0 to the total number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

sessionname. Property `sessionname` identifies the session referenced by a Embedded Coder software. When you create an object, you use `sessionname` to specify the session you are intending to interact with. To get the value for `sessionname`, use `listsessions` or the Analog Devices VisualDSP++ Configurator. The Configurator utility assigns the name for each session available on your system.

timeout. Property `timeout` specifies how long VisualDSP++ software waits for any process to finish. You set the global time-out when you create an object for a session in VisualDSP++ IDE. The default global time-out value 10 s. The following example shows the `timeout` value for object `vd2`.

```
display(vd2)

ADIVDSP Object:
  Session name     : ADSP-21060 ADSP-2106x Simulator
  Processor name   : ADSP-21060
  Processor type   : ADSP-21060
  Processor number : 0
  Default timeout  : 10.00 secs
```

Project Generator

In this section...

"Introducing Project Generator" on page 47-30

"Project Generator Tutorial" on page 47-31

"Model Reference" on page 47-35

Introducing Project Generator

Project generator provides the following features for developing projects and generating code:

- Automated project building for VisualDSP++ software that lets you create VisualDSP++ software projects from code generated by Simulink Coder and Embedded Coder software. Project generator populates projects in the VisualDSP++ software development environment.
- Blocks in the library `idelinklib_adivdsp` for controlling the scheduling and timing in generated code.
- Highly configurable code generation using model configuration parameters and Target Preferences block options.
- Capability to use one of two system target files to generate code specific to your processor.
- Highly configurable project build process.
- Automatic downloading and running of your generated projects on your processor.

To configure your Simulink software models to use the Project Generator component, do one or both of the following tasks:

- Add a Target Preferences block from the `idelinklib_common` library to the model.
- To use the asynchronous scheduler capability in Embedded Coder software, add one or more hardware interrupt blocks or idle task block from the `idelinklib_adivdsp` library.

The following sections describe the blockset and the blocks in it, the scheduler, and the Project Generator component.

Project Generator Tutorial

- "Building the Model" on page 47-32
- "Adding the Target Preferences Block to Your Model" on page 47-32
- "Specifying Simulink Configuration Parameters for Your Model" on page 47-32

In this tutorial you build a model and generate a project from the model into VisualDSP++ IDE.

Note The model demonstrates project generation only. You cannot build and run the model on your processor without additional blocks.

To generate a project from a model, complete the following tasks:

1. Use Simulink blocks, DSP System Toolbox blocks, and blocks from other blocksets to create the model application.
2. Add the Target Preferences block from the `idelinklib_common` library to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.
3. Set the configuration parameters for your model, including the following parameters:
   - Solver parameters such as simulation start and solver options
   - Simulink Coder software options such as processor configuration and processor compiler selection
4. Generate your project.
5. Review your project in VisualDSP++ software.

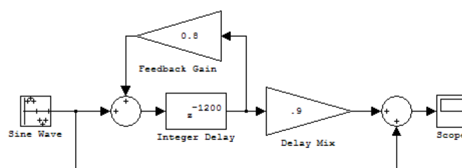
Building the Model

To build the model for audio reverberation, follow these steps:

1. Start Simulink software.

2. Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.

3. Use Simulink blocks and DSP System Toolbox blocks to create the following model.



   Look for the Integer Delay block in the Discrete library of Simulink and the Gain block in the Commonly Used Blocks library. Do not add the Target Preferences block at this time.

4. Save your model with a suitable name before continuing.

Adding the Target Preferences Block to Your Model

To configure your model to work with Analog Devices processors, add a Target Preferences block to your model, as described in "Target Preferences" on page 43-4.

You have completed the model. Next, configure the model configuration parameters to generate a project in VisualDSP++ IDE from your model.

Specifying Simulink Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink software.

Setting Solver Options. After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

1 Open the Configuration Parameters dialog box and set the appropriate options on the `Solver` category for your model and for Embedded Coder software.

   - Set **Start time** to `0.0` and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.
   - Under **Solver options**, select the `fixed-step` and `discrete` settings from the lists when you generate executable projects. When you use PIL, use any setting on the **Type** and **Solver** lists.
   - Set the **Fixed step size** to `Auto` and the **Tasking Mode** to `Single Tasking`.

Note Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

Ignore the `Data Import/Export`, `Diagnostics`, and `Optimization` categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Simulink Coder Software Options. To configure Simulink Coder software to use the correct processor files and to compile and run your model executable file, you set the options in the `Code Generation` category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the Simulink Coder software options to generate code tailored for your DSP:

1 Select `Code Generation` on the **Select** tree.

2 In `Target selection`, click **Browse** to select the system target file for Analog Devices processors—`vdsplink_grt.tlc`. It may already be the selected target file.

Clicking **Browse** opens the **System Target File Browser** to allow you to changes the system target file.

3 On the **System Target File Browser**, select the system target file `vdsplink_grt.tlc`, and click **OK** to close the browser.

Setting Embedded Coder Options. After you set the Simulink Coder options for code generation, set the options that apply to your Analog Devices processor.

1 Change the category on the **Select** tree to `Hardware Implementation`.

2 Verify that the Device type is the correct value for your processor—ADI Blackfin, ADI SHARC, or ADI TigerSHARC.

3 From the **Select** tree, choose `IDE Link` to specify code generation options that apply to the processor.

4 Under **Code Generation**, clear all of the options.

5 (optional) Under **Link Automation**, provide a name for the handle in **IDE handle name**.

6 Set the following options in the dialog box under **Project options**:

- Set **Project options** to `Custom`.
- Set **Compiler options string** and **Linker options string** to blank.

7 Set the following **Runtime** options:

- **Build action**: `Create_project`.
- **Interrupt overrun notification method**: `Print_message`.

You have configured the Simulink Coder options that let you generate a project for your processor. A few Simulink Coder categories on the **Select** tree, such as `Comments`, `Symbols`, and `Optimization` do not require configuration for use with Embedded Coder software. In some cases, you may decide to set options in the other categories.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options

in these categories to provide information during the build and to run TLC debugging when you generate code. Refer to your Simulink and Simulink Coder documentation for more information about setting the configuration parameters.

Creating Your Project. After you set the configuration parameters and configure Simulink Coder software to create the files you need, you direct the software to create your project:

1. Click **OK** to close the Configuration Parameters dialog box.

2. Click **Incremental Build** on the model toolbar to generate your project into VisualDSP++ IDE.

   When you perform an incremental build with **Build action** set to `Create_project`, the automatic build process starts VisualDSP++ software and populates a new project in the development environment.

Model Reference

- "How Model Reference Works" on page 47-36
- "Using Model Reference" on page 47-37
- "Configuring Targets to Use Model Reference" on page 47-38

Model reference lets your model include other models as modular components. This technique is useful because it provides the following capabilities:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and then only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Simulink Coder documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- The *Top model* is the root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- *Referenced models* are blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Simulink Coder documentation in the online Help system.

Model Reference in Simulation. When you simulate the top model, Simulink Coder software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (`.mex` file) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these setting through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation. Simulink Coder software requires executables to generate code from models. If you have not simulated your model at least once, Simulink Coder software creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, the software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded Coder software provides full support for generating code from models that use model reference.

Build Action Setting. The most important requirement for using model reference with the Analog Devices targets is that you must set the **Build action** (select **Configuration Parameters > IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action, perform the following steps:

1. Open your model.
2. Select **Simulation > Configuration Parameters** from the model menus.

   The Configuration Parameters dialog box opens.
3. From the **Select** tree, choose **IDE Link**.
4. In the right pane, under **Runtime**, select set `Archive_library` from the **Build action** list.

If your top model uses a reference model that does not have the build action set to `Archive_library`, the build process automatically changes the build action to `Archive_library` and issues a warning about the change.

Selecting the `Archive_library` setting removes the following options from the dialog box:

- **Interrupt overrun notification method**
- **Compiler options string**
- **Linker options string**
- **System stack size (MAUs)**
- **Profile real-time execution**

Target Preferences Blocks in Reference Models. Each referenced model and the top model must include a Target Preferences block for the correct processor. You must configure all the Target Preferences blocks for the same processor.

The referenced models need Target Preferences blocks to provide information about which compiler and which archiver to use. Without these blocks, the compile and archive processes do not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations. Model reference with Embedded Coder software does not allow you to use the following blocks or S-functions in reference models:

- No noninlined S-functions
- None of the following blocks:
  - Target Preferences
  - Memory Allocate
  - Memory Copy
  - Idle Task
  - Hardware Interrupt for SHARC, TigerSHARC, or Blackfin DSPs

Configuring Targets to Use Model Reference

When you create models to use in Model Referencing, keep in mind the following considerations:

- Your model must use a system target file derived from the ERT or GRT targets files.
- When you generate code from a model that references other models, you must configure the top-level model and the referenced models for the same system target file.

- Simulink Coder software builds and Embedded Coder software do not support external mode in model reference. If you select the external mode option, it is ignored during code generation.
- Your TMF must support use of the shared utilities folder, as described in Supporting Shared Utility Directories in the Build Process in the Simulink Coder documentation.

To use an existing processor, or a new processor, with Model Reference, set the `ModelReferenceCompliant` flag for the processor. For information about setting this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to MATLAB release R14SP3, use the following command to set the `ModelReferenceCompliant` flag to `On` to make your model compatible with model reference:

```
set_param(bdroot,'ModelReferenceCompliant','on')
```

Code that you generate from Simulink software models by using Embedded Coder software automatically include the model reference capability. You do not need to set the flag.

Reported Limitations and Tips

Reported Issues

Some long-standing issues affect the Embedded Coder software. When you are using `adivdsp` objects and methods to work with VisualDSP++ software and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. PIL means "processor-in-the-loop" and is similar to hardware-in-the-loop operations.

Using 64-bit Symbols in a 64-bit Memory Section on SHARC Processors

VisualDSP++ compiler design prevents Embedded Coder from generating code the accesses 64-bit memory locations correctly. To avoid unexpected results, do not allocate 64-bit data or symbols to 64-bit memory locations on SHARC processors.

When 64-bit data is in 64-bit memory, the compiler generates code that accesses the 64-bit locations as two 32-bit values. Thus, the code does not read and write the 64-bit data correctly. It reads or writes every other 32-bit location, returning or writing incorrect values and possibly exceeding the allocated memory.

Refer to pp. 5-33 in the *ADSP-2136x SHARC Processor Programming Reference, revision 1.0* for a description of how the compiler treats 64-bit (long word) data values.

Working with Eclipse IDE

- "Tested Software Versions" on page 48-2
- "Installing Third-Party Software for Eclipse" on page 48-4
- "Configuring Your MathWorks Software to Work with Eclipse" on page 48-10
- "Troubleshooting with Eclipse IDE" on page 48-15

Note To use the coder product with Eclipse IDE, complete the steps in "Installing Third-Party Software for Eclipse" on page 48-4 and "Configuring Your MathWorks Software to Work with Eclipse" on page 48-10

Tested Software Versions

MathWorks has tested the coder product with the specific software versions listed in the following tables.

Required for all platforms	Tested Versions
Sun™ Java Runtime Environment (JRE)	JRE 6.0 (Java 1.6.x)
Eclipse IDE for C/C++ Developers package, which includes the CDT feature	Ganymede (Eclipse 3.4)
CDT (If CDT is installed separately from Eclipse IDE for C/C++ Developers package, match CDT version with Eclipse version.)	CDT 5.0

Linux: Additional Software Required	Tested Versions
GNU GCC (compiler)	GCC 4.3.x
GNU as (assembler — part of the GNU binutils package)	as 2.18
GNU ar (archiver — part of the GNU binutils package)	ar 2.18
GNU GDB (debugger)	GDB 6.8.x
GNU make	make 3.81

Windows: Additional Software Required	Tested Versions
MinGW	5.1.x
GDB	GDB 6.3.x
MSYS	1.0.11

You can try untested versions and combinations of third-party software at your own risk.

For the most current information about using the coder product software with Eclipse IDE, see: www.mathworks.com/products/embedded-coder/eclipse-adaptor.html

Installing Third-Party Software for Eclipse

In this section...

"Installing Sun Java Runtime Environment (JRE)" on page 48-4

"Installing Eclipse IDE for C/C++ Developers" on page 48-4

"Verifying the GNU Tool Chain on Linux" on page 48-5

"Installing the GNU Tool Chain on Windows" on page 48-7

Installing Sun Java Runtime Environment (JRE)

To install the JRE, complete the following steps:

1 At your Windows or Linux command prompt, enter:

```
java -version
```

If Java is present, the command line responds with the version information, as this example shows.

```
$ java -version
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04)
$
```

2 If Java is missing or the version is lower than 1.6.x, download and install JRE 6.0 from http://www.java.com.

3 Verify that Java is working by entering `java -version` again or by visiting http://www.java.com/en/download/help/testvm.xml.

Installing Eclipse IDE for C/C++ Developers

Note The following instructions are based on Eclipse 3.4 (Ganymede). More recent versions of the Eclipse IDE can have different appearances, menus, or software package names.

The Eclipse IDE for C/C++ Developers package includes the Eclipse IDE and the C/C++ Development Tools (CDT). To install Eclipse IDE for C/C++ Developers package, complete the following steps:

1. Download the Ganymede SR2 zip file for Eclipse IDE for C/C++ Developers, from http://www.eclipse.org/downloads/packages/release/ganymede/sr2.

2. Extract the Eclipse files to a permanent location, such as `C:\eclipse\` and create a desktop shortcut to `eclipse.exe`.

3. Start Eclipse, and select **Help** > **Software Updates**.

4. Look under the **Installed Software** tab, and verify that Eclipse has the following three CDT software packages.

   - Eclipse C/C++ Development Platform
   - Eclipse C/C++ Development Tools
   - Mylin Bridge: C/C++ Development

If you have a previous Eclipse installation that does not include CDT, complete the following steps:

1. In Eclipse, select **Help** > **Software Updates**.

2. Click the **Available Software** tab.

3. Click **Ganymede Update Site**.

4. Select **C and C++ Development**, and click **Install**.

5. When the installation process completes, click the **Installed Software** tab, and verify that you have CDT.

Verifying the GNU Tool Chain on Linux

Most Linux distributions include the following GNU C/C++ development tools. Eclipse and CDT require these tools to compile code, build projects, and debug applications:

- Assembler (as)
- Archiver (ar)

- compiler and linker (gcc)
- debugger (gdb)
- build utility (make)

Verify that the GNU tools are present and set the tool chain path:

1 On the Linux command line, enter:

   - `gcc --version`
   - `gdb --version`
   - `as --version`
   - `ar --version`
   - `make --version`

2 Compare the version of each tool with the following list of tested versions:

   - gcc 4.3.x
   - as 2.18
   - ar 2.18
   - gdb 6.8.x
   - make 3.81

   If you are using Eclipse for targeting Linux, disregard the version numbers in the preceding list.

   You can use versions of the GNU tools that are more recent than the tested versions at your own risk.

   To install a missing tool or to change the version of the tool, use the software installation manager that comes with your Linux distribution.

   Alternatively, visit http://directory.fsf.org/GNU/ for more information about individual tools. Source files for the tools are available from:

   - binutils (includes as and ar), http://ftp.gnu.org/gnu/binutils/
   - gcc, http://ftp.gnu.org/gnu/gcc/

- gdb, http://ftp.gnu.org/gnu/gdb/
- make, http://ftp.gnu.org/gnu/make/

3 Modify the PATH environment using the appropriate commands for your running shell. You can also modify the path environment variable in your login scripts.

   If you are using a Bash shell prompt, enter:

```
PATH=my_tool_path:$PATH
```

   Where `my_tool_path` is the path to the GNU tool binaries. For example:

```
PATH=/bin:$PATH
```

   If you are using a C shell prompt, enter:

```
setenv PATH my_tool_path:$PATH
```

   Where `my_tool_path` is the path to the GNU tool binaries. For example:

```
setenv PATH /bin:$PATH
```

Installing the GNU Tool Chain on Windows

Windows typically does not include GNU C/C++ development tools. Eclipse and CDT require these tools to compile code, build projects, and debug applications.

Provide a GNU tool chain for Windows by installing MinGW:

1 Open http://sourceforge.net/projects/mingw/files/.

2 Download and run the latest version of the “Automated MinGW Installer".

   Note The earliest version of MinGW available is more recent than the tested version.

3 Start the MinGW installation wizard to perform a default installation.

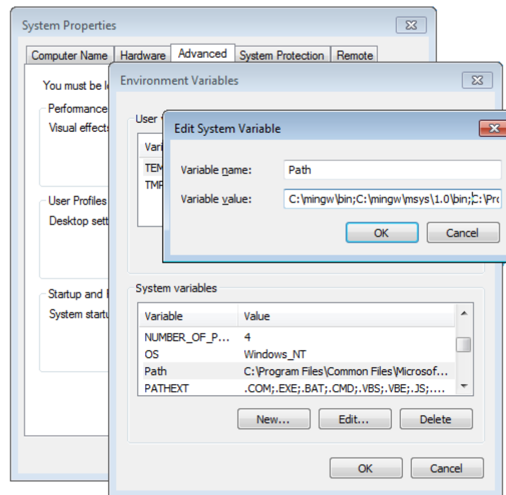
Perform a default installation until you reach **Select Components**. At that step, select **MSYS Basic System**.

Then, complete the default installation process. Wait for the installation wizard to download, and install additional files from the Internet.

Note If you alter the default installation path, C:/MinGW, do not use spaces in the new path.

Set the PATH environment variable:

1. In Windows, right-click **My Computer** or **Computer**, and choose **Properties**.
2. Then select **Advanced** or **Advanced system settings**, and click **Environment Variables**.
3. Under **System variables**, scroll down to the **Path** variable.
4. Select **Path**, and click **Edit**.
5. Ensure that the operating system calls the GNU tools when there are multiple paths:
   a. Add the paths of the MinGW and MSYS bin folders to the beginning of the **Variable value**.
   b. Use semicolons to separate the paths. For example, `C:\mingw\bin;C:\mingw\msys\1.0\bin;`



6 Reboot your system.

7 To verify the GNU tools installation and path settings, enter the following commands on the Windows command line:

- `gcc --version`
- `gdb --version`
- `as --version`
- `ar --version`
- `make --version`

Each command displays the tool name and version on the command line. If you receive a message that the command is not recognized, verify that you completed the preceding installation and path configuration instructions.

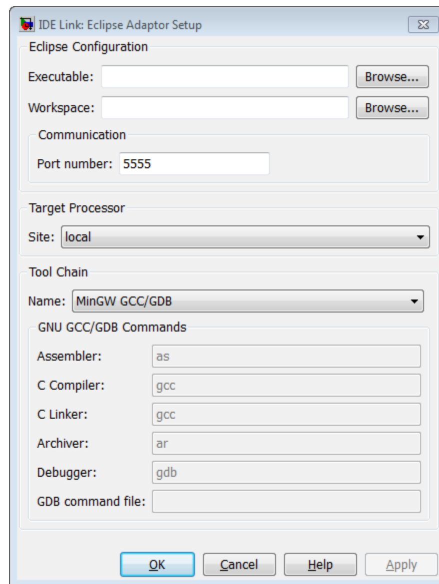
You can use versions of the GNU tools that are more recent than the tested versions at your own risk.

Configuring Your MathWorks Software to Work with Eclipse

After you install the third-party software, configure the coder product to work with Eclipse:

1. Close Eclipse IDE before you run `eclipseidesetup`. For more information, see "Build Errors" on page 48-16.

2. At the MATLAB command line, enter `eclipseidesetup`. The coder product displays the following dialog box.



3. Update **Executable** with the location and file name of the Eclipse application file. For example, `C:\eclipse\eclipse.exe`.

   You can get this value by right-clicking a shortcut for Eclipse and looking at the properties.

4. Update **Workspace** with the default location where Eclipse creates and saves new project files. For example, `C:\WINNT\Profiles\username\workspace`.

To find the current workspace, open Eclipse and select **File** > **Switch Workspace** > **Other**.

In the future, if you change the Eclipse workspace, repeat this configuration procedure.

Do not use workspace paths that contain spaces. If you have a path with spaces, recreate the workspace, and then update the path in Eclipse.

5. For **Port number**, enter a valid, unused, IP port number. For example, 5555.

6. For **Site**, identify where the coder product uploads and runs the executable file upon completing the build process. Use either of these options:

   - Choose `local` to run the executable on your Linux workstation.
   - Choose `remote` to download the executable to a remote target running Linux operating system over a network connection (for example, to connect to an embedded system connection to the Ethernet port on your workstation).

     You must perform additional steps to connect to a remote target running Linux. See:"Additional Configuration Steps to Run Your Executable on a Remote Embedded Linux Target" on page 48-13.

Note Later on, when you are working on your model, open the Target Preferences block, and set **Processor** to match the processor at the **Site** you selected.

---

7. To customize the Tool Chain settings, see the **Custom GCC/GDB** topic.

8. When you click **OK** or **Apply**, the coder product:

   - Verifies the locations of the Executable and Workspace in the Eclipse Adaptor Setup dialog box.
   - Verifies that the required third-party software is present.
   - Installs the coder product plug-ins in the Eclipse `plugins` folder. For example, in `C:\Program Files\eclipse\plugins\`.

- Saves configuration information to the `mwidelink.ini` file, located in the Eclipse `plugins` folder.

Note When Eclipse starts, it loads the coder product plug-in. The coder product plug-in loads the port number from `mwidelink.ini`. To resolve a port number conflict, change the port number by running `eclipseidesetup` again. Do not edit `mwidelink.ini`.

---

9 To verify that the configuration process is complete, create a handle object for the Eclipse IDE. Enter the following command in MATLAB

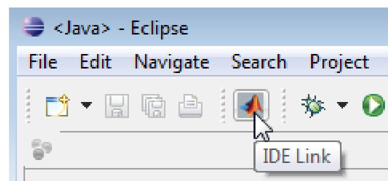
```
IDE_Obj = eclipseide
```

This command, starts Eclipse IDE if it is not already running, and creates a handle object. For example:

```
Starting Eclipse(TM) IDE...

ECLIPSEIDE Object:
  Default timeout  : 10.00 secs
  Eclipse folder   : C:\eclipse3.4\eclipse
  Eclipse workspace: C:\WINNT\Profiles\rolfedh\workspace
  Port number      : 5555
  Processor site   : local
```

If you are using more recent versions of the GNU tools, you can disregard command-line warnings about using untested versions.

10 In Eclipse, click the following icon to see the status of the IDE Link plug-in.



Additional Configuration Steps to Run Your Executable on a Remote Embedded Linux Target

On Linux host systems, you can configure the coder product to download and run an executable on a remote target running Embedded Linux.

During the "Configuring Your MathWorks Software to Work with Eclipse" on page 48-10 process, complete these additional steps:

1. Set **Site** to `remote`. The dialog box displays additional **Target Processor** and **GNU GCC/GDB Commands** parameters.

2. Under **Target Processor**, enter the values the coder product uses to connect to the target processor over the network:

   - **User login**: Supply a user name that has trusted "r-" access to the remote system. The user name must appear in the /etc/hosts.equiv or $HOME/.rhosts files on the remote system.
   - **IP address**: Enter the IP address of the remote system. To test the software on your local system instead of the remote system, enter `localhost`.
   - **Port number**: Enter the IP port number for communications between the two systems. For example, `10000`.
   - **Download path**: Enter the path on the remote system that receives the compiled executable and related files. For example, `./` sends the files to the home folder of the user login.

3. Under **GNU GCC/GDB Commands**, enter the tool chain commands and optional arguments the coder product uses to build executable for the target processor.

   For example, if you are using the generic GNU tool chain to build an executable for a target processor running Embedded Linux, enter:

   - **Assembler**: `as`
   - **C Compiler**: `gcc`
   - **C Linker**: `gcc`
   - **Archiver**: `ar`

- **Debugger**: `gdb`

For example, if you are using the MontaVista Linux tool chain to build an executable for an ARM processor running Embedded Linux, enter:

- **Assembler**: `arm_v5t_le-as`
- **C Compiler**: `arm_v5t_le-gcc`
- **C Linker**: `arm_v5t_le-gcc`
- **Archiver**: `arm_v5t_le-ar`
- **Debugger**: `arm_v5t_le-gdb`

4 Click **OK** to complete the Eclipse Adaptor Setup process.

Also see: Chapter 52, “Working with Linux Target”

Troubleshooting with Eclipse IDE

In this section...

"SIGSEGV Segmentation Fault for GDB" on page 48-15

"GDB Stops on Each Semaphore Post" on page 48-15

"Build Errors" on page 48-16

"Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux" on page 48-16

"Eclipse Message: "Can't find a source file"" on page 48-16

"Eclipse Message: "Cannot access memory at address"" on page 48-17

SIGSEGV Segmentation Fault for GDB

If you use Comodo Internet Security (CIS) software on your development system, CIS causes a SIGSEGV segmentation fault for GDB. When this fault occurs, you receive the following message:

```
Debugger name and version: GNU gdb (GDB) 7.0
Program received signal SIGSEGV, Segmentation fault.
In ntdll!RtlpWaitForCriticalSection () (C:\WINDOWS\system32\ntdll.dll)
Continuing...
Program received signal SIGSEGV, Segmentation fault.
In ?? () (C:\WINDOWS\system32\guard32.dll)
```

If you get this message, click **OK** and then click **Continue**.

For more information, see the "Guard32.dll causes SIGSEGV segmentation fault for GDB debugger CIS 3.9.95478 x32" topic athttp://forums.comodo.com/.

GDB Stops on Each Semaphore Post

If you use gdb to debug a target application running on Linux , gdb stops on each semaphore post. You can override this expected behavior adding the following text to `.gdbinit`, the GDB init file:

```
handle SIG34 nostop noprint pass
handle SIG35 nostop noprint pass
```

On Linux, .gdbinit resides on your home folder, by default. For more information about creating `.gdbinit` and configuring gdb, consult the *GDB User Manual*, available from http://www.gnu.org/software/gdb/documentation/.

Build Errors

If you use `eclipseidesetup` without closing Eclipse IDE, you may get build errors similar to the following ones:

```
The call to idelink_ert_make_rtw_hook, during the exit hook generated the following error:
Error while creating the project.

The build process will terminate as a result.
===
Error while creating the project.
===
Error creating a new project.
===
An exception occurred while performing this operation. 0
```

To solve this problem, close and restart Eclipse IDE.

Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux

Profiling is not available for Intel® x86/Pentium and AMD® K5/K6/Athlon processors running Windows or Linux.

Eclipse Message: "Can't find a source file"

With specific Configuration Parameters, while building and loading a target application, Eclipse IDE displays a message that it could not find a source file. This message appears even if the load action completes successfully. Here is an example of the message:

```
Can't find a source file at
"../../sumdiff_bash_eclipseide/sumdiff_bash_main.c
Locate the file or edit the source lookup path
```

```
to include its location.
```

In Configuration Parameters, on the IDE Link pane, in the Vendor Tool Chain section: If **Configuration** is set to `Release` or `Custom`, the coder product does not specify the `-g` compiler option for gcc. Therefore, the build process does not produce debugging information gdb requires. Without this information, gdb cannot map the executable to the source file, resulting in the "Can't find a source file" message.

To solve this problem, add `-g` to the **Compiler options string** for the `Custom` and `Release` configurations, or set **Configuration** to `Debug`.

Eclipse Message: "Cannot access memory at address"

If you use the coder product's `halt` method to stop the target application, Eclipse displays a message similar to the following example:

```
[Switching to thread 5528.0x1664]
Quit (expect signal SIGINT when the program is resumed)
Cannot access memory at address 0x720000
Cannot access memory at address 0x720000
```

This error is not related to Eclipse IDE. It is a bug with gdb/MinGW. It typically occurs when gdb tries to access an invalid or protected memory location.

49

Working with Freescale MPC5xx Processors

- "Getting Started" on page 49-2
- "Generating Stand-Alone Real-Time Applications" on page 49-24
- "PIL Simulation" on page 49-82
- "Configuration Parameters" on page 49-109
- "Toolchains and Hardware" on page 49-124

Getting Started

This section contains the following topics:

In this section...
"Overview" on page 49-2
"Additional Blocks on MATLAB Central Web Site" on page 49-8
"Using This Guide" on page 49-8
"CAN Hardware Requirements for Freescale MPC5xx" on page 49-9
"Supported Cross-Development Tools for Freescale MPC5xx" on page 49-9
"Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation" on page 49-10
"Setting Target Preferences for MPC5xx" on page 49-11
"Accessing Utilities for Freescale MPC555" on page 49-18
"Data Type Support and Scaling for Device Driver Blocks" on page 49-20

Overview

- "Introduction" on page 49-2
- "Feature Summary" on page 49-3
- "Applications for the coder product" on page 49-6

Introduction

The coder product provides a complete and unified set of tools for developing embedded applications for the Freescale MPC555 and MPC56x processors (MPC561, MPC562, MPC563, MPC564, MPC565 and MPC566). The MPC5xx family of processors are products of Freescale Semiconductor, Inc., formerly a division of Motorola, Inc.

Used in conjunction with Simulink and Stateflow, your coder product lets you:

- Design and model your system and algorithms.

- Compile, download, run and debug generated code on the target hardware, seamlessly integrating with industry-standard compilers and development tools for the MPC5xx.
- Use simulation and rapid prototyping techniques to evaluate performance and validate results obtained from generated code running on the target hardware.
- Deploy production code on the target hardware.

Feature Summary

Production Code Generation.

- The coder product generates production code for use on the target MPC5xx microcontroller.
- The coder product generates project or makefiles for popular cross-development systems:
  - Wind River Systems Wind River Compiler
  - Freescale CodeWarrior®
- Debugger support:
  - Wind River Systems SingleStep™ debugger
  - Freescale CodeWarrior debugger
- Support for ANSI C (ANSI X3.159-1989) math library for floating-point functions.

Device Driver Support.

- The Embedded Targets block library provides device driver blocks that let your applications access on-chip resources. The I/O blocks support the following features of the MPC555 and MPC56x:
  - Pulse width modulation (PWM) generation via the Modular Input/Output Subsystem (MIOS) PWM unit or the Time Processor Unit 3 (TPU) modules
  - Analog input via the Queued Analog-to-Digital Converter (QADC64)

- Digital input and output via the MIOS or TPU
- Digital input via the QADC64
- Frequency and pulse width measurement via the MIOS Double Action Submodule (MDASM)
- Transmit or receive Controller Area Network (CAN) messages via the MPC5xx TouCAN modules
- Driver blocks to support other functions of the TPU modules — Fast Quadrature Decode, New Input Capture/Input Transition Counter, and Programmable Time Accumulator
- Serial transmit and receive
- Utility blocks such as a watchdog timer

Code and Performance Analysis. Web-viewable code generation report includes

- Analysis of RAM/ROM usage and other variables
- Analysis of code generation options used, with optimization suggestions
- Hyperlinks to all generated code files
- Hyperlinks from generated code to source model in Simulink

Applications Development and Rapid Prototyping.

- Generation of real-time, stand-alone code for MPC5xx
- Scheduler and time functions for singlerate or multirate real-time operation
- CAN-based loader for download of generated code to RAM or flash memory
- CAN-based host-target communications for non-real-time retrieval of data on host computer

Simulation and Simulation.

- Automatic S-function generation lets you validate your generated code in software-in-the-loop (SIL) simulation.

- Processor-in-the-loop (PIL) simulation lets you integrate generated code, running on the target processor, into your simulation.
- SIL and PIL code components are generated by the coder product. These simulation components are in the same compact and efficient format as the production code generated for final deployment.

CAN Support.

- Transmit or receive CAN messages via the MPC5xx TouCAN modules.
- CAN Drivers (Vector) library provides blocks for transmitting, receiving, configuring, and connecting to Vector-Informatik CAN hardware and drivers. These can be used in simulation to connect to a real CAN bus.
- The CAN Message Blocks library includes blocks for transmitting, receiving, decoding, and formatting CAN messages. It also supports message specification via the Vector-Informatik CANdb standard. CAN is an industry standard protocol used in automotive electronics and many other embedded environments where dispersed components require sharing of information.

Code Validation and Performance Analysis.

Code Validation

Since signal data is available to Simulink during each sample interval in a PIL simulation, you can observe signal data on Scope blocks or other Simulink signal viewing blocks. You can also store signal data to MAT-files via To File blocks. To validate the results obtained by the generated code running on the target processor, you can compare these files to results obtained using a normal Simulink plant/controller simulation.

Determining Code Size

In control design it is critical to ensure that the size of the generated code does not exceed physical limitations of RAM and ROM. The coder product can automatically produce a code generation report that displays the RAM usage and ROM size of the generated code.

This capability is useful when selecting which code generation optimizations will be used. After determining the size of the required RAM and ROM, you can consider which code generation optimizations to use, and consider modifications to the modeling style.

Applications for the coder product

The coder product provides targets that support three application scenarios:

- Real-time (RT) execution for production and rapid prototyping
- Processor-in-the-loop (PIL) simulation target
- Algorithm export (AE) target

In the sections that follow, we summarize typical applications and the tasks you will need to perform for each; we also provide links to the relevant documentation.

Real-Time Execution and Rapid Prototyping. The Embedded Coder real-time target enables you to use your controller block diagram in real time to perform embedded control. With this target, you can add I/O blocks for the MPC5xx to your controller subsystem, generate and build code, download to the target, and run the generated C code.

When you first begin using the RT target, see “Tutorial: Creating a New Application” on page 49-26, which demonstrates the following topics through the use of a simple model with a device driver:

- Examining the demo model with a plant model and controller
- Adding the MPC555 Resource Configuration block to your subsystem
- Adding I/O device drivers from the Embedded Targets block library
- Selecting the RT target
- Generating code for real time
- Downloading code with
  - A BDM connector
  - CAN

- Running the generated code in real-time

You may also be interested in generating code analysis information from your RT target build. See "HTML Code Analysis (RAM/ROM) Report" on page 49-104 for details.

Processor-in-the-Loop. The processor-in-the-loop (PIL) target lets you run a simulation of a closed-loop Simulink model for the purpose of code validation and analysis. When running a PIL simulation, you use a closed-loop model with two major components: a plant model and a controller. The plant model may contain any Simulink blocks including a combination of continuous-time and discrete-time blocks.

To get started with the PIL target, see "Tutorial 1: Building and Running a PIL Simulation" on page 49-84. The tutorial covers the following topics:

- Opening the demo model and examining the plant model and controller
- Selecting the PIL target
- Generating the Embedded Real-Time (ERT) S-function and the corresponding library block
- Inserting the S-function back into the closed-loop model
- Automatic downloading of generated code with
  - Wind River Systems SingleStep debugger and a Background Debug Mode (BDM) port connector
  - CodeWarrior and a BDM connector
- Running a PIL simulation

You may also be interested in generating code analysis information from your PIL target build. See "HTML Code Analysis (RAM/ROM) Report" on page 49-104 for details.

Algorithm Export. The algorithm export (AE) target enables you to generate code for your controller subsystem and build the code as a stand-alone executable for use on the MPC5xx. The difference between the AE and the PIL target is that the AE target eliminates all extraneous code (such as serial communications code) used for simulation, and also eliminates any real-time interrupts. The AE target therefore generates code only for the basic controller subsystem (e.g. algorithm code). You can then modify or customize this code for your own special purposes.

In contrast, the RT target provides turnkey code including interrupt service routines, driver code, and underlying initialization code for the MPC5xx. Depending upon your particular application, you may find it more valuable to begin with the AE target baseline, and extend this environment for your own use.

The AE target is documented in “Algorithm Export Target” on page 49-103.

Like the PIL and RT targets, the AE target supports generation of code analysis information. See “HTML Code Analysis (RAM/ROM) Report” on page 49-104 for details.

Additional Blocks on MATLAB Central Web Site

Check the MATLAB Central Web site for user- and developer-contributed blocks and demos, such as the MPC555 Motor Control Function Blockset for Release 2006a.

The MPC555 Motor Control Function Blockset is an extensive collection of additional TPU I/O blocks for the coder product. This functionality is particularly useful in the context of motor and powertrain control, including functions for missing and additional tooth detection.

Using This Guide

To get acquainted with the coder product and gain hands-on experience with the features most relevant to your interests:

- Read “Getting Started” on page 49-2 in its entirety, paying particular attention to “Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation” on page 49-10.

- If you are interested in using the supplied device driver blocks and in deploying stand-alone, real-time applications on the MPC5xx, read "Generating Stand-Alone Real-Time Applications" on page 49-24. Work through the "Tutorial: Creating a New Application" on page 49-26.
- If you are interested in processor-in-the-loop (PIL) simulation, read "PIL Simulation" on page 49-82 to learn about the PIL target. Work through the "Tutorial 1: Building and Running a PIL Simulation" on page 49-84.
- Then, for in-depth information about the Embedded Targets device drivers and other blocks, see "Freescale MPC5xx" It is particularly important to read MPC5xx MPC555 Resource Configuration, as the MPC555 Resource Configuration block is required to use most of the device driver blocks.

CAN Hardware Requirements for Freescale MPC5xx

If you want to use CAN to transmit or receive CAN messages between your host PC and your target, you require Vector-Informatik CAN hardware supported by the Vector CAN Driver Library. You must install the correct driver libraries to support profiling, downloading, and external mode.

Note For CANcaseXL, you must install *both* the Vector XL-driver library and Vector CAN Driver Library `vcand32.dll`.

For older CAN hardware, you must install the Vector CAN Driver Library `vcand32.dll`.

Make sure that the library, `vcand32.dll`, is placed in the Windows `system32` folder.

For configuration steps, see "CAN Hardware and Drivers" on page 49-139.

Supported Cross-Development Tools for Freescale MPC5xx

In addition to the required MathWorks software, a supported cross-development environment is required. The coder product currently supports the cross-development tools listed below; please read carefully the limitations noted:

- Freescale CodeWarrior Development Studio, MPC5xx Edition, v8.7 (debug via Macraigor Systems Wiggler, Raven/ Blackbird, or On-board BDM).

  Search on the Freescale Web site: http://www.freescale.com

- Wind River Systems Wind River Compiler version 5.4.0, (formerly known as Diab), and Wind River Systems SingleStep debugger of the following versions:

  - Wind River Systems SingleStep with vision Version 7.7.5 (debug via Wind River visionPROBE) (for MPC5xx)

  - Wind River Systems SingleStep Version 7.6.6 (debug via Macraigor Systems Wiggler, Raven / Blackbird, On-board BDM) (for MPC555 only) (+ Fromelf patch from Wind River Support)

    You must download `fromelf.exe` for the Wind River Systems SingleStep debugger 7.6.6, otherwise builds with debug flag `-g` set will not load, with the following error: "aborting due to failure of ELF reader".

    To use these BDM devices you must set up nondefault target preferences, as detailed in "Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation" on page 49-10.

  See Wind River Products for information.

The full feature set (PIL, RT, and AE targets) is supported for both toolchains.

Before using the coder product with any of the above cross-development tools, please be sure to read and follow the instructions in "Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation" on page 49-10.

See also this solution for Information about the availablity of SingleStep.

Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation

The next sections describe how to configure your development environment (compiler, debugger, etc.) for use with the coder product and verify correct operation. The initial configuration steps are described in the following sections:

- You must set up your development environment and your target hardware. Information on these settings can be found in the "Toolchains and Hardware" on page 49-124:
  - "Setting Up Your Target Hardware" on page 49-133
  - "Setting Up Your Toolchain" on page 49-124

Note You MUST check your jumper settings. Incorrect operation or even hardware damage may occur if you do not. See "Jumper Settings" on page 49-134.

- You must configure the product to work with your toolchain by specifying the locations of your compiler and debugger. This is described in the section "Setting Target Preferences for MPC5xx" on page 49-11.
- We supply a test program to verify your installation. This confirms you have correctly set up your toolchain, and development board. See "Run Test Program" on page 49-15.
- The next step is to download boot code to the flash memory of your MPC5xx. See "Download Boot Code to Flash Memory" on page 49-16.

Note You must download the new boot code if you have used a previous release of the coder product with your hardware. See "Download Boot Code to Flash Memory" on page 49-16.

Once you have completed these steps, run the tutorials in subsequent sections to get started.

Setting Target Preferences for MPC5xx

- "Configuring the coder product for Your Cross-Development Toolchain" on page 49-12
- "Run Test Program" on page 49-15
- "Download Boot Code to Flash Memory" on page 49-16

Configuring the coder product for Your Cross-Development Toolchain

This section describes how to set target preferences associated with the coder product. These settings persist across MATLAB sessions and different models. Target preferences let you specify the location of your MPC5xx cross-compiler, the communications port to be used for downloading code, and other parameters affecting the generation, building, and downloading of code.

You must make sure you localize the settings to suit your PC and cross-development toolchain. It is important that you set the correct path to your compiler and debugger using the Target Preferences dialog box.

Instructions for setting up specific third-party toolchains for use with the coder product are in "Toolchains and Hardware" on page 49-124. Make sure you have followed the instructions to set up your toolchain first:

- "Setting Up Your Installation with Wind River Compiler and Wind River Systems SingleStep Debugger" on page 49-124
  - "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126. Note especially the settings you must change if you are not using the visionPROBE BDM device. The defaults are set up for the visionPROBE.
- "Setting Up Your Installation with Freescale CodeWarrior" on page 49-129
  - "Set Target Preferences for CodeWarrior" on page 49-131

You can modify target preference objects via the Target Preferences dialog box:

1. Enter `mpc555utils` in the Command Window. This opens the Utilities for Use with MPC555 dialog box.

2. Select `Target Preferences` from the drop-down list, and click **OK**. This opens the Target Preferences dialog box where you can edit the settings for your cross-development environment. When you first open the dialog the following settings are visible.

3. Select `Diab` or `CodeWarrior` from the drop-down **Toolchain** menu.

Note the Wind River Compiler was formerly known as Diab. Any appearances of the term *Diab* in the documentation and / or product should be understood to refer to the Wind River Compiler.

4 Expand **ToolChainOptions** (by clicking the plus sign) and type the correct path into **CompilerPath**. The following shows Wind River Compiler options. Note that the defaults are set up for the visionPROBE — see the Appendix for settings to use another BDM device, described in "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126.

Note The drive designated in the compiler and debugger paths must be either an actual hard drive on your PC, or a mapped drive. Do not use a Universal Naming Convention (UNC).

---

5 For Wind River Systems SingleStep you must also type the correct path into **Debugger Path**. This is not necessary for CodeWarrior as the compiler and debugger are integrated. The example below shows the CodeWarrior preferences.

There are other settings in the target preferences you can see by expanding all the options.

Serial Communications. These target preferences relate to Processor-in-the-Loop (PIL) simulation only.

SerialCommunications	mpc555.CommPort
BitRate	57600
HostPort	com1
TargetPort	com1
TimeOut	4

- `BitRate` — Bit rate (in bps) for host/target communications. The default is `57600`.
- `HostPort` — Host serial port for host/target communications. Select from `com1` to `com8`; the default is `com1`.
- TargetPort — Target board serial port for host/target communications. Select from `com1` to `com8`; the default is `com1`.

- `TimeOut` — Timeout value (in seconds) for the serial communications port. The default is `4`.

Target Board.

TargetBoard	mpc555.TargetBoard
OscillatorFrequency	20
ProcessorVariant	555

- `OscillatorFrequency` — Choose either `20` MHz (the default) or `4` MHz if you are using a 4MHz board.
- `ProcessorVariant` — Here you can select from `555`, `561`, `562`, `563`, `564`, `565` or `566` to match your target processor. The default is the MPC555.

When you install bootcode after setting target preferences the correct bootcode for your chosen target processor and oscillator frequency will be automatically installed. Note that you also need to make these settings match in your models for the non-default target processor and oscillator frequency. See "Configuration for Nondefault Hardware" on page 49-141.

Compiler Optimization Switches.

ToolChain	Diab
ToolChainOptions	mpc555.DiabOptions
CompilerOptimizationSwitches	mpc555.CompilerOptimizationSwitches
Debug	-g
Size	-XO -Xsize-opt
Speed	-XO
CompilerPath	d:\applications\WindRiver
DebuggerExecutable	visppc.exe
DebuggerPath	d:\applications\sds773
DebuggerSwitches	-g -r - -p visionPROBE:LPT1

ToolChain	CodeWarrior
ToolChainOptions	mpc555.CodeWarriorOptions
CompilerOptimizationSwitches	mpc555.CompilerOptimizationSwitches
Debug	-gdwarf2
Size	-opt space
Speed	-opt speed
CompilerPath	d:\applications\CodeWarrior

For both toolchains these settings configure optimizations for `speed`, `size`, and `debug`. The settings are compiler specific. These properties can be edited from the Target Preferences dialog box or from the Configuration Parameters dialog box, described below. The defaults should be adequate for most rapid prototyping purposes.

If you want to alter these settings, consult your compiler documentation for specific optimizations. To edit the settings,

- If you want your changes to apply to many models, edit them within the Target Preferences dialog box. Your settings will appear within the Configuration Parameters dialog box in the **Compiler optimization switches** field when you select `speed`, `size` or `debug` from the **Optimize compiler for** options in the drop-down menu. You must choose `ET MPC5xx real-time options (1)` from the **Code Generation** tree to reach these settings.
- If you want to customize these settings for a single model, edit them from the Configuration Parameters dialog box. **Optimize compiler for** will change to `custom` and the defaults for these settings will remain unchanged in the Target Preferences dialog box. When you edit these settings, you must place single quotation marks at either end of the string. These settings are then applied to model code.

Use Prebuilt Libraries

This check box option (selected by default) determines whether prebuilt libraries, compiled with default compiler switches, are linked against during compilation of the generated code. When this option is not selected, the source modules that comprise these libraries will be compiled individually in the model build folder, using the currently selected compiler switches. Using prebuilt libraries saves a considerable amount of time during the build process

Debugger Switches. This setting is specific to Wind River Systems SingleStep. See "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126.

Run Test Program

To verify your setup, you can download and run a simple test program on the phyCORE-MPC555 board:

1 .Enter `mpc555utils` in the Command Window. This opens the Utilities for Use with MPC555 dialog box.

2 Select `Run Simple MPC5xx Test Application (via BDM)` from the drop-down list, and click **OK**.

3 To the question `Do you wish to load the applications?  (Y/N)` type `Y` at the command line.

If you have not set up your target preferences properly the process will stop and ask you to do this now.

Watch as your toolchain downloads and runs the application on your phyCORE board. Successful execution results in a blinking LED.

You have now verified your installation and are ready to begin working with the coder product.

Download Boot Code to Flash Memory

The next step is to download the boot code to flash memory, if you have not already done so. Normally, you will only need to program the boot code into flash memory once. After this is done, new application code can be downloaded as often as required without any changes to the boot code.

The first time you program the boot code into the target hardware, you must download it via the BDM port. However, if existing boot code is already programmed into flash memory and must be replaced (for example, with a newer or modified version) it is also possible to download entirely over CAN or serial. If you are upgrading from a previous release of the coder product you must download the new boot code.

If your target does not have bootcode already you can only install new bootcode with a BDM. See the next section "Installing Bootcode via BDM and Serial or CAN" on page 49-17. For existing bootcode, you can use a BDM or CAN; with bootcode from version 1.2 or later you can also download over Serial. See "Installing Bootcode Without a BDM" on page 49-17.

The first time you use the coder product you must use a toolchain to download boot code to the MPC555 flash memory. Once the boot code is loaded into flash memory, you can download code to the processor entirely over serial or the CAN network as described in the tutorials. See "Overview of Memory Organization and the Boot Process" on page 49-40 for more information.

Installing Bootcode via BDM and Serial or CAN. To install bootcode, follow these steps:

1. Connect the BDM cable to the target, and a serial or CAN cable. If you do not have a BDM available, see "Installing Bootcode Without a BDM" on page 49-17.

2. .Enter `mpc555utils` in the Command Window. This opens the Utilities for Use with MPC555 dialog box.

3. Select `Install MPC5xx Bootcode` from the drop-down list, and click **OK**.

   A dialog appears asking if you are connected to the target via BDM. Read the information on the dialog.

4. Click **Yes**.

   Your toolchain is launched and prepares to download.

   The **Download Control Panel** appears.

5. If you are using CAN (the default) you can proceed to step 5. If you are using serial to connect to the target, click the **Communications Options** tab in the **Download Control Panel** and select `Serial` from the **Connection type** drop-down menu.

6. On the **Download** tab, click **Start Download**.

Your development tools execute a command to install the boot code. When the process stops, the messages in the **Download Control Panel** complete, and the **Stop Download** button reverts to **Start Download**. The boot code should now be installed.

Installing Bootcode Without a BDM. If your target does not have bootcode already you can only install new bootcode with a BDM. For targets with existing bootcode, if you do not have a BDM available you can install bootcode as follows:

- For a target with R14 bootcode, you can install new bootcode using the **Start** menu exactly as described above except step 4 - click **No** when asked

if you are connected via BDM. The download should complete successfully over serial or CAN.

- If existing bootcode on the target is version 1.1 (R13+SP1), you can install bootcode without a BDM if you have CAN. Use the **Start** menu bootcode installer as described above and click **No** when asked if connected by BDM. The download should complete successfully over CAN.

Note If the existing bootcode is earlier than version 1.1 (if it is R12.1 or R13), you need to upgrade bootcode with a BDM. If no BDM is available, please contact Technical Support for a solution.

Once you have successfully downloaded boot code to your target, you have completed your installation and are ready to use all the product features. If necessary, please consult your toolchain documentation.

Now turn to "Generating Stand-Alone Real-Time Applications" on page 49-24 to get hands-on experience with using the coder product and your toolchain to generate, download, and execute application code on your phyCORE-MPC555 board. You can then also work through the tutorials in "PIL Simulation" on page 49-82 to start using processor-in-the-loop simulation for development via the coder product.

Accessing Utilities for Freescale MPC555

Utilities for Use with MPC555 Dialog

Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window or double-clicking Launch MPC555 Utilities in the Simulink block library. You see the following options:

- **Target Preferences** — Opens the Target Preferences dialog box. See "Setting Target Preferences for MPC5xx" on page 49-11.
- **Run Simple MPC5xx Test Application (via BDM)** — Downloads and runs a simple test application that uses your hardware to make an LED blink. See "Run Test Program" on page 49-15.

- **Install MPC5xx Bootcode** — Installs the appropriate boot code on your target processor. See "Download Boot Code to Flash Memory" on page 49-16.
- **Inspect the MPC5xx Hardware (via BDM)** — Opens your debugger so you can inspect the hardware.
- **Debug RAM Based Application (via BDM)** — Downloads and then enables you to debug a RAM application in `.elf` format.
- **Debug FLASH Based Application Already in FLASH (via BDM)** — Enables you to debug an application (in `.elf` format) already in FLASH.
- **Download RAM / FLASH Based Application (via CAN / Serial)** — Launches the Download Control Panel, for downloading applications in `.s19` format to your hardware. See "Tutorial: Creating a New Application" on page 49-26, and "Downloading Application Code" on page 49-42.

- **Download FLASH Based Application (via BDM and CAN / Serial)** — Allows you to use a BDM and the Download Control Panel to download an application in `.s19` format to FLASH memory. See "Downloading Application Code" on page 49-42.
- **Initialize visionPROBE for Selected Target Board (WindRiver Only)** — If you are using a visionPROBE, run this option to initialize the device, after setting target preferences (and again if you change target processor). See "Initialize visionPROBE" on page 49-128.
- **Rebuild the MPC5xx Driver Library** — Recompiles the MPC5xx Driver libraries. See "Boot Code Parameters for CAN Download" on page 49-52.

Data Type Support and Scaling for Device Driver Blocks

The following table summarizes the input and output data types supported by the device driver blocks in the Embedded Targets library and the scaling applied to block inputs and outputs.

I/O Data Types and Scaling for MPC5xx Device Driver Blocks

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
MPC5xx MIOS Digital In			`Boolean`	0 or 1 only
MPC5xx MIOS Digital Out	Any Simulink supported data type	logic 1 if input > 0, logic 0 if input <= 0		
MPC5xx MIOS Digital Out (MPWMSM)	Any Simulink supported data type	logic 1 if input > 0, logic 0 if input <= 0		

I/O Data Types and Scaling for MPC5xx Device Driver Blocks (Continued)

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
MPC5xx MIOS Pulse Width Modulation Out	`double` or `single`	0 to 1		
MPC5xx MIOS Waveform Measurement			`double` or `single`	Seconds
MPC5xx QADC Analog In			`uint16` or `int16` (defined by **Justification** parameter)	(defined by **Justification** parameter)
MPC5xx QADC Digital In			`Boolean`	0 or 1 only
MPC5xx TouCAN Receive			`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`	N/A
MPC5xx TouCAN Transmit	`CAN_MESSAGE_STANDARD` or `CAN_MESSAGE_EXTENDED`	N/A		
MPC5xx TouCAN Warnings			`Boolean`	N/A
MPC5xx TouCAN Error Count			`uint8`	N/A
MPC5xx TouCAN Fault Confinement State			`uint16`	N/A
MPC5xx TPU3 Digital In			`Boolean`	0 or 1 only
MPC5xx TPU3 Digital Out	Any Simulink supported data type	Logic 1 if input > 0, logic 0 if input <= 0		

I/O Data Types and Scaling for MPC5xx Device Driver Blocks (Continued)

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
MPC5xx TPU3 Fast Quadrature Decode	Fast Mode input `Boolean`		`uint16`	N/A
MPC5xx TPU3 New Input Capture/Input Transition Counter			`uint16`	N/A

MPC5xx TPU3 Programmable Time Accumulator			Time Accumulation
			`uint32`
			Period Count
			`uint8`
				N/A

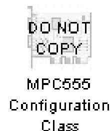
MPC5xx TPU3 Pulse Width Modulation Out	Duty cycle input (top if 2 inputs): `double` or `single`	0 to 1		
	Pulse period register input — `uint16`	Saturated to be in the range 0 to 32768		

I/O Data Types and Scaling for MPC5xx Device Driver Blocks (Continued)

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
MPC5xx Serial Transmit	Data: `uint8` (vector or scalar) Number of bytes: `uint32` (scalar)	N/A	Number of bytes: `uint32`	0-16 (for SCI1); 0 or 1 (for SCI2)
MPC5xx Serial Receive	Number of bytes: `uint32` Reset: `Boolean`	N/A 0 or 1	Data: `uint8` Actual number of bytes: `uint32` Framing and parity error: `Boolean` Overrun flag: `Boolean`	N/A N/A 0 or 1 0 or 1

Configuration Class Blocks

Each sublibrary of the Embedded Targets library contains a *configuration class block* that has an icon similar to the one shown in this figure.



Configuration class blocks exist only to provide information to other blocks. *Do not copy these objects into a model.*. If you do you see an error dialog box to warn you. This causes build failures.

Generating Stand-Alone Real-Time Applications

This section includes the following topics:

In this section...

"Overview" on page 49-24

"Tutorial: Creating a New Application" on page 49-26

"Downloading Boot and Application Code" on page 49-39

"Parameter Tuning and Signal Logging" on page 49-53

"HTML Code Profile (RAM/ROM) Report" on page 49-67

"Execution Profiling" on page 49-68

"Summary of the Real-Time Target" on page 49-76

"Performance Tips" on page 49-79

Overview

- "Generating Real-Time Applications" on page 49-24
- "Deploying Generated Code" on page 49-25

Generating Real-Time Applications

This section describes how to generate a stand-alone real-time application for the MPC555. The components required to generate stand-alone code are

- The Embedded Coder real-time target features
- The MPC555 Resource Configuration object provided in the Embedded Targets library
- I/O driver blocks provided in the Embedded Targets library
- Utilities for downloading generated code to the target hardware

Using these together with your toolchain, you can build a complete application. You do not need to manually write any C code to integrate the generated code into a final application.

The tutorial "Tutorial: Creating a New Application" on page 49-26 uses two blocks from the Embedded Targets library. For complete information on the Embedded Targets library blocks, see "Freescale MPC5xx".

Before reading this section and using the Embedded Targets library, you should have at least a basic understanding of the architecture of the MPC555. To learn about the MPC555, study the *MPC555 Users Manual*. We recommend that you read the introduction to the processor and familiarize yourself with all the major subsystems of the MPC555. You can find this document at the following URL:

```
http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf
```

Deploying Generated Code

You can load a generated program into the MPC555 flash memory for permanent deployment. You can also load your code into external RAM (if available on your development hardware).

Alternatively, you can use the automatic code generation process for rapid prototyping and investigate a range of different design alternatives before making a deployment decision.

Your generated program can run on any Electronic Control Unit (ECU) that is based on the MPC5xx processor. Your application can use any of the supported MPC5xx on-chip I/O devices. We provide driver blocks for the MPC5xx's MIOS, TPU, QADC and TouCAN modules, providing you with drivers for the on-chip analog input, digital I/O, PWM, serial and CAN devices.

See "Freescale MPC5xx" for further information on the device driver blocks in the Embedded Targets library.

In addition to on-chip I/O resources, an ECU typically provides additional I/O devices. If you want to access such custom I/O devices, you must write device drivers and integrate them with the automatically generated code.

Once the application has been programmed into memory on the target system, you may need to monitor signals or tune parameters. The coder product supports signal monitoring and parameter tuning via Simulink external mode or a third party calibration tool. In both cases you must

include a CAN Calibration Protocol (CCP) block in your model. The CAN Calibration Protocol block implementation of CCP has been tested against CANape from Vector-Informatik and ATI Vision. See “Parameter Tuning and Signal Logging” on page 49-53 and MPC5xx CAN Calibration Protocol for further information.

Tutorial: Creating a New Application

- “Tutorial Overview” on page 49-26
- “Before You Begin” on page 49-27
- “The Example Model” on page 49-28
- “Generating Code” on page 49-31
- “Downloading the Application to RAM via Serial or CAN” on page 49-33
- “Downloading the Application to RAM via BDM” on page 49-37

Tutorial Overview

In this tutorial, you build a stand-alone real-time application from a model incorporating blocks from the Embedded Targets library. We assume that you are already familiar with the Simulink product and with the code generation and build process.

In the following sections, you will

- Configure the model
- Generate code from a subsystem
- Download code by one of the following methods:
  - Download to target RAM via a serial connection, using the **Download Control Panel** utility (provided with the coder product)
  - Download to target RAM via a CAN connection, using the **Download Control Panel** utility
  - Download to target RAM via a BDM connection
- Execute the code on the target

After you complete this tutorial, you may want to learn how to deploy generated code into the MPC555 flash memory. See "Downloading Boot and Application Code" on page 49-39 for that information.

Before You Begin

This tutorial requires the following specific hardware and software in addition to the coder product:

- Phytec phyCORE-MPC555 development board

  The tutorial model utilizes two LEDs on the phyCORE-MPC555 board. These LEDs are connected to pins `MPIO32B0` and `MPIO32B1` on the MPC555 MIOS digital output pins. If you are using a different development board, you may be able to obtain the same functionality by making similar connections.

- A supported toolchain for compiling and debugging. Currently supported toolchains are

  - Wind River Compiler and Wind River Systems SingleStep from Wind River Systems
  - CodeWarrior from Freescale

  See "Setting Up Your Toolchain" on page 49-124 for details.

- Hardware to enable downloading:

  - If you want to download generated code to the target board over serial you will need a serial cable to connect your host PC to the target board.
  - If you want to download over BDM you will need a BDM device.
  - If you want to download via CAN, you will need a supported CAN card and drivers from Vector-Informatik. See "CAN Hardware and Drivers" on page 49-139.

Configuring Target Preferences and Boot Code.

- Make sure that your target preferences are set correctly for your development tools. See "Setting Target Preferences for MPC5xx" on page 49-11.

- Once your target preferences are set for your toolchain you must download bootcode to the target before you can work through this tutorial. See “Download Boot Code to Flash Memory” on page 49-16.

The Example Model

In this tutorial we will use a simple example model, `mpc555rt_led`.

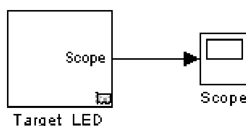
This folder is on the default MATLAB path. The path *`matlabroot`* is the location where MATLAB is installed.

1 Open the model.

```
mpc555rt_led
```

2 Save a local copy to your working folder. We will work with this copy throughout this exercise.

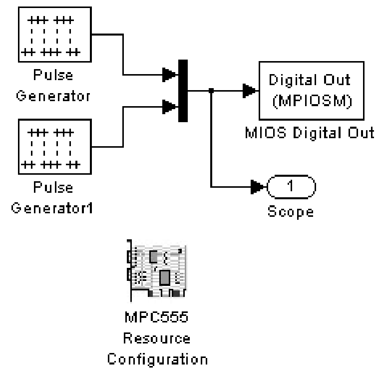
mpc555rt_led_demo Model, Root Level shows the example model at the root level. We will only use this level in simulation.



mpc555rt_led_demo Model, Root Level

3 Double-click on the `Target_LED` subsystem block.

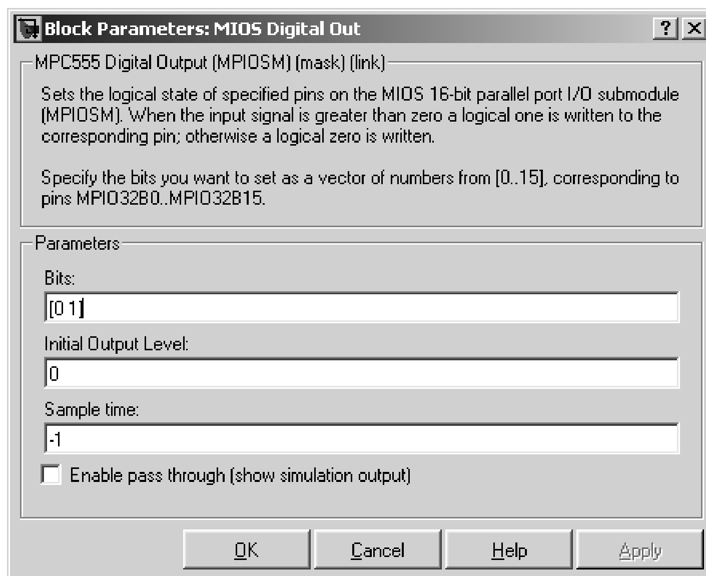
Target_LED Subsystem shows the `Target_LED` subsystem, from which we will generate code.



Target_LED Subsystem

In the `Target_LED` subsystem, two square wave signals are multiplexed and routed to the MIOS Digital Out block. The MIOS Digital Out block accepts a vector of numbers representing pins 0-15 on the MIOS 16-bit Parallel Port I/O Submodule (MPIOSM) on the MPC555. As the square wave signals oscillate between 0 and 1, the MIOS Digital Out block writes corresponding logic values to the appropriate pin on the port.

This figure shows the parameters of the MIOS Digital Out block.



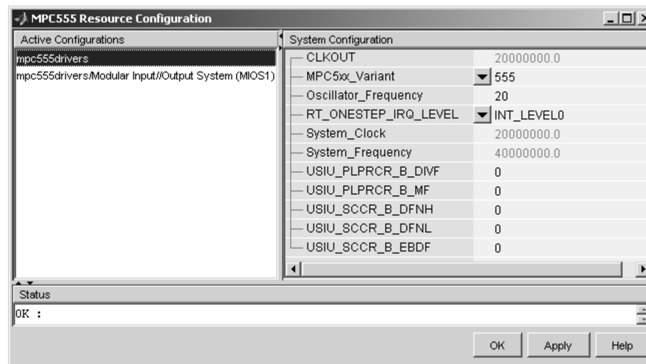
The **Bits** field is set to the vector `[0 1]`. The block maps this vector to the MPC555 MIOS digital output pins `MPIO32B0` and `MPIO32B1`. When the application runs, it will send a pulse signal to these output pins. On the phyCORE-MPC555 board, these signals are connected to two of the LEDs, which will switch on and off at the frequency set in the respective pulse generator blocks.

In addition to the Pulse Generator, Mux, MIOS Digital Out, and Output blocks, the `Target_LED` subsystem contains a MPC555 Resource Configuration object. When building a model with driver blocks from the Embedded Targets library, you must always place a MPC555 Resource Configuration object into the model (or the subsystem from which you want to generate code) first.

The purpose of the MPC555 Resource Configuration object is to provide information to other blocks in the model. Unlike conventional blocks, the MPC555 Resource Configuration object is not connected to other blocks via input or output ports. Instead, driver blocks (such as the MIOS Digital Out block in the example model) query the MPC555 Resource Configuration object for required information.

For example, a driver block may need to find the system clock speed that is configured in the MPC555 Resource Configuration object. The MPC555 has a number of clocked subsystems; to generate correct code, driver blocks need to know the speeds at which these clock busses will run.

The MPC555 Resource Configuration window lets you examine and edit the MPC555 Resource Configuration settings. To open the MPC555 Resource Configuration window, double-click on the MPC555 Resource Configuration icon. This figure shows the **MPC555 Resource Configuration** window for the `Target_LED` subsystem.



In this tutorial, we will use the default MPC555 Resource Configuration settings. Observe, but do not change, the parameters in the MPC555 Resource Configuration window. To learn more about the MPC555 Resource Configuration object, see MPC5xx MPC555 Resource Configuration.

Close the **MPC555 Resource Configuration** window before proceeding.

The next step in this tutorial is generating code.

Generating Code

We will now look at settings and then generate application code:

1. Select **Simulation > Configuration Parameters**. The **Configuration Parameters** dialog opens.

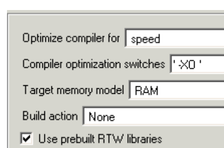
2. Select **Code Generation** in the tree.

3 Notice the **System target file** for real-time deployment is `mpc555rt.tlc`.

   To see how to change from real-time deployment to processor-in-the-loop or algorithm export, click on the **Browse** button to open the **System Target File Browser**. Click **Cancel** to keep the default real-time setting and return to the **Code Generation** pane.

4 Select `ET MPC5xx real-time options (1)` in the tree. The `RAM` option should be selected from the **Target memory model** menu. This option directs the build process to generate a code file suitable for downloading and execution in RAM. The files for both RAM and flash are in Motorola S-record format.

   Leave the options set to their defaults. The code generation options should appear as shown below (though optimization switches settings vary between toolchains).

5 You are now ready to build the application. Do this by right-clicking on the `Target_LED` subsystem and selecting **Code Generation > Build Subsystem**. Then click the **Build** button in the following dialog.

6 On successful completion of the build process, two files are created in the working folder:

   a `Target_LED_ram.s19`: This file is for serial or CAN download. It is code only, without symbols, suitable for execution on the target system.

   b `Target_LED_ram.elf`: This file is for BDM download.

      If `debug` is selected in the compiler optimization settings, the `elf` file will contain debugging symbols as well as code. These symbols are suitable for use with a symbolic debugger such as Wind River Systems SingleStep or Freescale CodeWarrior. The default optimization setting is `speed`, so no symbols are included. Symbols are only generated for a `debug` build. See “Compiler Optimization Switches” on page 49-14.

You can download to RAM:

- Via Serial or CAN, using the **Download Control Panel** utility (with Vector-Informatik hardware if you are using CAN), as described in "Downloading the Application to RAM via Serial or CAN" on page 49-33.
- Via the BDM port, as described in "Downloading the Application to RAM via BDM" on page 49-37.

Downloading the Application to RAM via Serial or CAN

The **Download Control Panel** utility can be used to download application code to MPC555 RAM or to MPC555 flash memory.

In this section, you will use the **Download Control Panel** utility to download the generated `Target_LED_ram.s19` file to RAM on the target system. The `s19` file is for download over serial or CAN.

`Target_LED_ram.elf` is for BDM download, as described in the next section, "Downloading the Application to RAM via BDM" on page 49-37. Recall you can perform a debug build to include debugging symbols in the `elf` file.

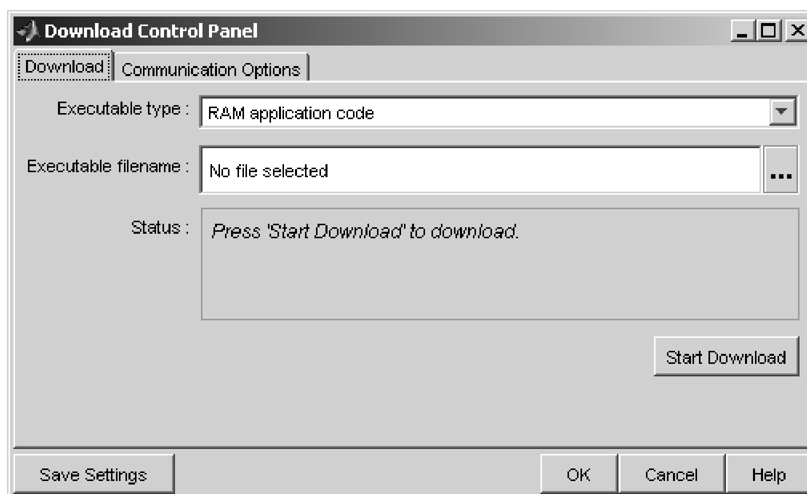
Do the following before you begin:

- If you are using serial, make sure you have connected a serial port on your PC to serial port 1 (RS232-1) on the target hardware.
- If you are using CAN, make sure that your Vector-Informatik CAN card and drivers are installed and configured properly. See "CAN Hardware and Drivers" on page 49-139. Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware.
- Make sure that you have set up your toolchain as described in "Setting Up Your Toolchain" on page 49-124, and downloaded boot code to the flash memory of the MPC555 as described in "Download Boot Code to Flash Memory" on page 49-16.
- Make sure that nothing is connected to the BDM port of your development board.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in "Phytec MPC555 Jumper Settings" on page 49-134.

- Cycle the power (or perform a hard reset) on your development board, to clear the RAM.

To download the generated `Target_LED_ram.s19` file to RAM:

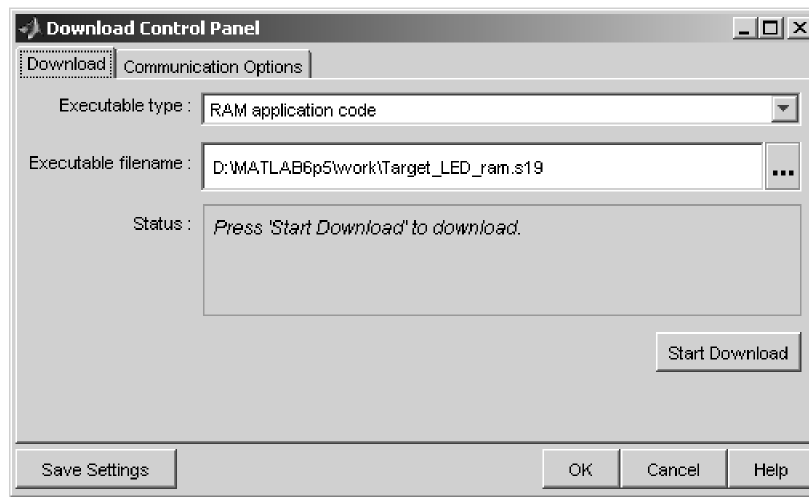
1. Start the **Download Control Panel** in one of the following ways:
   - Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window. Select `Download RAM / FLASH Based Application (via CAN / Serial)` from the drop-down list, and click **OK**.
   - Enter `embedded_target_download` in the Command Window.
   - You can also open the **Download Control Panel** automatically at the end of the build process. Before you start the build, you can select **Launch Download Control Panel** from the **Build action** options on the `ET MPC5xx real-time options (1)` pane of the **Configuration Parameters** dialog.
2. After using any of these three options, the **Download Control Panel** dialog opens.



   Note `RAM application code` is automatically selected in the **Executable type** menu. You can use exactly the same process to download application

code to flash memory by changing this option to `Flash application code`. Note that you need to build a *`model`*`_flash.s19` file in order to use this option, as described in "Downloading Application Code to Flash Memory via Serial or CAN" on page 49-44. For this exercise leave the `RAM` option selected.

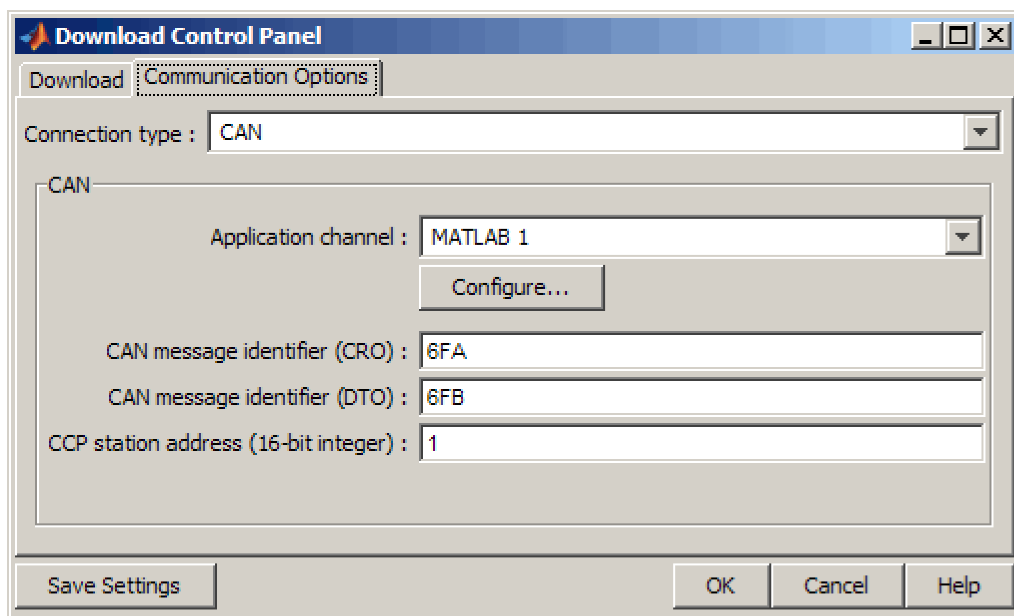
3 Enter the name of the file to be downloaded into the **Filename** field, in this case, `Target_LED_ram.s19`. Alternatively, you can use the browse button (right of the edit box) to navigate to the desired file. The **Download Control Panel** should now appear as shown in this figure.



4 Click on the **Communications Options** tab.

- If you are using serial, select `Serial` from the **Connection Type** drop-down menu. Select the appropriate host PC connection port from `COM1` to `COM8`. You can save your preferences by clicking the **Save Preferences** button.
- If you are using CAN, select `CAN` from the **Connection Type** drop-down menu. Click **Configure** to select an appropriate card and port from the **CAN hardware** drop-down menu. You must create a MATLAB application channel to assign to a CAN channel. See "CAN Hardware and Drivers" on page 49-139 for instructions. The default settings for the other parameters are appropriate for most cases. You can save your

preferences by clicking the **Save Settings** button. The following figure shows the **Communications Options**.



5 Click the **Download** tab. Then click the **Start Download** button.

When you click **Start**, the **Download Control Panel**'s **Status** box changes to read `Press reset or power-cycle your development board to start download`.

6 Press the Reset button on your PhyCORE-MPC555 board (or cycle the power). The **Download Control Panel** changes its **Status** box to inform you that the connection is OK.

Downloading commences, and the **Start** button caption changes to **Stop**.

7 While downloading proceeds, progress messages are displayed in the **Download Control Panel**. After the download, the **Stop** button caption changes back to **Start**.

If the download does not succeed, reset your development board and return to step 5.

8 Close the **Download Control Panel** dialog box.

9 A few seconds after a successful download, the boot code transfers control to the application program. At this point, you should see two LEDs (red and green) blinking on the target board. This indicates that the program is operating correctly.

Note that you can monitor the progress of a CAN download using a program such as CANalyzer. Alternatively, you can use the btest32 utility supplied with the Vector Informatik driver software. You can invoke the btest32 utility from the PC command prompt. The following example runs btest32 with a bit rate of 500000 (500 kbaud):

```
btest32 500000
```

Downloading the Application to RAM via BDM

You can choose to automatically download to the target over BDM on completion of the build process. Follow these steps to generate, download and execute the `Target_LED_ram.elf` file in RAM on the target system. `Target_LED_ram.elf` can contain both code and symbols for use with the debugger if you perform a debug build. You will not perform a debug build in this tutorial, so the file will contain code only.

If you want to download application code to MPC555 flash you can use serial or CAN. The download process is exactly the same as described in "Downloading the Application to RAM via Serial or CAN" on page 49-33, except you change the **Download** option from `RAM` to `Flash`. Note that you also need to generate a *`model`*`_flash.s19` file to download to flash memory, as described in "Downloading Application Code to Flash Memory via Serial or CAN" on page 49-44. If you want to download the application to flash memory over BDM manually using your own tools, then the file you need to download is the S-record file *`model`*`_flash.s19`.

Do the following before you begin:

- Make sure that you have downloaded boot code to the flash memory of the MPC555. See "Download Boot Code to Flash Memory" on page 49-16.

- Connect the BDM port of your development board to parallel port LPT1 of your host PC (or the port specified for your toolchain if different, see "Setting Up Your Toolchain" on page 49-124).
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in "Phytec MPC555 Jumper Settings" on page 49-134.

To generate and download the `Target_LED_ram.elf` file to RAM over BDM,

1. Select **Simulation > Configuration Parameters**.

   The **Configuration Parameters** dialog appears.

2. Under **Code Generation** in the tree, click to select `ET MPC5xx real-time options (1)`.

3. Select `Run_via_BDM` or `Debug_via_BDM` from the **Build action** drop-down menu.

4. Ensure the **Target memory model** selected is `RAM` (not `FLASH`).

   Notice the default **Optimize compiler for** setting is `speed`. If you change this setting to `debug`, the generated `elf` file will contain both code and symbols for use with a symbolic debugger. See "Compiler Optimization Switches" on page 49-14 for more information on these settings. For this tutorial, leave this setting at the default.

5. Right click on the `Target_LED` subsystem and select **Code Generation > Build Subsystem**.

You will see progress messages in the MATLAB Command Window as code is generated. Your debugger will be automatically started and will download the code to the target.

Also available is the **Start** menu option **Debug RAM-Based Application (Via BDM)**. Use this option to select a `*.elf` file and debug over BDM as described above. You can use this option to debug a model you have already built without having to go through the build process again.

Downloading Boot and Application Code

- “RAM vs. Flash Memory” on page 49-39
- “Overview of Memory Organization and the Boot Process” on page 49-40
- “Downloading Application Code” on page 49-42
- “Using the Download Control Panel as a Standalone Application” on page 49-47
- “Downloading Boot or Application Code via CAN Without Manual CPU Reset” on page 49-49
- “Rebuilding the Boot Code and Device Driver Libraries” on page 49-51
- “Running Applications with a Debugger” on page 49-53

RAM vs. Flash Memory

The coder product creates a file containing the application executable code that must be programmed onto the MPC555. It can also write a file including symbolic information suitable for use with a debugger. The files are written to your working folder.

The format of the code and symbol files is the same for both RAM and flash memory targets, suitable for downloading into RAM or on-chip flash memory. The naming convention for these files is

- *model*_flash.s19 or *model*_ram.s19 (for serial and CAN download)
- *model*_flash.elf or *model*_ram.elf (for BDM download, can contain debugging symbols).

You can download code to RAM or flash memory via serial or CAN download, or via the MPC555’s BDM port.

There are advantages and disadvantages to each memory model.

Loading the application code into RAM is faster than loading it into flash memory. In addition, by using RAM you can avoid using up the programming cycles of the flash memory; this lengthens the usable lifetime of the flash memory. Running the application from RAM is a good option for initial testing of the application.

Note The MPC5xx flash memory has a limited lifetime, which is shortened each time the flash memory is programmed. To extend product life, Freescale recommends using flash programming only when necessary.

To program applications into RAM, your target hardware must have additional RAM external to the MPC555 on-chip RAM. The coder product does not support downloading of code to MPC5xx on-chip RAM, because the MPC555 has only 26K of on-chip RAM and the MPC565 has 36K.

For final deployment, or to load code onto a test board for use at a test site, you will generally want to program your code into the nonvolatile flash memory. 416K of flash memory is available for application code (992K on the MPC565). Code programmed into flash memory is persistent and restarts when the board is powered on.

To download code to flash memory, you must first load a binary boot code file into the flash memory. The coder product provides the boot code file. You must load the boot code into flash memory in order to run application code. The boot code is always required even for RAM applications.

To understand the download process, it is first necessary to review the memory organization on the MPC555 and the operation of the boot code. This is described in the next section, "Overview of Memory Organization and the Boot Process" on page 49-40.

- If you just want to know how to download application code, you can jump ahead to the section "Downloading Application Code" on page 49-42.
- If you want to know how to download boot code, see the Getting Started section "Download Boot Code to Flash Memory" on page 49-16.

Overview of Memory Organization and the Boot Process

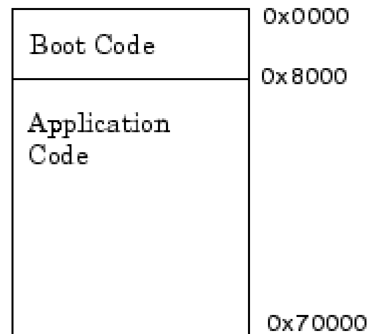
Purpose of Flash Memory Boot Code. When reading this section, you may want to refer to the internal memory map of the MPC555 in section 1.3 of the *MPC555 User's Guide*. You can find this document at the following URL:

```
http://www.freescale.com/files/microcontrollers/doc/user_guide/MPC555UM.pdf
```

To run generated code from the RAM or flash memory, you must load the first 32K flash sector with boot code. The primary purpose of the boot code is to load and start application code when the board is powered on or reset. The boot code also acts as a download agent that downloads generated code into RAM or flash memory via CAN or serial.

The boot code manages the exception handling for the MPC555. Applications don't directly handle exceptions but receive them from the boot code. If the boot code is not installed, then applications will not work correctly.

Memory Organization. The MPC555 has a total of 448K of on-chip flash memory (1024K on the MPC565). This memory is organized into banks of 32K each. The first bank is always used to store the boot code and the remaining 416K is available for application code (992K on the MPC565). When using the coder product, the on-chip flash memory is located at absolute address `0x0000` in the MPC555 address space.



Organization of Flash Memory

To run a stand-alone application on the MPC555, it is first necessary to program the boot code into the first bank of flash memory.

The Boot Process. The boot code is executed following power on or reset (unless a probe is connected to the BDM port). Normally, the boot code performs basic hardware initialization and then branches to the application code. Once the application code is running, there is no way to return to the boot code except by performing a reset.

One of the important functions of the boot code is to serve as agent that allows program code to be downloaded over CAN or serial. There are two methods of initiating a program download over CAN or serial:

- The default method for initiating a program download is to send a special serial or CAN message during a short window of time while the boot code is executing. In the supplied boot code, this window is set to 40ms. If this special message is received during the window while the boot code is executing, a program download sequence commences and a new application can be programmed into RAM or flash memory. See "Downloading Application Code to Flash Memory via Serial or CAN" on page 49-44 for details.
- Alternatively, it is possible to commence a program download over CAN while application code is running on the target. To initiate a download over CAN, you must include a special block in your Simulink model. This block is the CAN Calibration Protocol block. See "Downloading Boot or Application Code via CAN Without Manual CPU Reset" on page 49-49 for details.

The bootcode download process erases the non-volatile flash memory (including the shadow area) before writing the new bootcode, and the previous configuration word is removed. The bootcode download process does not write a replacement configuration word to the shadow flash. Typically, users of the coder product do not use a Hard Reset Configuration Word that is stored in non-volatile memory (the shadow flash). Instead, the development board is generally assumed to source the configuration word from the data bus.

If you want to use a custom configuration word, you must manually program the shadow flash to an appropriate value for the system. This would only need to be done along with the irregular updates to the bootcode.

Downloading Application Code

The following sections describe how to download generated image files and run generated code on the target hardware. They also describe how to download to RAM and to flash memory, via either serial, CAN, or the BDM port.

Downloading the Application Code to RAM. To download application code to RAM, you must generate a code file in Motorola S-Record format, which is suitable for downloading and execution in RAM. To do this, select the `RAM` option from the **Target memory model** menu in the `ET MPC5xx real-time options (1)` category of the **Configuration Parameters** dialog. The build process creates two files in the working folder:

- Files created:
  - *`model`*`_ram.s19`: For serial or CAN download. Code only, without symbols, suitable for execution on the target system.
  - *`model`*`_ram.elf`: For BDM download. Can also contain symbols if you perform a debug build, suitable for use with a symbolic debugger such as Wind River Systems SingleStep.

- You can download to RAM via serial or CAN, using the **Download Control Panel** utility (with Vector-Informatik CAN hardware if applicable), as described in "Downloading the Application to RAM via Serial or CAN" on page 49-33.
- You can also download to RAM via BDM, as described in "Downloading the Application to RAM via BDM" on page 49-37.

Downloading the Application Code to Flash Memory. To download application code to flash memory, you must generate a code file which is suitable for downloading and execution in flash memory. To do this, select the `FLASH` option from the **Target memory model** menu in the `ET MPC5xx real-time options (1)` category of the **Configuration Parameters** dialog. The build process creates the file *`model`*`_flash.s19` which contains an image of the executable code, in the working folder.

You can download the file to flash memory via serial or CAN, using the **Download Control Panel** utility (with Vector-Informatik hardware if using CAN), as described in the following section. Note you can also use the **Start** menu option to use a BDM (and serial or CAN) to download application code to flash memory. If you want to download the application to flash memory over BDM manually using your own tools, then the file you need to download is the S-Record file *`model`*`_flash.s19`.

Downloading Application Code to Flash Memory via Serial or CAN. You can use the **Download Control Panel** to download generated application code to the MPC555 flash memory. Note that except for changing the **Download** option from `RAM` to `Flash`, the process is the same as downloading to RAM.

Do the following before you begin:

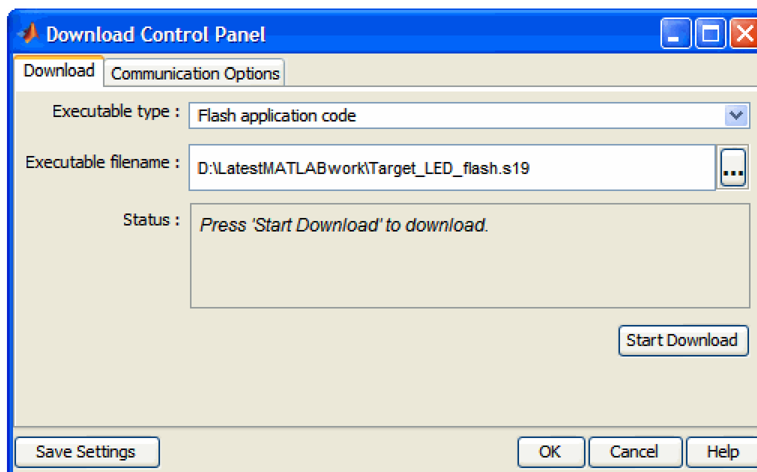
- If you are using serial, make sure you have connected the serial port on your PC to serial port 1 (RS232-1) on the target hardware.
- If you are using CAN, make sure that your Vector-Informatik CAN card and drivers are installed, and are configured properly. See "CAN Hardware and Drivers" on page 49-139. Make sure that the desired CAN port on the PC card is connected to the CAN A port on the target hardware.

- Make sure that you have set up your toolchain and downloaded boot code to the flash memory of the MPC555, as described in "Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation" on page 49-10.
- Make sure that nothing is connected to the BDM port of your development board.
- Make sure that the jumpers on the phyCORE-MPC555 board are set as described in "Phytec MPC555 Jumper Settings" on page 49-134.

To download the generated `*model*_flash.s19` file to flash:

1. Open the **Download Control Panel** in one of the following ways:
   - Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window. Select `Download RAM / FLASH Based Application (via CAN / Serial)` from the drop-down list, and click **OK**.
   - Type `embedded_target_download` at the MATLAB command prompt.
   - You can also open the **Download Control Panel** automatically at the end of the build process. Before you start the build process, you can select **Launch Download Control Panel** from the **Build action** options on the `ET MPC5xx real-time options (1)` tab of the Model Explorer.

   After using any of these three options, the **Download Control Panel** window opens.
2. Select `Flash application code` from the **Executable type** menu.
3. Enter the name of the file to be downloaded into the **Executable filename** field. Alternatively, you can use the browse button to navigate to the desired file. Remember the *model*`_flash.s19` files are for serial or CAN download to flash. The **Download Control Panel** should now appear as shown in this figure.



4 Click on the **Communications Options** tab. If you have not saved your preferences already, select `Serial` or `CAN` from the **Connection Type** drop-down menu. If necessary, select an appropriate card/port. The default settings for the other parameters are appropriate for the default download process. You can save your preferences by clicking the **Save Preferences** button. The **Communications Options** configured for a Vector-Informatik CAN-AC2-PCI card, channel 1, are shown in the section "Downloading the Application to RAM via Serial or CAN" on page 49-33.

5 The next step is to download code. Click the **Download** tab, and then click the **Start** button.

- If there is an application currently running on the target that contains a CAN Calibration Protocol (CCP) kernel, the download proceeds automatically. For more details see "Downloading Boot or Application Code via CAN Without Manual CPU Reset" on page 49-49.
- If the CCP condition is not met, you must immediately press the reset button on your PhyCORE-MPC555 board after clicking the **Start** button. You will see a message prompt in the **Status** box: `Press reset or power-cycle your development board to start download.`

  When you press the Reset button on your PhyCORE-MPC555 board (or cycle the power), the **Download Control Panel** changes its **Status** box to read `CCP Connection OK. Please wait till completion or press Stop to terminate the download.`

Downloading commences, and the **Start** button caption changes to **Stop**. While downloading proceeds, progress messages are displayed in the **Download Control Panel**. A successful download ends with an information dialog and the **Stop** button caption changes back to **Start**.

6. If the download does not succeed, reset the board and return to step 5.

   You can monitor the progress of the flash download over CAN by using a program such as CANalyzer. Alternatively, you can use the btest32 utility supplied with the Vector Informatik driver software. You can invoke the btest32 utility from the PC command prompt. The following example runs btest32 with a bit rate of 500000 (500kbaud):

   ```
   btest32 500000
   ```

7. Close the **Download Control Panel** window.

Once the download process is complete, the application starts running immediately on the target hardware.

Using the Download Control Panel as a Standalone Application

You can use the Download Control Panel utility as a standalone application, separate from MATLAB.

To install the utility, complete the following steps:

1. Using the MATLAB command prompt, download the utility to a local folder. For example:

   ```
   embedded_target_download('install', 'c:\dcp_utility')
   ```

   If the folder does not exist, this command creates the folder.

2. Using the operating system command prompt, change directories to the folder. For example:

   ```
   cd c:\dcp_utility
   ```

3. Using the operating system command prompt, display the Download Control Panel prerequisites by entering:

```
embedded_target_download.bat -help
```

A static list of software environment prerequisites appears. For example:

```
-----------------------------------------------------

Embedded Target Download (Standalone) Help


DOWNLOAD_WORK_DIR:

The location used to look for application files to
download. You can edit this batch file
(embedded_target_download.bat) to set your own location.

Requirements:

Java Virtual Machine (JVM):
    This utility is written using Java, and requires a JVM
    in order to run.   Please install a Java Runtime Environment
    on your system and ensure that the path to the Java
    Interpreter is added to the system path, so that java.exe can
    be executed from the command line. (http://java.sun.com)

For downloading over CAN:

Vector-Informatik CAN Programming DLL (vcand32.dll):
    This file is available from Vector-Informatik, and
    must be somewhere on the system path (includes current dir)
    (http://www.vector-informatik.de/english)

Vector-Informatik CAN Drivers:
    Hardware drivers for your CAN hardware must be installed on the system.
    These drivers are available from Vector-Informatik
    (http://www.vector-informatik.de/english)

---------------------------------------------------------------------------
```

4 Address any unsatisfied prerequisites before starting the utility.

Note To set a default location for the utility to look for application files, edit the `DOWNLOAD_WORK_DIR` variable in embedded_target_download.bat.

To start Download Control Panel as a stand-alone application, complete the following steps using the operating system command prompt:

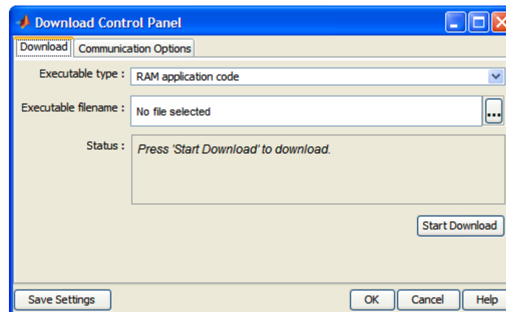
1. Change directories to the installation folder. For example:

   ```
   cd c:\dcp_utility
   ```

2. Start the utility:

   ```
   embedded_target_download.bat
   ```

   This command displays the Download Control Panel utility.



To move the Download Control Panel utility to another location, such as another computer, copy and paste the installation folder to that new location. Verify that the new environment meets the Download Control Panel prerequisites.

Downloading Boot or Application Code via CAN Without Manual CPU Reset

The default method for download over CAN requires that the target processor be manually reset in order for the download process to begin. This requirement may be problematic if the target hardware is not physically accessible or if it cannot be individually reset or powered down/up.

It is possible to remove this requirement for manual reset if a suitably prepared application is already running on the target. To do this, include a CAN Calibration Protocol block within the model (see MPC5xx CAN Calibration Protocol).

When the currently running application includes the CAN Calibration Protocol block, the download process begins when you click on the **Start** button of the **Download Control Panel**; it is not necessary to manually reset the target hardware to initiate the download. A reset of the processor is triggered by a CCP Program Prepare message. After the Program Prepare message is received at the target, there will be a short delay until the processor resets and continues the download process by transmitting a response to the Program Prepare message.

The length of the delay will be the watchdog timeout period of the application. By default, for a 20MHz application, this will be approximately 7 seconds; for a 40MHz application, this will be approximately 3 seconds.

It is possible to explicitly set the timeout period of the watchdog timer, by using the Watchdog block in the MPC555 device driver library. See MPC5xx Watchdog.

The **Download Control Panel** is configured to allow a maximum delay of 10 seconds between sending the Program Prepare message and receiving a response from the target. If this delay is exceeded, an error will be reported by the download tool.

When using the CAN Calibration Protocol block, you must specify

- CAN message identifier for Command Receive Objects
- CAN message identifier for Data Transmit Objects
- CAN Calibration Protocol Station Address

Note that the values specified are permitted to differ from the default values for these parameters that are programmed in the boot code. When performing the download procedure using the **Download Control Panel**, you must ensure that the parameters specified on the **Communications Options** tab match those specified in the currently running application.

For an example of how to use the CAN Calibration Protocol block for signal monitoring, parameter tuning and automatic download, see the demo model `mpc555rt_ccp`.

Rebuilding the Boot Code and Device Driver Libraries

You must rebuild the libraries to enable execution profiling for device driver interrupt service routines. See "Enabling Execution Profiling for Device Driver Interrupt Service Routines" on page 49-75 for instructions in that case.

You cannot change the default boot code parameter values except by modifying and recompiling the boot code. If it is absolutely necessary to do this, you can recompile the boot code as follows:

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window. Select `Rebuild the MPC5xx Driver Library` from the drop-down list, and click **OK**.

   The Build Driver Libraries dialog box opens.

2. Select the compiler optimization setting you want to use for the build (from `speed`, `size`, `debug`, or `clean`).

   - See "Compiler Optimization Switches" on page 49-14 for more information on the `speed`, `size` and `debug` settings, which are compiler-specific. You can edit these settings in the **Target Preferences** dialog.
   - The `clean` option deletes all object files. Note that to ensure a rebuild of all files you should run a `clean` build followed by a build using your required optimization setting. Otherwise only files which have changed since last library build will be rebuilt.

   The coder product automatically recompiles the code, using your settings in target preferences.

Note You should not make changes to the boot code without fully understanding the effect of your changes. Note also that the boot code may be changed without notice in future releases of this product.

If a required prebuilt library is not found during the build process, then you see a dialog box with instructions to rebuild the missing library. For example, a prebuilt copy of the Signal Processing Library is not installed with the product.

It is preferable to rebuild via the **Start** menu rather than using the commands suggested in the dialog box, because any rebuild done via the dialog is dependent on the options selected in the **Code Generation> Interface > Software Environment > Support** options, and any library created is based on these settings. You then need to rebuild your model to complete the build process.

Boot Code Parameters for CAN Download. The boot code parameters for download over CAN determine

- CAN bit rate
- CAN message identifier for Command Receive Objects (CRO)
- CAN message identifier for Data Transmit Objects (DTO)
- CAN Calibration Protocol Station Address
- The duration of the window during which the boot code listens for a download command message

Default Boot Code Parameters on page 49-52 shows the default values for these parameters. These defaults should be suitable for most applications.

Default Boot Code Parameters

Parameter	Default Value
CAN bit rate	500000
CCP station address	1
CAN message identifier (CRO)	6FA
CAN message identifier (DTO)	6FB
Duration of listening window	40 ms

Running Applications with a Debugger

It is possible to run an application with a debugger. To have full debugging capabilities it is necessary that both the application and device driver libraries are built with debug switches enabled.

To run an application with a debugger it is necessary you must go through the following steps.

1. In the model **Configuration Parameters** dialog, under the **MPC5xx** options section ensure that **Optimize compiler for** is set to `debug`.

2. In the Target Preferences, ensure that the debug compiler switches are set appropriately for your configuration; see "Setting Target Preferences for MPC5xx" on page 49-11 for examples.

3. By default the device driver libraries are compiled without debug flags; if you need to be able to debug device driver code as well as model code you must re-build the device driver libraries using the debug option. See "Boot Code Parameters for CAN Download" on page 49-52.

Once you have performed the above steps and built your model, you can run it with the source level debugger. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window. Select `Debug RAM-Based Application (via BDM)` from the drop-down list, and click **OK**.

Parameter Tuning and Signal Logging

- "Methods for Parameter Tuning and Signal Logging" on page 49-53
- "Using External Mode" on page 49-54
- "Using a Third Party Calibration Tool" on page 49-64
- "Data Acquisition (DAQ) List Configuration" on page 49-66

Methods for Parameter Tuning and Signal Logging

The coder product supports parameter tuning and signal logging either using Simulink external mode or with a third party calibration tool. In both cases the model must include a special block, the CAN Calibration Protocol block (see MPC5xx CAN Calibration Protocol).

Using External Mode

The Simulink external mode feature enables you to log signals and tune parameters without requiring a calibration tool. This section describes the steps for converting a model to use external mode.

External mode is supported using the CAN Calibration Protocol block and ASAP2 interface. The CAN Calibration Protocol block is used to communicate with the target, downloading parameter updates and uploading signal information. The ASAP2 interface is used to get information about where in the target memory a parameter or signal lives.

Note You must configure the CAN application channel. See "Configuring the Host Vector CAN Application Channel" on page 49-55.

To prepare your model for external mode, follow these steps:

1. Add a CCP driver block.

2. Add a Switch External Mode Configuration Block (for ease of use; you can also make changes manually).

3. Identify signals you want to tune, and associate them with `Simulink.Parameter` or `canlib.Parameter` objects with `ExportedGlobal` storage class. It is important to set the data type and value of the parameter object. See "Using Supported Objects and Data Types" on page 49-56.

4. Identify signals you want to log, and associate them with `canlib.Signal` objects. It is important to set the data type of the `canlib.Signal`. See "Using Supported Objects and Data Types" on page 49-56.

   For information about visualizing logged signal data, see "Viewing and Storing Signal Data" on page 49-59.

5. Load the `Simulink.Parameter` or `canlib.Parameter` and `canlib.Signal` data objects into the base workspace.

6. Configure the model for building by double-clicking the Switch External Mode Configuration block. In the dialog box, select **Building an executable**, and click **OK**.

7. Build the model, and download the executable to the target

8. After downloading the executable to the target, you can switch the model to external mode by double-clicking the Switch External Mode Configuration Block. In the dialog box that appears, select **External Mode**, and click **OK**.

9. You can now connect to the target using external mode by clicking the **Connect** button.

10. If you have set up tunable parameters, you can now tune them. See "Tuning Parameters" on page 49-57.

If you do not want to use the Switch External Mode Configuration block, you can configure for building and then external mode manually. For instructions, see "Manual Configuration For External Mode" on page 49-62.

See the following topics for more information:

- "Configuring the Host Vector CAN Application Channel" on page 49-55
- "Using Supported Objects and Data Types" on page 49-56
- "Tuning Parameters" on page 49-57
- "Viewing and Storing Signal Data" on page 49-59
- "Manual Configuration For External Mode" on page 49-62
- "Limitations" on page 49-62

Configuring the Host Vector CAN Application Channel. External mode expects that the Host CAN connection is using the `'MATLAB 1'` application channel. To configure the application channel used by the Vector CAN drivers, enter the following at the MATLAB command line:

```
TargetsComms_VectorApplicationChannel.configureApplicationChannels
```

The Vector CAN Configuration tool appears. Use this tool to configure your Host CAN channel settings.

If you try to connect using an application channel other than `'MATLAB 1'`, then you see the following warning in the command window:

```
Warning:
```

```
It was not possible to connect to the target using CCP.
An error occurred when issuing the CONNECT command.
```

If you have not already installed the Vector CAN drivers, you will get the following error message:

```
??? Error using ==>
TargetsComms_VectorApplicationChannel.TargetsComms_VectorApplicationChannel>
TargetsComms_VectorApplicationChannel.configureApplicationChannels at 40
Unable to launch the application channel configuration utility. The "vcanconf"
 utility was not found on the Windows
System Path. To fix this error, make sure the required CAN drivers are
installed on this computer; refer to the product
documentation for details.
```

If you want to use CAN to transmit or receive CAN messages between your host PC and your target, you require Vector-Informatik CAN hardware supported by the Vector CAN Driver Library. You must install the correct driver libraries to support profiling, downloading, and external mode.

Note For CANcaseXL, you must install *both* the Vector XL-driver library and Vector CAN Driver Library `vcand32.dll`.

For older CAN hardware, you must install the Vector CAN Driver Library `vcand32.dll`.

Make sure that the library, `vcand32.dll`, is placed in the Windows `system32` folder.

Using Supported Objects and Data Types. Supported objects:

- `Simulink.Parameter` or `canlib.Parameter` for parameter tuning
- `canlib.Signal` for signal logging

Supported data types:

- uint8, int8

- uint16, int16
- uint32, int32
- single

You need to define data objects for the signals and parameters of interest for ASAP 2 file generation. For ease of use, create a MATLAB file to define the data objects, so that you only have to set up the objects once.

To set up tuneable parameters and signal logging:

1. Associate the parameters to be tuned with `Simulink.Parameter` or `canlib.Parameter` objects with ExportedGlobal storage class. It is important to set the data type and value of the Simulink.Parameter object. See the following code for an example of how to create such a Simulink.Parameter object for tuning:

   ```
   stepSize = Simulink.Parameter;
   stepSize.DataType = 'uint8';
   stepSize.RTWInfo.StorageClass = 'ExportedGlobal';
   stepSize.Value = 1;
   ```

2. Associate the signals to be logged with canlib.Signal objects. It is important to set the data type of the canlib.Signal. The following code example shows how to declare such a canlib.Signal object for logging:

   ```
   counter = canlib.Signal;
   counter.DataType = 'uint8';
   ```

3. Associate the data objects you have defined in the file with parameters or signals in the model. For the previous code examples, you could set the **Constant value** in a Source block to `stepSize`, and set a **Signal name** to `counter` in the Signal Properties dialog box. Remember that `stepSize` and `counter` are data objects defined in the code.

Tuning Parameters. To tune a parameter, follow these steps:

1. Set `*dataobject*.value` in the workspace while the model is running in external mode. For example, to tune the parameter `stepSize` (that is, to change its value) from 1 to 2, enter the following at the command line:

```
stepSize.value = 2
```

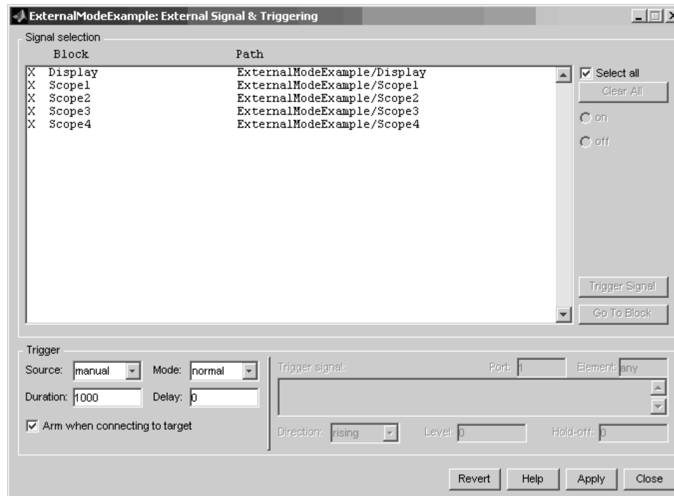
You see output similar to the following:

```
stepSize =

Simulink.Parameter (handle)
        RTWInfo: [1x1 Simulink.ParamRTWInfo]
    Description: ''
       DataType: 'uint8'
            Min: -Inf
            Max: Inf
       DocUnits: ''
          Value: 2
     Complexity: 'real'
     Dimensions: [1 1]
```

2 Return to your model, and update the model (press **Ctrl+D**) to apply the changed parameter.

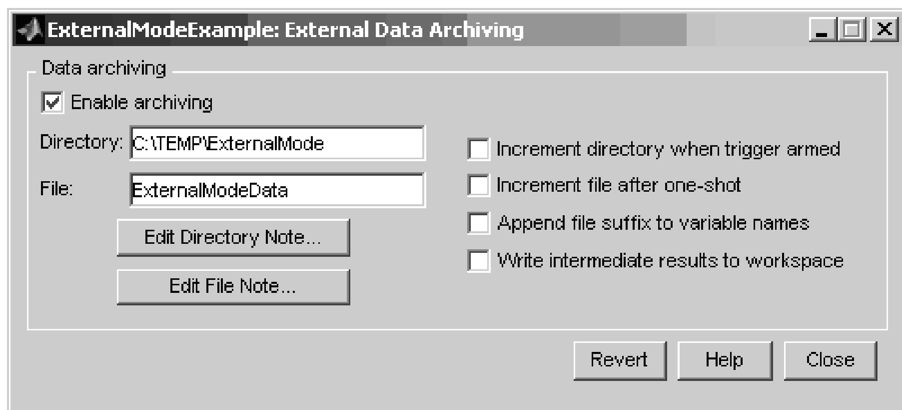
Viewing and Storing Signal Data. To view the logged signals attach a supported scope type to the signal (see "Limitations" on page 49-62 for supported scope types).

Select which signals you want to log by using the External Signal & Triggering dialog box. Access the External Mode Control Panel from the Tools menu, and click the **Signal & Triggering** button. By default, all displays appear as selected to be logged, as shown in the following example. Edit these settings if you do not want to log all displays. Individual displays can be selected manually.



Storing signal data for further analysis. It is possible to store the logged data for further analysis in MATLAB.

1 To use the Data Archiving feature of external mode, click **Data Archiving** in the External Mode Control Panel. The External Data Archiving dialog box appears.

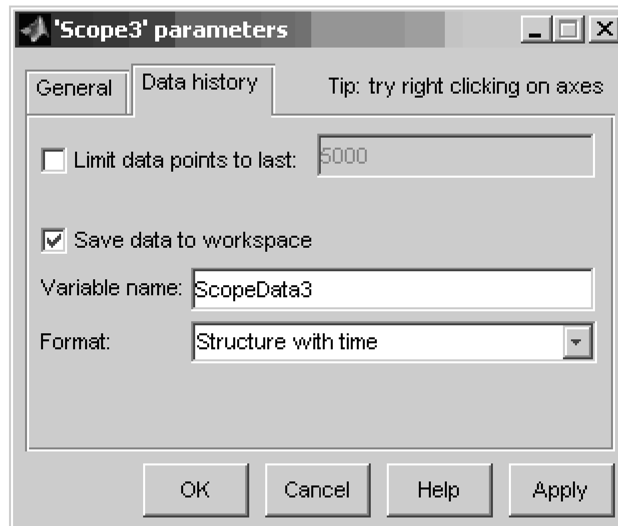


a Select the check box **Enable archiving**

b Edit the **Directory** and **Filename** and any other desired settings.

c Close the dialog box.

2 Open the Scope parameters, and select the check box **Save data to workspace**.



3 You may want to edit the **Variable name** in the edit box. The data that is displayed on the scope at the end of the external mode session is available in the workspace with this variable name.

The data that was previously displayed in the scope is stored in `.mat` files as previously setup using Data Archiving.

For example, at the end of an external mode session, the following variable and files could be available in the workspace and current folder:

- A variable `ScopeData5` with the data currently displayed on the scope:

```
ScopeData5

ScopeData5 =

         time: [56x1 double]
      signals: [1x1 struct]
    blockName: 'mpc555rt_ccp/Scope1'
```

- In the current folder, .mat files for the three previous **Durations** of scope data:

```
ExternalMode_0.mat
ExternalMode_2.mat
ExternalMode_1.mat
```

Manual Configuration For External Mode. As an alternative to using the Switch External Mode Configuration block, you can configure models manually for build and execution with external mode.

To configure a model to be built for external mode:

1. Select **Inline parameters** (under Optimization in the Configuration Parameters dialog box). The **Inline parameters** option is required for ASAP2 generation.

2. Select **Normal** simulation mode (in either the Simulation menu, or the drop-down list in the toolbar).

3. Select `ASAP2` as the **Interface** (under **Code Generation**, **Interface**, in the **Data Exchange** pane, in the Configuration Parameters dialog box).

After you build the model, you can configure it for external mode execution:

1. Make sure **Inline parameters** are selected (under **Optimization** in the Configuration Parameters dialog box). The **Inline parameters** option is required for external mode.

2. Select **External** simulation mode (in either the **Simulation** menu, or the drop-down list in the toolbar).

3. Select `External mode` as the **Interface** (under **Code Generation**, **Interface**, in the **Data Exchange** pane, in the Configuration Parameters dialog box).

Limitations. Logging of multiple signals feeding the same scope block is not supported. Instead, log each signal with its own scope block. These multiple signals can be on the same Simulink line, or can be multiple lines feeding the same scope (i.e. the scope can have multiple axes).

Only the following kinds of scopes are supported with External Mode Logging:

- Simulink Scope block
- Simulink Display block
- Viewer type: scope — To use this option, right-click a signal in the model, and select **Create & Connect Viewer > Simulink > Scope**. The other scope types listed there are not supported (e.g., floating scope).

  Before connecting to external mode, you also need to right-click the signal, and select **Signal Properties**. In the dialog box, select the **Test point** check box, and click **OK**.

GRT is supported but only for parameter tuning.

If a signal comes directly from a Rate Transition block, external mode may fail to detect the correct sample time. To work around this, place a nonvirtual block (e.g., Contiguous Copy) in between the Rate Transition block and the signal to log.

It is not possible to log signals with very fast sample times (e.g., 0.0001) without losing data.

Subsystem builds are not supported for external mode, only top-level builds are supported.

Logging and tuning of nonscalars is not supported. It is possible to log nonscalar signals by breaking the signal down into its scalar components. For an example of how to do this signal deconstruction, see the CCP demo models, which use a Demux and Signal Conversion block with contiguous copy.

Logging and tuning of complex numbers is not supported. It is possible to work with complex numbers by breaking the complex number down into its real and imaginary components. This breakdown can be performed using the following blocks in the Simulink Math Operations library: Complex to Real-Imag, Real-Imag to Complex, Magnitude-Angle to Complex, Complex to Magnitude-Angle.

Using a Third Party Calibration Tool

The coder product allows an ASAP2 data definition file to be generated during the code generation process. This file can be used by a third party tool to access data from the real-time application while it is executing.

ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description used for data measurement, calibration, and diagnostic systems. You can use the Embedded Coder real-time target features to export an ASAP2 file containing information about your model during the code generation process.

Before you begin generating ASAP2 files with the real-time target, you should read the “Generating ASAP2 Files” section of the product help. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

The process of generating an ASAP2 file from your model with the real-time target is similar to that described in the product help.

How the Process Works. The `mpc555rt_ccp` demo provides an example of the ASAP2 file generation feature.

The coder product generates an initial ASAP2 file during the code generation process. At this point, the addresses of signals and parameters on the target system are unavailable, since the code has not been compiled and linked. The initial ASAP2 file contains placeholders for the unresolved addresses.

To supply the required memory addresses, the generated code must be compiled and the compiler must generate a MAP file.

After the build process, if the real-time target detects the presence of the ASAP2 file and a MAP file in the required format, it performs a post-processing phase. During this phase, the MAP file is used to propagate the required address information back into the ASAP2 file.

MAP file formats differ between compilers, so the post processing phase is compiler-specific. The coder product provides the post-processing mechanism for both supported toolchains (Wind River and CodeWarrior).

To use the ASAP2 file generation feature, you simply need to select the **ASAP2** file option in the Configuration Parameters dialog box, as described in the following section "ASAP2 File Generation Procedure" on page 49-65. If it is appropriate to back propagate addresses from the MAP file into the ASAP2 file, then this will also be done automatically. No other steps are necessary to ensure that the generated MAP and ASAP2 files are automatically post processed.

The names of the ASAP2 file and the MAP file derive from the source model. The MAP file is generated in the same folder as the source model. The ASAP2 file is written to the build folder.

ASAP2 File Generation Procedure.

1. Create the desired model. Use appropriate parameter names and signal labels to refer to ASAP2 `CHARACTERISTICS` and `MEASUREMENTS` respectively.

2. Define the corresponding `Simulink.Parameter` and `Simulink.Signal` objects in the MATLAB workspace.

3. Configure the data objects to generate unstructured global storage declarations in the generated code by assigning one of the following storage classes to the `RTWInfo.StorageClass` property for each object:

   - `ExportedGlobal`
   - `ImportedExtern`
   - `ImportedExternPointer`

   `ExportedGlobal` is the default storage class.

4. Configure the other data object properties for each object. See "Defining ASAP2 Information" in the Simulink Coder documentation.

5. In your model window, select the menu item **Simulation > Configuration Parameters**.

   The **Configuration Parameters** dialog box appears.

6. Select **Optimization** in the tree.

7. Select the **Inline parameters** option.

Note that you should *not* configure the parameters associated with your data objects in the **Model Parameter Configuration** dialog box (reached via the **Configure** button). If a parameter that resolves to a Simulink data object is configured using the **Model Parameter Configuration** dialog box, the dialog box configuration is ignored. You can, however, use the **Model Parameter Configuration** dialog to configure other parameters in your model.

8. Under **Code Generation**, select **Interface** in the tree.

9. Select the **ASAP2** option from the **Interface** drop-down menu in the `Data exchange` frame.

10. Click **Apply**.

11. Select **Code Generation** in the tree, then click **Build**.

    The ASAP2 file is generated as part of the build process.

Data Acquisition (DAQ) List Configuration

The coder product supports the Data Acquisition (DAQ) List feature of the CAN Calibration Protocol (CCP). DAQ lists allow efficient synchronous signal monitoring. The CCP block supports DAQ lists (see MPC5xx CAN Calibration Protocol for details).

`Simulink.Signal` objects are used for monitoring a signal in the CCP polling mode of operation. To monitor a signal in a DAQ list, however, you must configure the signal somewhat differently. The differences are as follows:

- Instead of defining a `Simulink.Signal` in the MATLAB workspace (and associated signal in the Simulink model), define a `canlib.Signal` object instead.
- There is no need to set the `RTWInfo.StorageClass` property of the `canlib.Signal` object. By default, the storage class is set to `Custom`.
- You should enter data in the other fields of the `canlib.Signal` object in the same way you would do for a `Simulink.Signal` object.

Note In order to use the `canlib.Signal` objects, the model must contain a CAN Calibration Protocol block. See MPC5xx CAN Calibration Protocol.

During code generation, the coder product automatically determines how to configure the DAQ lists in the generated code. For each distinct sample rate (of the set of `canlib.Signal` objects assigned by the user) one DAQ list in the model is created. The CCP DAQ List Object Descriptor Tables (ODTs) are shared equally between the created DAQ lists.

The sample rates of the `canlib.Signal` objects are mapped to CCP event channels in an extra file, `DAQ_LIST_EVENT_MAPPINGS`, that is generated in the build folder. This shows how to assign event channels to `MEASUREMENT` signals in a calibration package.

The event channels periodically transmit events that are used to trigger the sending of DAQ data to the host. By assigning event channels as defined in `DAQ_LIST_EVENT_MAPPINGS`, consistent and efficient transmission of DAQ data is achieved.

It is the responsibility of the calibration tool (see "Compatibility with Calibration Packages") to assign an event channel and data to the available DAQ lists using CCP commands, and to interpret the synchronous response.

It is the responsibility of the user to make sure the calibration tool is set up correctly and that the event channels assigned to `MEASUREMENT` signals correspond to those defined in the file `DAQ_LIST_EVENT_MAPPINGS`.

HTML Code Profile (RAM/ROM) Report

The coder product supports an extended version of the HTML code generation report.

For instructions, see "HTML Code Analysis (RAM/ROM) Report" on page 49-104. You can generate reports for the real-time target, processor-in-the-loop (PIL) target and algorithm export (AE) target.

Execution Profiling

- "Overview of Execution Profiling" on page 49-68
- "The Profiling Command" on page 49-69
- "Execution Profiling Definitions" on page 49-71
- "MPC5xx Options for Execution Profiling" on page 49-72
- "Interpreting the Execution Profiling Graphic" on page 49-74
- "Enabling Execution Profiling for Device Driver Interrupt Service Routines" on page 49-75

Overview of Execution Profiling

The coder product provides a set of utilities for recording, uploading and analyzing execution profile data for timer-based tasks and asynchronous Interrupt Service Routines (ISRs). With these utilities, you can

- Generate a graphical display that shows when timer-based tasks and interrupt service routines are activated, preempted, resumed and completed.
- Generate a report with information on
  - Maximum number of overruns for each timer-based task since model execution began
  - Maximum turnaround time for each timer-based task since model execution began
  - Analysis of profiling data for timer-based tasks and asynchronous interrupts over a period of time

You can use the demo model `mpc555rt_multitasking` to see an example. This demo model illustrates both execution profiling and the preemptive multitasking scheduler with configurable overrun handling. For instructions, click the link MPC555 Multitasking Demo.

To perform execution-profiling analysis on a model, you must perform the following steps:

1. Depending on whether you are using serial or CAN, place a copy of the appropriate execution profiling block in your model (MPC555 Execution Profiling via SCI1 or MPC555 Execution Profiling via CAN A).

2. Connect a serial or CAN cable between the target processor and your host PC.

3. Check the box to enable Execution profiling in the Configuration Parameters dialog box. See “MPC5xx Options for Execution Profiling” on page 49-72.

4. Build, download and run the model.

5. Initiate execution profiling by running the command `profile_mpc555`. See below for more information on the profiling command.

Two forms of execution profiling are provided:

1. The worst-case values for task turnaround times and number of task overruns since model execution began are updated whenever a previous worst-case value is exceeded.

2. A snapshot of task and ISR activity may be recorded over a period of time; the length of this period depends on how much memory is available to log the data.

Note You need additional steps if device drivers use interrupt service routines (may include CAN, TPU, and QSPI). See “Enabling Execution Profiling for Device Driver Interrupt Service Routines” on page 49-75. If this is not done, then no profiling information will be recorded.

The Profiling Command

Use the profiling command as follows:

```
profile_mpc555(connection)
```

Specify your `connection` as `'can'` or `'serial'`, to collect data via a CAN or serial connection between the target and the host computer. Make sure the

model includes the appropriate MPC5xx execution profiling block (CAN or SCI1), to provide an interface between the target-side profiling engine and the host-side computer from which this command is run.

`PROFDATA = profile_mpc555(connection)` collects and displays execution profiling data from a MPC5xx target microcontroller that is running a suitably configured application generated by the coder product. `PROFDATA` contains the execution profiling data in the format documented by `exprofile_unpack`.

The data collected is unpacked then displayed in a summary HTML report and as a MATLAB graphic.

To use the serial connection, the MPC5xx board must be connected via a serial cable to one of the host computer's serial ports. This function defaults to port SCI1 on the MPC5xx and port COM1 on the host computer. If the `'BitRate'` argument is not provided, the default of 57600 baud is used.

```
PROFDATA = PROFILE_mpc555('serial','SerialPort',serialport)
```

sets the serial port to the specified `serialport`, which should be one of COM1, COM2, etc.

Optionally, you can specify the bit rate as follows:

```
PROFDATA = PROFILE_mpc555('serial', 'BitRate', bitrate)
```

This specification sets the `bitrate` for serial connection to the target. `bitrate` must be the same as the bit rate specified for the application that is running on the target.

Alternatively, you can set the bitrate for the serial connection to the target automatically as follows:

```
profdata = profile_mpc555('serial', 'ModelName', modelname)
```

This specification automatically sets the bit rate by analyzing *`modelname`* and extracting the correct serial connection bit rate setting from the model. *`modelname`* should be set to the name of a model which is currently open and running on the target.

To use the CAN connection, you must have suitable CAN hardware installed. If no Application Channel is specified, this function will use the channel 'MATLAB 1'. The bit rate is a property of the Application Channel; to change the bit rate, you must use a different Application Channel, or change the bit rate by running the Vector Informatik configuration utility. To run this utility, make sure that vcanconf.exe is on your System Path, then type vcanconf from a Windows command prompt.

You can specify the Application Channel as follows:

```
profdata = profile_mpc555('can', 'CANChannel', canchannel)
```

canchannel specifies the Vector Informatik CAN Application Channel, and must be of the form 'MATLAB 1', 'MATLAB 2' etc.

Execution Profiling Definitions

Task turnaround time
: the elapsed time between start and finish of a task. If the task is not preempted then the task turnaround time is equal to the task execution time.

Task execution time
: that part of the time between task start and finish when the task is actually running and not preempted by another task. Note that the task execution time cannot be measured directly, but is inferred from the task start and finish time and the intervening periods during which it was preempted by another task. Note that, in performing these calculations, no account is taken of processor time consumed by the scheduler while switching tasks: this means that, in cases where preemption has occurred, the reported task execution times will overestimate the true values.

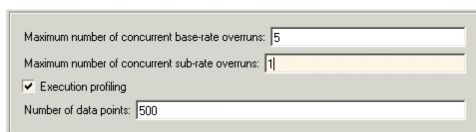
Task overruns
: occur when a timer task does not complete before that same task is next scheduled to run. Depending on how the real-time scheduler is configured, a task overrun may be handled as a real-time failure. Alternatively, a small number of concurrent task overruns may be allowed in order to accommodate cases where a task occasionally takes longer than normal to complete.

See also "Interpreting the Execution Profiling Graphic" on page 49-74.

The Execution Profiling Block. See the Block Reference section MPC5xx MPC555 Execution Profiling via SCI1 or MPC5xx MPC555 Execution Profiling via CAN A.

MPC5xx Options for Execution Profiling

You can see these options on the `ET MPC5xx real-time options (2)` section (under **Code Generation** in the tree) of the Configuration Parameters dialog box.



Execution profiling

If this option is checked then the generated code for the model will be instrumented with function calls at the beginning and end of each task or ISR to be profiled. These function calls read a timer (on MPC555 it is the decrementer timer) and log this reading along with a task identifier.

When code for the model is generated, these functions will update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous worst case value is exceeded. Additionally, when a trigger is provided, data will be logged over a period of time to record all task start and task finish times. The trigger signal can be supplied by the execution profiling blocks. See MPC5xx MPC555 Execution Profiling via SCI1 or MPC5xx MPC555 Execution Profiling via CAN A.

Number of data points

When a snapshot of task and ISR activity is logged this data is stored in memory that is statically allocated at build time. Each data point requires 8 bytes on the MPC555. The larger the number of data points to be stored, the more RAM that must be reserved for this purpose. At the end of a logging run, the data must be uploaded to the host computer for analysis; this is typically achieved by using the execution profiling blocks.

Overrun Options. These options configure the allowable number of task overruns. You can see these options on the `ET MPC5xx real-time options (2)` section (under **Code Generation** in the tree) of the **Configuration Parameters** dialog.

Maximum number of concurrent base-rate overruns: 5
Maximum number of concurrent sub-rate overruns: 1
Execution profiling
Number of data points: 500

You can use the options **Maximum number of concurrent base-rate overruns** and **Maximum number of concurrent sub-rate overruns** to configure the behavior of the scheduler when any of the timer based tasks do not complete within their allowed sample time. It is useful to allow task overruns in the case where a task may occasionally take longer than usual to complete (e.g. if extra processing is required when a special event occurs); if the task overrun is only occasional then it is possible for the scheduler to 'catch up' after the extra processing has been completed.

If the maximum number of concurrent overruns for any task is exceeded, this is deemed to be a failure and the real-time application is stopped. This in turn will result in a watchdog timer timeout and the processor will be reset.

As an example, if the base rate is 1 ms and the maximum number of concurrent base-rate overruns is set to 5 then it is possible for the base rate task to run for almost 6 ms before failure occurs. Once the overrun has occurred, it is necessary for subsequent executions of the base rate to complete in less than 1 ms in order that the lost time is recovered.

The occurrence of base-rate overruns does not affect the numerical behavior of the algorithm (although reading/writing external devices will of course be delayed).

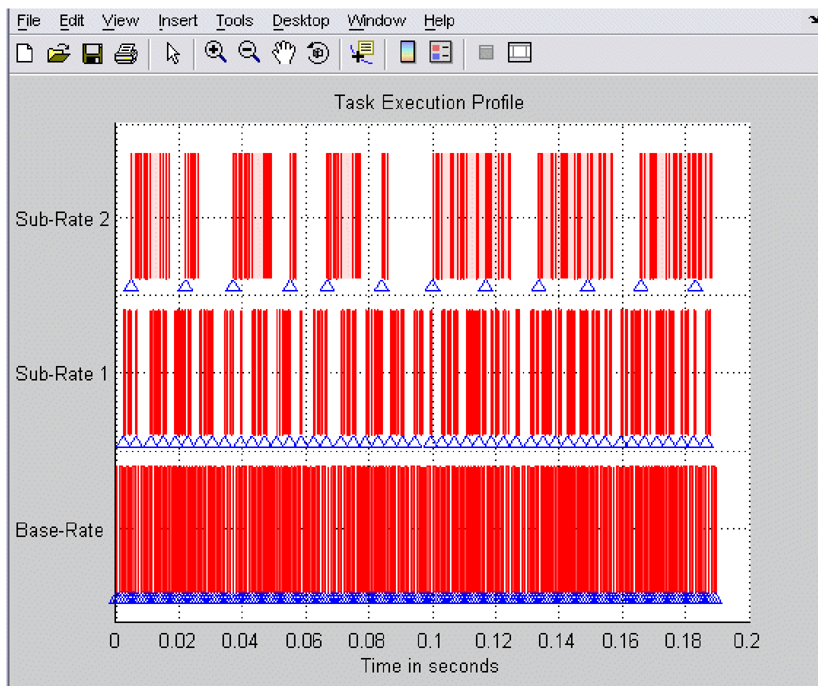
If sub-rate overruns are allowed then the transfer of data between different rates (via rate-transition blocks) in the model may be affected; this causes the numerical behavior in real-time to differ from the behavior in simulation. To see an illustration of this effect try running the demo model `mpc555rt_multitasking`. To disallow sub-rate overruns and ensure that this

effect does not occur, you should set **Maximum number of concurrent sub-rate overruns** to zero.

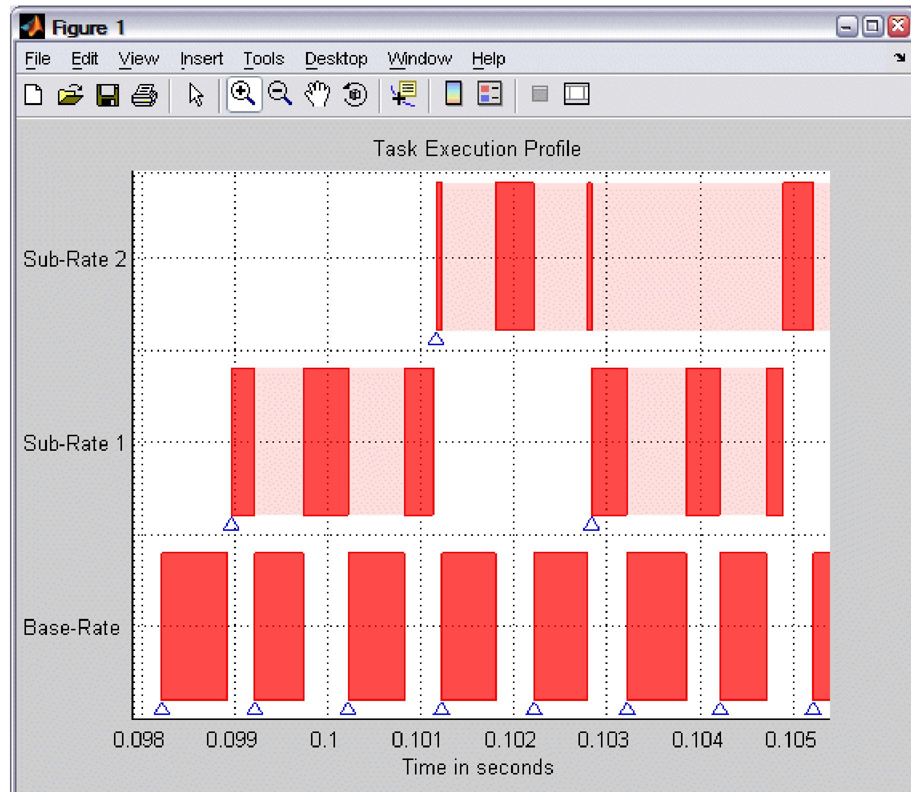
Note If the option "Maximum number of concurrent sub-rate overruns" is set to a value greater than zero, then the behavior of any Rate-Transition blocks may be affected. Specifically, if the model contains a Rate Transition block where the option "Ensure deterministic data transfer (maximum delay)" is selected, then this setting may not be honored.

Interpreting the Execution Profiling Graphic

Dark shaded areas show the region where a task is executing. Light shaded areas show the region where a task is preempted by a higher priority task or ISR. Triangles indicate the beginning of a task. An example is shown following.



Zoom in to see the details of tasks executing and being preempted, as shown in the following example.



Enabling Execution Profiling for Device Driver Interrupt Service Routines

By default, execution profiling is not enabled for device driver interrupt service routines. Device drivers that may use interrupt service routines include CAN, TPU and QSPI device drivers.

You can enable execution profiling for device driver interrupt service routines. To do this, you must rebuild the device drivers libraries with a macro `PROFILING_ENABLED` defined. Follow these steps:

1 Remove the previously built device driver code using one of the following methods.

   a Run the command:

   ```
   mpc555_build_drivers('clean')
   ```

   b Delete the contents (compiled object code) of the folder

   ```
   matlab\toolbox\rtw\targets\mpc555dk\drivers\src\libsrc\standard\src\bin\COMPILER\XXX
   ```

   where *COMPILER* is one of DIAB or CODE_WARRIOR and *XXX* is the MPC5xx variant you are using.

   The second approach will result in a faster rebuild in the next step.

2 Run the command:

   ```
   mpc555_build_drivers(BUILD_OPTION,'ProfileDeviceDrivers', 'on')
   ```

   Set *BUILD_OPTION* to one of the options 'speed', 'size', or 'debug'.

When rebuilding the driver library using the command mpc555_build_drivers, the compiler and compiler switches used are taken from the currently selected compiler configuration in the Target Preferences.

Summary of the Real-Time Target

- “Code Generation Options” on page 49-76
- “Requirements and Restrictions” on page 49-78

Code Generation Options

The real-time target is an extension of the embedded real-time (ERT) target configuration. The real-time target inherits the code generation options of the ERT target, as well as the general code generation options. These options are available under **Code Generation**, in the tree on the **Configuration Parameters** dialog box.

Some code generation options of the ERT target are not relevant to the real-time target, and are either unsupported, or restricted in their operation. See “Requirements and Restrictions” on page 49-78 for details.

Target-Specific Options. The real-time target has several target-specific code generation options. To view or change the setting of these options, select the `ET MPC5xx real-time options(1)` section in the **Configuration Parameters** dialog.

- **Optimize compiler for** — Select `speed`, `size`, `debug`, or `custom`.

  This option controls compiler optimization switches used during the build process. The exact effect of the optimization switches depends on whether you are using the Wind River or CodeWarrior compiler. You can optimize for performance by choosing the `speed`, `size`, or `debug` options, or define your own (the `custom` option). You can edit these preferences here in the **Compiler optimization switches** edit box if you want to apply changes to the current model (**Optimize compiler for** will change to `custom`). You can also edit the defaults for these settings in the **Target Preferences** dialog if you want to apply these changes to several models. See “Compiler Optimization Switches” on page 49-14 for more information.

- **Target memory model** Select either `FLASH` or `RAM`.

  If you select the `FLASH` option, files in a format suitable for downloading into the MPC555 on-chip flash memory are written. If you select the `RAM` option, files in a format suitable for downloading into RAM are generated.

  In both cases these two files are generated, with this naming convention:

  - *`model`*`_flash.s19` or *`model`*`_ram.s19` — code only, for CAN download
  - *`model`*`_flash.elf` or *`model`*`_ram.elf` — for BDM download, containing code and optional debugging symbols if you choose a `debug` build in the **Optimize compiler for** settings

- **Build action**

  - `None` — code generation only.
  - `Launch_Download_Control_Panel` — on completion of code generation the **Download Control Panel** utility is opened.
  - `Run_via_BDM` — on completion of code generation download over BDM connection automatically starts and on completion the code is run.

- Debug_via_BDM — on completion of code generation download over BDM connection automatically starts. When the download is complete the code stops at the first line while debugging, so you can step through the code.

- **Use prebuilt libraries**

  This check box option (selected by default) determines whether prebuilt libraries, compiled with default compiler switches, are linked against during compilation of the generated code. When this option is not selected, the source modules that comprise these libraries will be compiled individually in the model build folder, using the currently selected compiler switches.

  Using prebuilt libraries saves a considerable amount of time during the build process.

Requirements and Restrictions

MPC555 Resource Configuration Block Required. To generate code from a model using the real-time target, an MPC555 Resource Configuration block must be included in the model. The MPC555 Resource Configuration block is required even for models that do not contain any MPC555 device driver blocks.

Note When using device driver blocks from the Embedded Targets libraries in conjunction with the MPC555 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the MPC555 Resource Configuration block will operate incorrectly. See MPC5xx MPC555 Resource Configuration for further information.

Model Reference and Driver Blocks. Referenced sub-models that contain driver blocks (including the MPC555 Resource Configuration block) cause build failures. All driver blocks must be placed in the top level model. It is not possible to include driver blocks in any of the referenced sub-models.

Restricted Code Generation Options. Certain ERT code generation options are not supported by the real-time target. If these options are selected, the real-time target either ignores the option or issues an error message during the build process. Real-Time Target Restricted Code Generation Options on page 49-79 summarizes these restricted options.

Real-Time Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Block type	Error if `SIL` is selected; build process terminates
External mode	Error if selected; build process terminates
Generate an example main program	This option should not be selected for the real-time target. The real-time target supplies a target-specific main program, `mpc555dk_main.c`. Ignored if selected; build proceeds with a warning.
Generate reusable code	Error if selected; build process terminates
Terminate function required	Ignored if selected; if your model includes a block (for example, a custom S-function block) that attempts to generate code for the *`model`*`_terminate` function, then a warning is issued and code for this function is not generated.

Performance Tips

- "Run the Model Advisor" on page 49-79
- "Increase the System Clock Beyond the Default 20 MHz" on page 49-80
- "Use Flash Instead of RAM" on page 49-80
- "TouCAN Interrupt Generator Block Performance Tips" on page 49-80
- "Optimized Target Function Library" on page 49-80

Run the Model Advisor

Following the suggestions in the Model Advisor report may result in faster on-target execution.

Increase the System Clock Beyond the Default 20 MHz

The default system clock frequency is 20 MHz. For higher performance, you should consider increasing the system clock frequency up to 40 MHz, which is the maximum for the MPC555 device. Other processor variants may support higher System Clock Frequencies depending on your development board. Please consult your development board documentation for details.

For more information, see:

- "**System Clock and Related Parameters** " for information on how to change system clock parameters.
- MPC5xx Switch Target Configuration; this is a utility block you can use to apply some predefined configurations.

Use Flash Instead of RAM

Configure the model to run from internal Flash (rather than external RAM) because this has faster memory access. See "Downloading Application Code" on page 49-42.

TouCAN Interrupt Generator Block Performance Tips

When using the TouCAN Interrupt Generator block, you can improve performance as follows:

- Disable **Use floating point** in the TouCAN Interrupt Generator (if possible). This will save significant time during ISR context switches (of which there may be many, depending on the application).
- Minimize the code that runs in the context of the ISR. Try and move as much code out of the ISR (function-call subsystem) as possible to speed up individual ISRs. This should allow an increase in the rate at which CAN messages can be received on that buffer.

Optimized Target Function Library

If your model contains floating-point mathematical function blocks (e.g., trigonometric functions, log functions), then you should use target optimized function libraries. Select the `Freescale MPC5xx (ISO)` option for the **Target function library** (on the **Code Generation > Interface** pane of

the Configuration Parameters dialog box) to use the CodeWarrior or Diab ISO C function library. This generates calls to the Freescale CodeWarrior or WindRiver Diab ISO/IEC 9899:1999 math library for floating-point functions as appropriate.

When you create new models with the `mpc555rt.tlc`, `mpc555rt_grt.tlc` or `mpc555exp.tlc` **System target file**, the `Freescale MPC5xx (ISO)` is automatically selected for the **Target function library** setting.

PIL Simulation

This section includes the following topics:

In this section...
"Overview of PIL Simulation" on page 49-82
"Tutorial 1: Building and Running a PIL Simulation" on page 49-84
"Tutorial 2: Using the Demo Model in Simulation" on page 49-97
"PIL Target Summary" on page 49-98
"Algorithm Export Target" on page 49-103
"HTML Code Analysis (RAM/ROM) Report" on page 49-104
"Algorithm Export Target Summary" on page 49-106

Overview of PIL Simulation

- "What Is PIL Simulation?" on page 49-82
- "Why Use Simulation?" on page 49-82
- "How Simulation Works" on page 49-83

What Is PIL Simulation?

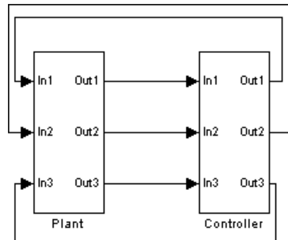
The coder product supports *processor-in-the-loop* (PIL) *simulation*, a technique that is designed to help you evaluate how well a candidate control system operates on the actual target processor selected for the application.

The processor-in-the-loop target is an extended version of the embedded real-time (ERT) target configuration, designed specifically for PIL simulation. We refer to this configuration as the *PIL target*.

Why Use Simulation?

PIL simulation is particularly useful for simulating, testing and validating a controller algorithm in a system comprising a *plant* and a *controller*. In classic closed-loop simulation, the Simulink and Stateflow products model

such a system as two subsystems and the signals transmitted between them, as shown in this block diagram.



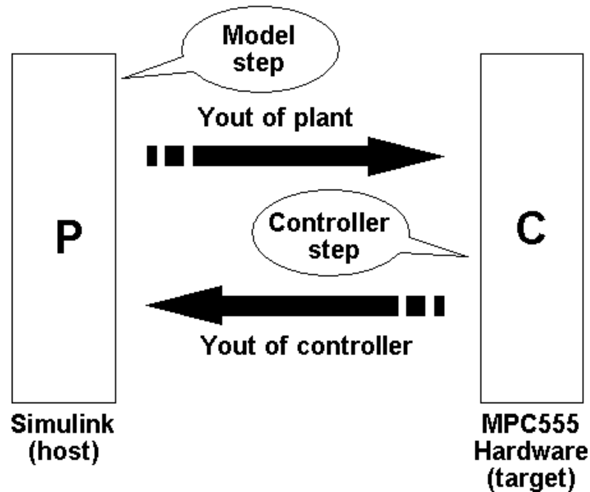
Your starting point in developing a plant/controller system is to model the system as two subsystems in closed-loop simulation. As your design progresses, you can use Simulink external mode with standard targets (such as GRT or ERT) to help you model the control system separately from the plant.

However, these simulation techniques do not help you to account for restrictions and requirements imposed by the hardware. When you finally reach the stage of deploying controller code on the target hardware, you may need to make extensive adjustments to the controller system. Once these adjustments are made, your deployed system may diverge significantly from the original model. Such discrepancies can create difficulties if you need to return to the original model and change it.

PIL simulation addresses these issues by providing an intermediate stage between simulation and deployment. The term *simulation* reflects a division of labor in which Simulink models the plant, while code generated from the controller subsystem runs on the actual target hardware. In a PIL simulation, the target processor participates fully in the simulation loop — hence the term *processor-in-the-loop*.

How Simulation Works

This figure illustrates how the plant (P) and controller (C) components interact in a PIL simulation



In a PIL simulation, coder product generates efficient code for the control system. This code runs (in simulated time) on a target board using the intended microcontroller. The plant model remains in Simulink without the use of code generation.

During PIL simulation, Simulink simulates the plant model for one sample interval and exports the output signals (`Yout` of the plant) to the target board via a communications link. When the target processor receives signals from the plant model, it executes the controller code for one sample step. The controller returns its output signals (`Yout` of the controller) computed during this step to Simulink, via the same communications link. At this point one sample cycle of the simulation is complete and the plant model proceeds to the next sample interval. The process repeats and the simulation progresses.

To learn about PIL simulation though hands-on experience, see "Tutorial 1: Building and Running a PIL Simulation" on page 49-84.

Tutorial 1: Building and Running a PIL Simulation

- "Before You Begin" on page 49-85

- "Hardware Connections" on page 49-85
- "The Demo Model" on page 49-86
- "Setting Up the Model" on page 49-89
- "Building PIL and Simulation Components" on page 49-91
- "Using the Demo Model In a PIL Simulation" on page 49-94
- "Modifying the Controller Subsystem" on page 49-96

Before You Begin

In this tutorial, you will use a subsystem in a Simulink model as a component in simulations on your host computer, and also in a PIL simulation running on your phyCORE-MPC555 board.

Before working with this tutorial, you should read and follow the procedures in "Setting Up and Verifying Your Configuring the Host Vector CAN Application ChannelInstallation" on page 49-10. Make sure that the target preferences are set up appropriately for your development system (CodeWarrior or Wind River) as described in "Setting Target Preferences for MPC5xx" on page 49-11.

Hardware Connections

The PIL target requires that you have a serial cable connection. You can also use serial and CAN, or serial with a BDM connection.

Serial cable is required for host/target PIL communications whilst the model is running, and downloads can occur over serial or CAN so the minimal requirement is a single serial cable. BDM is not required but can be used if desired.

We assume that you have made the following connection, as described in the "Interfacing the phyCORE-MPC555 to a Host PC" section of the *phyCORE-MPC555 Quickstart Instructions* manual: Host PC serial (COM1) port to the RS232-1 (P2) connector on the phyCORE-MPC555 board.

The Demo Model

The Fault-Tolerant Fuel Control System model, shown in Fault-Tolerant Fuel Control System Model on page 49-86, consists of a plant model with a controller subsystem, the `Fuel Rate Controller` subsystem.

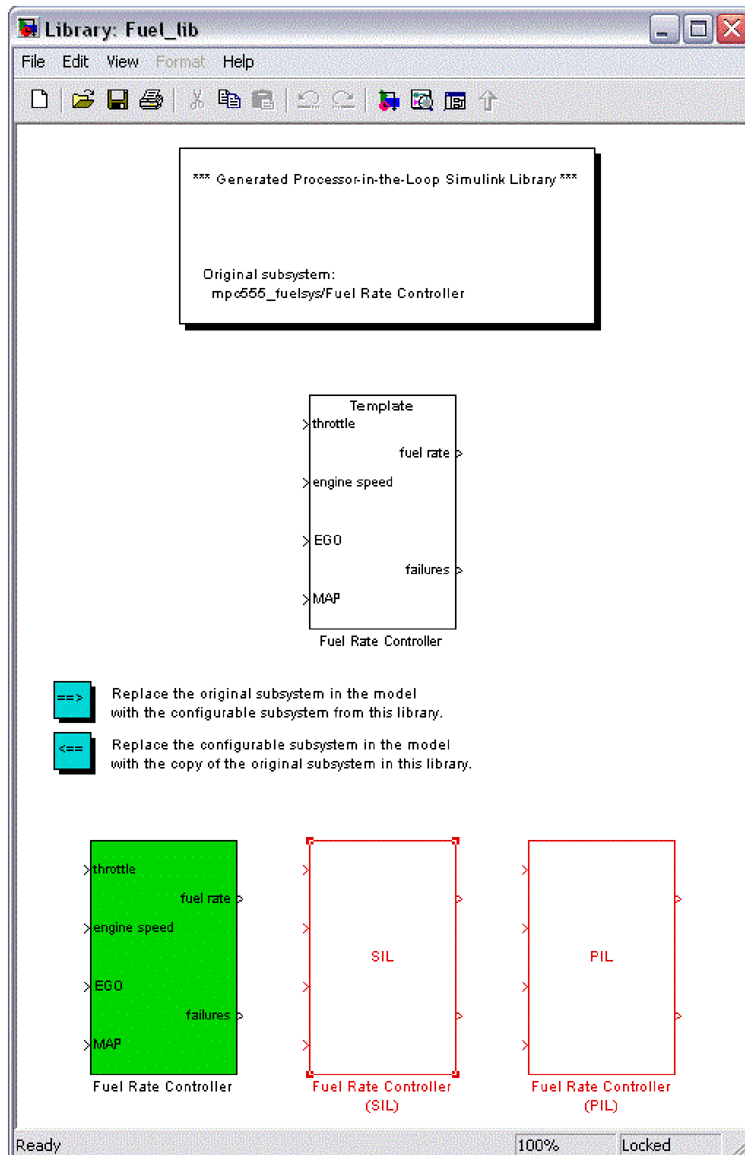
Fault-Tolerant Fuel Control System Model

In the following sections, you will use the demo model and the PIL target to generate the following:

- PIL code to run on the target board. The PIL target automatically invokes the appropriate cross-development tools to compile, link, and (optionally) download and run a target executable.
- A library containing
  - The original `Fuel Rate Controller` subsystem block for use in simulation.
  - An S-function wrapper block, generated by coder product, that implements the `Fuel Rate Controller` subsystem for use in software-in-the-loop (SIL) simulation.

- A subsystem block that implements the `Fuel Rate Controller` subsystem on the host side during simulation. This subsystem communicates with generated PIL code running on the target board.
- A master configurable subsystem block that represents the above three components. You will plug this block into a plant model and select each of the three components in turn for use in a simulation.

This figure shows a library generated by the PIL target.



Once you start the build process, there is almost no manual intervention required to build all these components.

After building the components, you will use them in normal simulation, SIL simulation, and PIL simulation. You will monitor the results of each simulation via the Scope blocks in the model.

Setting Up the Model

In this section you will make a local copy of the demo model and configure the model as required by this exercise:

1. Open the demo model by clicking the link or typing at the command line:

   ```
   mpc555_fuelsys
   ```

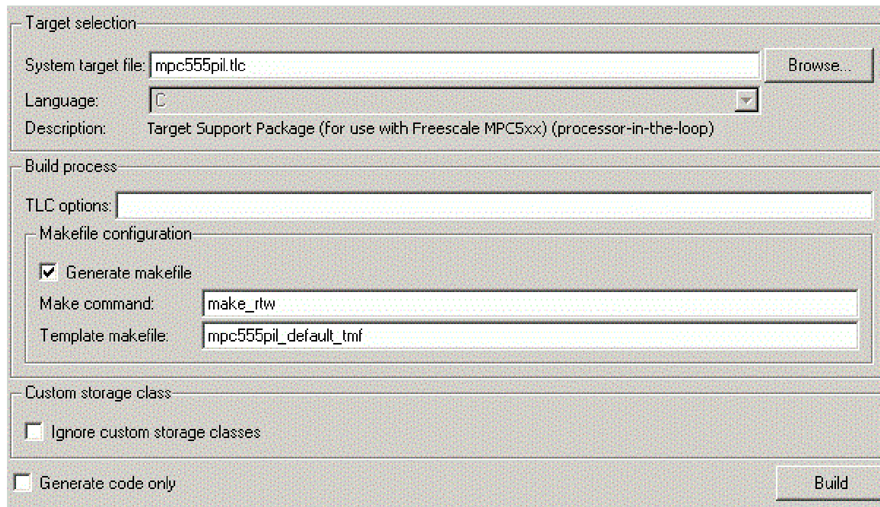
2. Save a copy of the demo model, `mpc555_fuelsys.mdl` to your working folder.

Next, check that the model is correctly configured for use with the coder product.

1. Click on the `Fuel Rate Controller` subsystem, then choose **Configuration Parameters** from the **Simulation** menu. The **Configuration Parameters** dialog opens.

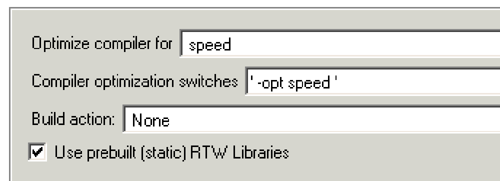
2. Select **Code Generation** in the tree.

3. Observe the `System target file` setting on the **General** tab. The target configuration should be as shown in this figure.



4 To see how to change target configuration settings, click the **Browse** button to open the System Target File Browser, and observe the available system target files — for algorithm export, processor-in-the-loop, and real-time target. Leave the selected file at `mpc555pil.tlc`. Click **Cancel** to close the Browser and return to the **Code Generation** pane.

5 Select`ET MPC5xx (processor-in-the-loop) options` in the tree.



6 Select `Launch_Download_Control_Panel` from the **Build action** drop-down menu. This option automatically invokes the appropriate downloading utility.

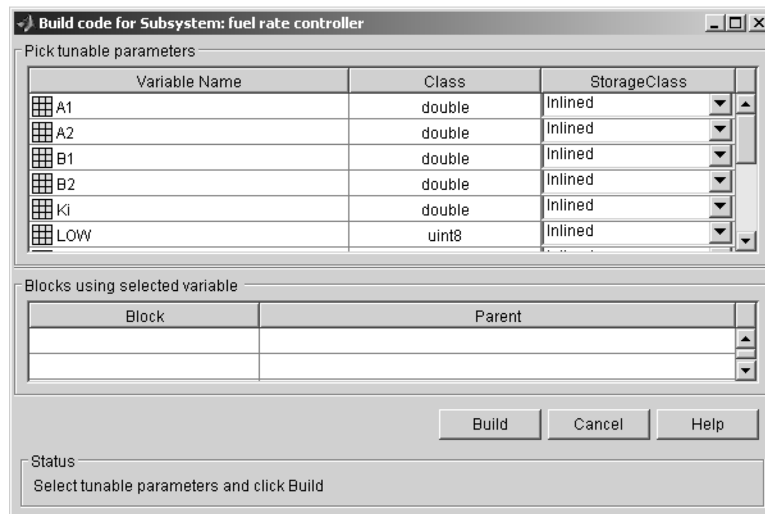
7 Click **Apply**. Then close the **Configuration Parameters** dialog box. If needed, save the model to preserve any changes you have made.

Building PIL and Simulation Components

In this section, you will build a library of simulation, SIL, and PIL components from the `Fuel Rate Controller` subsystem:

1. Right-click on the `Fuel Rate Controller` subsystem. A context menu appears. Select **Build Subsystem** from the **Code Generation** submenu of the context menu.

2. The **Build code for Subsystem** window opens. This window displays information about each variable (or data object) that is referenced as a block parameter in the subsystem. The window lets you inline or set the storage class of individual parameters. We will not be concerned with these features in this exercise. Click the **Build** button to continue the code generation and build process.



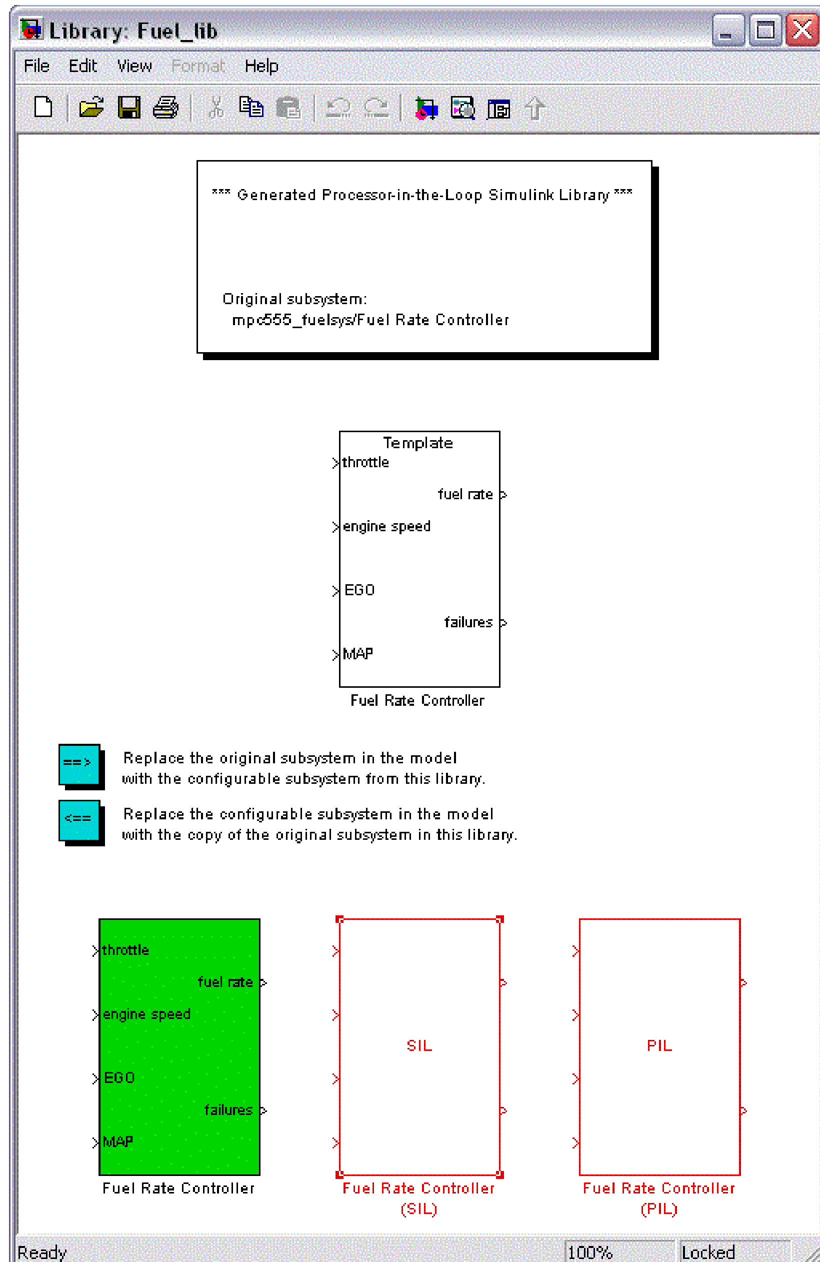
3. The build process displays status messages in the MATLAB command window. Intermediate Simulink windows are displayed as the build process creates various components.

4. When the code generation process completes, the PIL target automates the process of compiling, downloading, and executing the generated PIL code that is to run on the target hardware. To accomplish this, the PIL target launches your cross-development system (Wind River or CodeWarrior),

compiles and makes the executable, and invokes the **Download Control Panel** to download the code to the target. Click **Start Download** in the **Download Control Panel** to complete the process.

5. At this point, the generated program is running on the target hardware and waiting for communication to be established with Simulink on the host PC.

6. The build process has created and opened a library named `Fuel_lib`, as shown in this figure.


Library: Fuel_lib
File Edit View Format Help
*** Generated Processor-in-the-Loop Simulink Library ***
Original subsystem:
mpc555_fuelsys/Fuel Rate Controller
Template
throttle
fuel rate
engine speed
EGO
failures
MAP
Fuel Rate Controller
==>
Replace the original subsystem in the model
with the configurable subsystem from this library.
<==
Replace the configurable subsystem in the model
with the copy of the original subsystem in this library.
throttle
fuel rate
engine speed
EGO
failures
MAP
Fuel Rate Controller
SIL
Fuel Rate Controller
(SIL)
PIL
Fuel Rate Controller
(PIL)
Ready
100%
Locked

The library contains

- A copy of the original `Fuel Rate Controller` subsystem.
- A S-function, labeled `Fuel Rate Controller (SIL)`.
- A subsystem block that communicates with generated PIL code running on the target board during simulation, labeled `Fuel Rate Controller (PIL)`.
- A master configurable subsystem block referencing the other three blocks. The default block choice for this subsystem is the original `Fuel Rate Controller` subsystem.

The configurable subsystem, when plugged into the model, lets you choose which of the three library components will perform the controller functions in the model. We will use the configurable subsystem in the following sections.

The library window also contains the following controls:

- A button that lets you replace the original (generating) subsystem in the model with the generated configurable subsystem.
- A button that lets you do the inverse, i.e., remove the configurable subsystem from the model from the original model and replace it with the original (generating) subsystem from the library.

The library window documents the name of the original model/subsystem from which the library was generated,

Using the Demo Model In a PIL Simulation

In this section, we will plug the configurable subsystem into the demo model, select the PIL component, and use it in a PIL simulation:

1. Click on the `Fuel_lib` library window to activate it. Double-click on the button labeled **Replace the original subsystem in the model with the configurable subsystem from this library**.

2. The `mpc555pil_fuelsys` model window is now the active window. The original `Fuel Rate Controller` subsystem has been deleted from the model. It has been replaced by the configurable subsystem from the

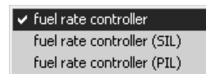
`Fuel_lib` library. The configurable subsystem is automatically connected to the same signals that the original `Fuel Rate Controller` subsystem was connected to.

Note It is important to be aware that the insertion of the configurable subsystem into the containing model establishes a link between the model, `mpc555pil_fuelsys`, and the library, `Fuel_lib`. The library has information about the model and subsystem from which it was generated. The model, in turn, has information about the library from which the configurable subsystem comes. This linkage is based on the names of the library and the model, and will be broken if either is renamed. To avoid errors, treat the model and library as a single unit, and do not rename either.

---

3 Save the model.

4 Right-click on the configurable subsystem in the model. A context menu appears. Select the **Block choice** menu item and observe the block choice submenu. This figure shows the default block choice selection.



5 From the **Block choice** submenu of the context menu, select `Fuel Rate Controller (PIL)`.

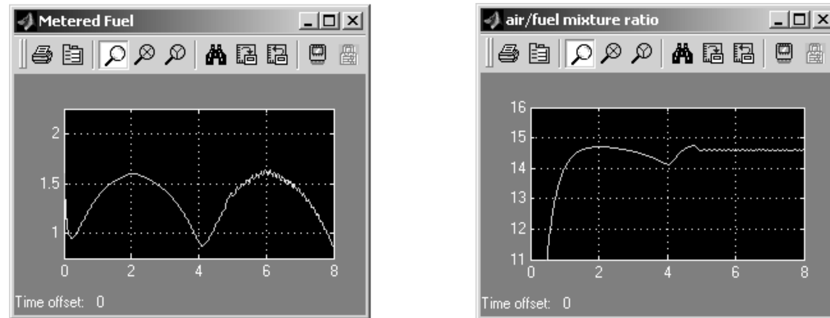
6 Open the model's two Scope blocks, if they are not already opened.

7 Make sure that Simulink is in `Normal` mode. For more information, see the Simulink documentation on Simulation Modes.

8 You are now ready to run the simulation. To start the simulation, click the **Start simulation** button in the Simulink toolbar.

   The target system now starts executing the controller code. Observe that the output signals computed on the target are displayed on the scopes. The updating of the Scope blocks is slow, relative to a normal simulation, because data is transmitted over the serial line on every model step.

9 When the simulation completes, the signals displayed on the scopes should appear as shown in Signals Displayed at End of Simulation or Simulation on page 49-96.



Signals Displayed at End of Simulation or Simulation

10 When the simulation has completed, or has stopped or paused, the target code enters a wait state until it receives a command to start (or resume) from the host. Restart the simulation by clicking the **Start simulation** button again. You can start, stop, restart, pause, or continue a simulation exactly as you would a normal simulation. Try each of these operations a few times.

Once your target has been reset, your application will be lost from memory. In this case, you can download the application again by using the **Download Control Panel** from the **Start** menu. Select the `*.s19` file. In this case it will be `fuel_ram.s19`.

See "Build Process Files and Folders" on page 49-100 for information on the files and folders created by the build process.

Modifying the Controller Subsystem

Typically during algorithm development you will wish to make modifications to the Controller Subsystem. You can apply your modifications to the Controller Subsystem by changing the original model.

Note that in the `mpc555_fuelsys` demo model the Controller Subsystem is actually a Simulink library block from the `mpc555_fuelsys_project` library, so making modifications may require modification of the library block.

Once you have completed making your modifications to the Controller Subsystem you can go back to step "Building PIL and Simulation Components" in this tutorial to rebuild and download the Controller Subsystem for PIL.

Tutorial 2: Using the Demo Model in Simulation

- "Closed-Loop Simulation" on page 49-97
- "SIL Simulation" on page 49-97

Closed-Loop Simulation

In this section, you will continue to use the configurable subsystem in the demo model, using it first in a normal closed-loop simulation and then in a SIL simulation.

1. Right-click on the configurable subsystem and select `Fuel Rate Controller` from the **Block choice** submenu of the context menu. This selects the controller subsystem that was used in the original model.

2. Open the Scope blocks and start the simulation. When the simulation completes (simulation time is set to 8 seconds), the signals displayed on the scopes should appear identical to those displayed during the previous simulation (see Signals Displayed at End of Simulation or Simulation on page 49-96).

SIL Simulation

1. Right-click on the configurable subsystem and select `Fuel Rate Controller (SIL)` from the **Block choice** submenu of the context menu.

   Selecting this option directs the Simulink application to call a generated wrapper S-function that implements the controller algorithm in highly efficient generated code. You can now run a SIL simulation.

2. Start the simulation. You will notice that the simulation completes much more quickly, due to the efficiency of the generated code. Also, observe that the generated code displays results, on the scopes, that are identical to the previous simulation and simulation (see Signals Displayed at End of Simulation or Simulation on page 49-96).

PIL Target Summary

- "Code Generation Options" on page 49-98
- "Build Process Files and Folders" on page 49-100
- "Restrictions" on page 49-101

Code Generation Options

The PIL target is an extension of the embedded real-time (ERT) target configuration. The PIL target inherits the code generation options of the ERT target, as well as the general code generation options. These options are available under **Code Generation**, in the tree on the **Configuration Parameters** dialog box.

Some code generation options of the ERT target are not relevant to the PIL target, and are either unsupported, or restricted in their operation, by the PIL target. See "Restrictions" on page 49-101 for details.

Target-Specific Options. The PIL target has four target-specific code generation options: **Optimize compiler for**, **Compiler optimization switches**, **Build action** and **Use prebuilt (static) libraries**. To view or change the setting of these options, select `ET MPC5xx (processor-in-the-loop) options` under **Code Generation** in the tree on the **Configuration Parameters** dialog.

- **Optimize compiler for** — Select `speed`, `size`, `debug`, or `custom`.

  This option controls compiler optimization switches used during the build process. The exact effect of the optimization switches depends on whether you are using the Wind River or CodeWarrior compiler. You can optimize for performance by choosing the `speed`, `size`, or `debug` options, or define your own (the `custom` option). You can edit these preferences here in the **Compiler optimization switches** edit box if you want to apply changes to the current model (**Optimize compiler for:** will change to `custom`). You can also edit the defaults for these settings in the **Target Preferences** dialog if you want to apply these changes to several models. See "Compiler Optimization Switches" on page 49-14 for more information.

- The **Build action** menu has two options that control what action the PIL target takes after completing the code generation process:

  - `Launch_Download_Control_Panel`: When this option is selected, the PIL target automatically invokes the **Download Control Panel**. When you click **Start Download** the PIL target downloads the generated code to the target board and begins execution of the code.

    Before using this option, make sure that the target preferences (Compiler and Debugger paths) are set correctly.

  - `None`: When this option is selected, the PIL target does not take any action after code generation completes. To download and run your application, you must do so manually, using your development tools.

  - `Run_via_BDM` — on completion of code generation download over BDM connection automatically starts and on completion the code is run.

  - `Debug_via_BDM` — on completion of code generation download over BDM connection automatically starts. When the download is complete the code stops at the first line while debugging, so you can step through the code.

- **Use prebuilt (static) libraries**

  This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time.

Manual Download. Once a subsystem has been built using the PIL target, it is possible to use the **Download Control Panel** to manually download the generated code to the target without repeating the entire build process. To do this, use the following procedure:

1 .Enter `embedded_target_download` in the Command Window to open the Download Control Panel dialog box.

2 Select the required `*.s19` file, and click **Start Download**.

Build Process Files and Folders

The PIL target creates the following in your working folder:

- A build folder, containing generated source code, object files in their own folder, and a makefile and other control files. The build folder also may contain subfolders used by Stateflow software and by the HTML code generation report generator (see "HTML Code Analysis (RAM/ROM) Report" on page 49-104).

  The naming convention for the build folder is *`source`*`_mpc555pil`, where *`source`* is the first word of the generating subsystem or model. For example, the `Fuel Rate Controller` subsystem used in the PIL tutorials generates the build folder `fuel_mpc555pil`.

- The generated library, *`source`*`_lib.mdl`, and the`.mexw32` components that are bound to the generated PIL and SIL blocks in the library. Note that if you rebuild *`source`*`_lib.mdl` in the same working folder, a revision number is appended to the *`source`* string. For example, building from the `Fuel Rate Controller` subsystem used in the PIL tutorials generates `Fuel_lib.mdl`, `fuel1_lib.mdl`, `fuel2_lib.mdl`... `fuel`*`n`*`_lib.mdl`.

- Executable PIL code in a format suitable for downloading to the target and execution by your development system (Wind River Systems SingleStep or CodeWarrior).

- Project files, debugging symbol files, link maps, and other files specific to your development system (Wind River Systems SingleStep or CodeWarrior).

If you do not select the `Launch_Download_Control_Panel` option when you generate code (or if you want to rerun PIL code after it is built), you can use

the **Download Control Panel** to manually download and run the generated executable. To do this, see "Manual Download" on page 49-100.

Restrictions

Please note the following restrictions on the use of the PIL target:

- The PIL target does not support code generation from device driver blocks from the block libraries. Do not include device driver blocks in your PIL models. See `mpc555_fuelsys_project.mdl` for an example of PIL modeling. This example manages multi-model modeling to deal with RT & PIL operation.
- Do not include To File blocks in your PIL models, they will cause the build to fail.
- Self modifying blocks (such as the Resource Configuration block and other blocks) that modify the PIL subsystem during simulation, may cause an error during simulation of the generated Configurable Subsystem (in original subsystem mode).

  As a workaround it is possible to set the MaskSelfModifiable parameter of the original subsystem in the generated PIL library. To do this select the original subsystem in the generated PIL library with the mouse, and then run the following command in the MATLAB command prompt:

  ```
  set_param (gcb, 'MaskSelfModifiable', 'on')
  ```

  Note that we recommend not placing driver blocks (such as the Resource Configuration block) inside the PIL subsystem.
- If you change the cross-compiler you use with the PIL target (from Wind River to CodeWarrior or vice versa), you should rebuild your PIL models in a clean folder, or delete all files from the models' code generation folders. The PIL build process expects to start with a clean folder, or a folder created in the process of building with the same compiler. Leftover components built by a different compiler cause errors.
- In a plant/controller simulation where the controller is built via the PIL target, the plant model can contain any Simulink blocks, including a combination of continuous-time and discrete-time blocks. However, the controller subsystem must not include any continuous-time blocks. This is because PIL uses the S-function Generation feature; this feature does

not support continuous sample times. However, note that, standard code generation, as used by the MPC555 RT target, does support continuous-time blocks.

- The superseded version of the Vector CAN Configuration block should not be placed inside a PIL subsystem. Instead, the model can be updated to use the current Vector CAN Configuration block, which can be placed inside a PIL subsystem.
- Parameters with the following storage requirement are not supported for PIL. If a model contains parameters where the storage class (e.g., custom storage class) of the data objects requires storage in the `model.c` module, then "unresolved external symbol" link errors occur during the processor-in-the-loop (PIL) build process.
- Model folders must be located either an actual hard drive on your PC, or a mapped drive. Do not use a UNC network path. If you run (simulate) a model from a Universal Naming Convention (UNC) network folder (such as \\Server\user\work), errors are produced.
- Vectors are not supported at the PIL boundary.
- Nonvirtual busses are not supported at the PIL subsystem boundary. PIL only supports virtual buses at the PIL boundary. Note: A virtual bus at a root level inport, with properties specified via a bus object, is treated as a nonvirtual bus. To avoid an error, make the inport a virtual bus.
- Certain ERT code generation options are not supported by the PIL target. If these options are selected, the PIL target either ignores the option or issues an error message during the build process. PIL Target Restricted Code Generation Options on page 49-102 summarizes these restricted options.

PIL Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Generate ASAP2 file	Ignored if selected; build process proceeds
External mode	Error if selected; build process terminates
Generate an example main program	This option should not be selected for the PIL target.

PIL Target Restricted Code Generation Options (Continued)

Option	Restriction
Generate reusable code	Error if selected; build process terminates
Target function library	`C89/C90(ANSI)` is the default and is not configurable.

Algorithm Export Target

The Algorithm Export (AE) target is an aid to code analysis and interfacing. The target generates only the code that implements the algorithm of your model or subsystem, without any overhead for PIL host/target communications or other operations extraneous to the model. Such purely algorithmic code is easier to interface to your manually written or legacy code than code generated by the PIL or RT targets.

Another application of the AE target is to use it to produce a code generation report. Since only model code is included, you can more easily analyze the code generated from your model.

The AE target supports both the CodeWarrior and Wind River cross-compilers, as specified in your target preferences (see "Setting Target Preferences for MPC5xx" on page 49-11).

To use the AE target,

1 Select **Configuration Parameters** from the **Simulation** menu. The **Configuration Parameters** dialog opens.

2 Select **Code Generation** in the tree.

3 In the **Target selection** pane, click on the **Browse** button to open the **System Target File Browser**. In the browser, select `)mpc555exp.tlc`. Click **OK** to close the browser and return to the **Configuration Parameters** dialog.

4 Select **Templates** in the tree and make sure **Generate an example main program** is not selected.

5 Follow the usual procedure for generating code from your model or subsystem.

We recommend using the AE target in conjunction with the HTML code generation report (see “HTML Code Analysis (RAM/ROM) Report” on page 49-104). If you select the **Create Code Generation report** option as described in the next section, you can view a profiling report that includes detailed itemization of RAM and ROM usage for all code and data sections, and a complete memory map of the generated code. You can also easily examine the generated code via hyperlinks in the code generation report.

HTML Code Analysis (RAM/ROM) Report

The coder product supports an extended version of the HTML code generation report. You can generate reports for the real-time target as well as the processor-in-the-loop (PIL) target and algorithm export (AE) target.

The extended code generation report includes an additional section, the **Code profile report** for the generated application. See the product help for information on the other report sections.

The code profile report section includes a detailed itemization of RAM and ROM usage for all code and data sections, and a complete memory map of the generated code. The report is generated from the memory map file (MAP file) created during the application build process (compilation and linking). This MAP file is appended to the end of the report to provide the complete details of the memory layout of the application.

An example Code Profile Report is shown below.

To generate a code generation report and view the profiling report,

1 On the **Code Generation** options in the **Configuration Parameters** dialog, make sure that the **Generate code only** option is not selected.

   The reason for this step is that the extended code generation report obtains information from MAP files that are created by your cross-compiler during the build process. If the **Generate code only** option is on, these files are not generated, which prevents the generation of the code generation report.

2. Select **Report** in the tree, and select the check box **Create Code Generation report**.

3. Follow the usual procedure for generating code from your model or subsystem.

   The code generation report file is placed in the build folder. The file is named *`model`*`_codegen_rpt.html` or *`subsystem`*`_codegen_rpt.html`.

   The MATLAB Help browser automatically opens and displays the code generation report. Alternatively, you can view the code generation report in your Web browser.

4. To view the profiling report, click on the **Code profile report** link in the **Contents** pane of the report.

A portion of an example code profile report is shown following. The raw memory map (MAP file) is at the bottom of the report.

Back Forward

Contents

Summary
List of eliminated blocks
Subsystem Report
Code profile report

Generated Source Files

Target_LED.c
mpc555_main.c
Target_LED.h
Target_LED_private.h
Target_LED_types.h
profile_vars.h
rtwtypes.h

Code Profile Report

Compiler: Diab

- Model Memory Summary
- Entire Memory Summary
- Memory MAP Summary
- Model Memory Detail
- Entire Memory Detail
- Memory MAP
- Original MAP file

Model Memory Summary

Memory Regions	Size [in bytes]
Read/Write Memory Usage(without rtwlib)	32
Read Only Memory Usage(without rtwlib)	640
Read/Write Memory Usage(with rtwlib)	32
Read Only Memory Usage(with rtwlib)	640

Entire Memory Summary

Memory Regions and Section Mapping	Size [in bytes]
Read/Write Memory Usage (.bss .sbss .sdata .data)	464
Read Only Memory Usage (.text .sdata2 .application_bios .ram_entry)	3706

Memory MAP Summary: RAM Target :: Off-chip RAM, On-chip RAM

Memory Type and Section Mapping	Size [in bytes]
OFF-CHIP_RAM (.ram_entry .text .sdata2 .data .sdata .sbss .bss .application_bios)	4184
ON-CHIP_RAM (.application_bios)	120

Model Memory Detail

Read/Write Memory Usage (with rtwlib)	File	Section	Size [in bytes]
Target_LED_Y	[COMMON]	.bss	16
Target_LED_DWork	bin/DIAB/555/Target_LED.o	.sbss	8
Target_LED_M_	bin/DIAB/555/Target_LED.o	.sbss	4
Target_LED_M	bin/DIAB/555/Target_LED.o	.sdata	4
Read Only Memory Usage (with rtwlib)	**File**	**Section**	**Size [in bytes]**
Target_LED_initialize	bin/DIAB/555/Target_LED.o	.text	312
Target_LED_step	bin/DIAB/555/Target_LED.o	.text	328

Algorithm Export Target Summary

- "Code Generation Options" on page 49-106
- "Restrictions" on page 49-107

Code Generation Options

The Algorithm Export (AE) target is an extension of the embedded real-time (ERT) target configuration. The AE target inherits the code generation options of the ERT target, as well as the general code generation options. These options are available under **Code Generation**, in the tree on the

Configuration Parameters dialog box; they are documented in the product help.

Some code generation options of the ERT target are not relevant to the AE target, and are either unsupported, or restricted in their operation, by the AE target. See “Restrictions” on page 49-107 below for details.

The only target-specific option for AE target is **Use prebuilt (static) libraries**. This check box option (selected by default) saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time.

Restrictions

Certain ERT code generation options are not supported by the AE target. If these options are selected, the AE target either ignores the option or issues an error message during the build process. AE Target Restricted Code Generation Options on page 49-108 summarizes these restricted options.

AE Target Restricted Code Generation Options

Option	Restriction
MAT-file logging	Ignored if selected; build process proceeds
Block type	Error if `SIL` is selected; build process terminates
Generate ASAP2 file	Ignored if selected; build process proceeds
External mode	Error if selected; build process terminates

You must not include driver blocks in your model for Algorithm Export. The AE target is designed to generate only the code that implements the algorithm of your model or subsystem, without any overhead for PIL host/target communications or other operations extraneous to the model, so you should not be including driver blocks.

Configuration Parameters

In this section...
"Code Generation Pane: ET MPC5xx (Algorithm Export) Options" on page 49-109
"Code Generation Pane: ET MPC5xx (Processor-in-the-Loop) Options" on page 49-111
"Code Generation Pane: ET MPC5xx Real-Time Options (1)" on page 49-115
"Code Generation Pane: ET MPC5xx Real-Time Options (2)" on page 49-119

Code Generation Pane: ET MPC5xx (Algorithm Export) Options

- "ET MPC5xx (Algorithm Export) Options Tab Overview" on page 49-109
- "Use prebuilt (static) libraries" on page 49-111

ET MPC5xx (Algorithm Export) Options Tab Overview

Control recompiling of libraries for faster build times.

Configuration. This pane appears only if you specify the `mpc555exp.tlc` system target file.

Tips.

- The Algorithm Export (AE) target generates only the code that implements the algorithm of your model or subsystem. This is useful for code analysis and interfacing to manually written or legacy code.
- Use the HTML code generation report to view a profiling report that includes detailed itemization of RAM and ROM usage for all code and data sections, and a complete memory map of the generated code. You can also easily examine the generated code via hyperlinks in the code generation report.

  The **Code profile report** is an additional section in the HTML Code Analysis (RAM/ROM) Report. To generate the report,

1. On the Code Generation General pane, make sure **Generate code only** is not selected.
2. On the Code Generation Report pane, select **Create Code Generation report**.

To get help on an option.

1. Right-click the option's text label.
2. Select **What's This** from the popup menu.

See Also.

- Algorithm Export Target
- HTML Code Analysis (RAM/ROM) Report

Use prebuilt (static) libraries

Use prebuilt `rtwlib` for faster build time.

Settings. **Default:** On

On

Use prebuilt libraries. This saves a considerable amount of time during the build process, as the libraries do not need to be recompiled every time.

Off

Recompile libraries and do not use prebuilt libraries.

Command-Line Information.

Parameter: `STATIC_RTWLIB`
Type: string
Value: `'on'` | `'off'`
Default: `'on'`

See Also. Algorithm Export Code Generation Options

Code Generation Pane: ET MPC5xx (Processor-in-the-Loop) Options

- "ET MPC5xx (Processor-in-the-Loop) Options Tab Overview" on page 49-111
- "Optimize compiler for" on page 49-113
- "Compiler optimization switches" on page 49-113
- "Build action" on page 49-115

ET MPC5xx (Processor-in-the-Loop) Options Tab Overview

Specify compiler and build action code generation options for processor-in-the-loop.

Configuration. This pane appears only if you specify the `mpc555pil.tlc` system target file.

To get help on an option.

1 Right-click the option's text label.

2 Select **What's This** from the popup menu.

See Also.

- Processor-in-the-Loop Code Generation Options
- Overview of PIL Simulation

Optimize compiler for

Choose whether to optimize C compiler settings for fastest execution speed, smallest code size, debugging, or custom settings.

Settings. Default: `speed`

`speed`
: Optimize C compiler settings to minimize execution time.

`size`
: Optimize C compiler settings to minimize code size.

`debug`
: Optimize C compiler settings for debugging.

`custom`
: Define your own optimization switches.

Tip. The exact effect of the optimization switches depends on whether you are using the Wind River or CodeWarrior compiler. Consult your compiler documentation for specific optimizations.

Dependency. Setting this parameter changes the **Compiler optimization switches** to the appropriate switches, which depend on the compiler used and may be user-defined.

Command-Line Information.

Parameter: `MPC555_OPTIMIZATION_SWITCH`
Type: string
Value: `'speed' | 'size' | 'debug' | 'custom'`
Default: `'speed'`

See Also. Processor-in-the-Loop Code Generation Options

Compiler optimization switches

Observe or edit compiler optimization switches.

Settings. Default: Depends on the toolchain, and also customizable — there are many possibilities if you modify target preferences.

Tip. To apply changes to the current model only, you can edit the switches in the Compiler optimization switches edit box (**Optimize compiler for**: changes to custom). If you want to apply these changes to several models, you can edit the defaults for these settings in the Target Preferences dialog box.

Dependency. Editing this parameter changes the **Optimize compiler for** to custom.

Command-Line Information.

Parameter: MPC555_OPTIMIZATION_FLAGS
Type: string
Value: customizable — there are many possibilities if you modify target preferences or make custom edits
Default: depends on toolchain.

See Also. Processor-in-the-Loop Code Generation Options

Build action

Choose action to perform after build process completes.

Settings. **Default:** `None`

`None`
: No action after code generation.

`Launch_Download_Control_Panel`
: Launch Download Control Panel utility on completion of code generation.

`Run_via_BDM`
: Download over BDM connection automatically starts on completion of code generation. When the download is complete the code is run.

`Debug_via_BDM`
: Download over BDM connection automatically starts on completion of code generation. When the download is complete the code stops at the first line while debugging, so you can step through the code.

Command-Line Information.

Parameter: `BuildAction`
Type: string
Value: `'None'` | `'Launch_Download_Control_Panel'` | `'Run_via_BDM'` | `'Debug_via_BDM'`
Default: `'None'`

See Also. Processor-in-the-Loop Code Generation Options

Code Generation Pane: ET MPC5xx Real-Time Options (1)

- "ET MPC5xx Real-Time Options (1) Tab Overview" on page 49-116
- "Target Memory Model" on page 49-117

ET MPC5xx Real-Time Options (1) Tab Overview

Specify compiler and build action code generation options for real-time standalone execution.

Configuration. This pane appears only if you specify the `mpc555rt.tlc` or `mpc555rt_grt.tlc` system target file.

To get help on an option.

1 Right-click the option's text label.

2 Select **What's This** from the popup menu.

See Also.

- Real-Time Code Generation Options
- Generating Stand-Alone Real-Time Applications

Target Memory Model

Select either FLASH or RAM.

Settings. **Default:** `RAM`

`RAM`
: Generate files in a format suitable for downloading into external RAM.

`FLASH`
: Generate files in a format suitable for downloading into the MPC555 on-chip flash memory.

In both cases these two files are generated, with this naming convention:

- *model*_flash.s19 or *model*_ram.s19 — code only, for CAN download
- *model*_flash.elf or *model*_ram.elf — for BDM download, containing code and optional debugging symbols if you choose a `debug` build in the **Optimize compiler for** settings.

Tips.

- Loading the application code into RAM is faster than loading it into flash memory. In addition, by using RAM you can avoid using up the programming cycles of the flash memory; this lengthens the usable lifetime of the flash memory. Running the application from RAM is a good option for initial testing of the application.
- The MPC5xx flash memory has a limited lifetime, which is shortened each time the flash memory is programmed. To extend product life, Freescale recommends using flash programming only when necessary.
- To program applications into RAM, your target hardware must have additional RAM external to the MPC555 on-chip RAM. The coder product does not support downloading of code to MPC5xx on-chip RAM, because the MPC555 has only 26K of on-chip RAM and the MPC565 has 36K.

- For final deployment, or to load code onto a test board for use at a test site, you will generally want to program your code into the nonvolatile flash memory. 416K of flash memory is available for application code (992K on the MPC565). Code programmed into flash memory is persistent and restarts when the board is powered on.

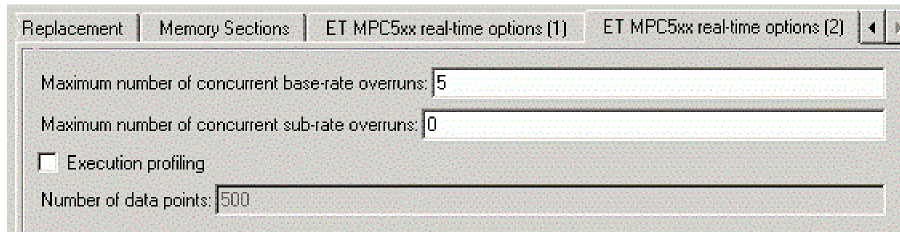
Command-Line Information.

Parameter: `TARGET_MEMORY_MODEL`
Type: string
Value: `'RAM' | 'FLASH'`
Default: `'RAM'`

See Also.

- RAM vs. Flash Memory
- Overview of Memory Organization and the Boot Process

Code Generation Pane: ET MPC5xx Real-Time Options (2)



- "ET MPC5xx Real-Time Options (2) Tab Overview" on page 49-119
- "Maximum number of concurrent base-rate overruns" on page 49-120
- "Maximum number of concurrent sub-rate overruns" on page 49-121
- "Execution profiling" on page 49-122
- "Number of data points" on page 49-122

ET MPC5xx Real-Time Options (2) Tab Overview

Control execution profiling and scheduling overrun behavior.

Configuration. This pane appears only if you specify the `mpc555rt.tlc` or `mpc555rt_grt.tlc` system target file.

To get help on an option.

1 Right-click the option's text label.

2 Select **What's This** from the popup menu.

See Also.

- MPC5xx Options for Execution Profiling

- Execution Profiling

Maximum number of concurrent base-rate overruns

Configure allowable base-rate overruns.

Settings. **Default:** 5

Minimum: 0

Maximum: No maximum value — it depends on available memory.

Tips.

- Use this option to configure the behavior of the scheduler when timer based tasks do not complete within their allowed sample time.
- It is useful to allow task overruns in the case where a task may occasionally take longer than usual to complete (e.g. if extra processing is required when a special event occurs); if the task overrun is only occasional then it is possible for the scheduler to 'catch up' after the extra processing has been completed.
- If the maximum number of concurrent overruns for any task is exceeded, this is deemed to be a failure and the real-time application is stopped. This in turn will result in a watchdog timer timeout and the processor will be reset.
- The occurrence of base-rate overruns does not affect the numerical behavior of the algorithm (although reading/writing external devices will of course be delayed).

Command-Line Information.

Parameter: `BaseRateMaxOverrunsValue`
Type: int
Value: `0 | 1 | 2...`
Default: `5`

See Also. MPC5xx Options for Execution Profiling

Maximum number of concurrent sub-rate overruns

Configure allowable sub-rate overruns.

Settings. Default: 0

Minimum: 0

Maximum: No maximum value — it depends on available memory.

Tips.

- If this option is set to a value greater than zero, then the behavior of any Rate-Transition blocks may be affected. Specifically, if the model contains a Rate Transition block where the option "Ensure deterministic data transfer (maximum delay)" is selected, then this setting may not be honored.
- If sub-rate overruns are allowed then the transfer of data between different rates (via rate-transition blocks) in the model may be affected; this causes the numerical behavior in real-time to differ from the behavior in simulation. To see an illustration of this effect try running the demo model `mpc555rt_multitasking`. To disallow sub-rate overruns and ensure that this effect does not occur, you should set **Maximum number of concurrent sub-rate overruns** to zero.

Command-Line Information.

Parameter: `SubRateMaxOverrunsValue`
Type: int
Value: `0 | 1 | 2...`
Default: `0`

See Also. MPC5xx Options for Execution Profiling

Execution profiling

Specify whether to configure code for execution profiling.

Settings. **Default:** Off

On

Include function calls in the generated code for the model at the beginning and end of each task or asynchronous Interrupt Service Routine (ISR) to be profiled. When you perform an execution profiling run, these function calls read a timer and log this reading, along with a task identifier, for uploading and analyzing.

Off

Do not add function calls for execution profiling.

Tip. When code for the model is generated, these function calls update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous worst case value is exceeded. Additionally, when a trigger is provided, data can be logged over a period of time to record all task start and task finish times. The trigger signal can be supplied by the execution profiling blocks.

See Also.

- Execution Profiling
- MPC5xx Options for Execution Profiling

Number of data points

Specify number of data points to log for execution profiling runs.

Settings. **Default:** 500

Minimum: This depends on the number of tasks. Three is a sensible minimum to get useful information back.

Maximum: No maximum value - it depends on available memory.

Tip. When a snapshot of task and ISR activity is logged this data is stored in memory that is statically allocated at build time. Each data point requires 8 bytes on the MPC555. The larger the number of data points to be stored, the more RAM that must be reserved for this purpose. At the end of a logging run, the data must be uploaded to the host computer for analysis; this is typically achieved by using the execution profiling blocks.

Command-Line Information.

Parameter: `ExecutionProfilingNumSamples`
Type: int
Value: `3 | 4 | 5...`
Default: `500`

See Also.

- Mpc5xx Options for Execution Profiling
- Execution Profiling

Toolchains and Hardware

This section discusses specific settings for different cross-development environments:

In this section...

"Setting Up Your Toolchain" on page 49-124

"Setting Up Your Installation with Wind River Compiler and Wind River Systems SingleStep Debugger" on page 49-124

"Setting Up Your Installation with Freescale CodeWarrior" on page 49-129

"Setting Up Your Target Hardware" on page 49-133

"CAN Hardware and Drivers" on page 49-139

"Configuration for Nondefault Hardware" on page 49-141

"Integrating External Blocksets" on page 49-144

Setting Up Your Toolchain

The currently supported toolchains are WindRiver (Wind River Compiler and Wind River Systems SingleStep) and Freescale CodeWarrior. You must first install and configure your toolchain to work with the coder product. The necessary steps are described in the following sections:

- "Setting Up Your Installation with Wind River Compiler and Wind River Systems SingleStep Debugger" on page 49-124
- "Setting Up Your Installation with Freescale CodeWarrior" on page 49-129

Setting Up Your Installation with Wind River Compiler and Wind River Systems SingleStep Debugger

- "Required Hardware and Software" on page 49-125
- "Procedure" on page 49-125

Required Hardware and Software

To use the coder product with the Wind River Compiler, you need the following:

- An MPC5xx development board (such as the phyCORE-MPC555 development board, or an Axiom board) and a debugger connector (such as the WindRiver visionPROBE or the BDM Wiggler from Macraigor Systems). Note the phyCORE-MPC555 board comes with built-in debugger connector into which you can directly plug a parallel port connector, in which case you may not require a BDM connector. See "Setting Up Your Target Hardware" on page 49-133.
- Wind River Systems Wind River Compiler and Wind River Systems SingleStep debugger, as detailed in "Supported Cross-Development Tools for Freescale MPC5xx" on page 49-9.

Procedure

- "Install Wind River Compiler" on page 49-125
- "Install Wind River Systems SingleStep Debugger" on page 49-126
- "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126
- "Initialize visionPROBE" on page 49-128
- "Configure MPC5xx Jumpers" on page 49-128

Install Wind River Compiler. If you have not already done so, install the Wind River Compiler, following the installation instructions provided by Wind River Systems.

You do not need to set a default processor or other compiler defaults. During the code generation and build process, the coder product will generate a makefile that sets the correct options.

You will need to note the path to the installed compiler in order to configure your target preferences (see "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126).

Install Wind River Systems SingleStep Debugger. The Wind River Systems SingleStep debugger, in conjunction with the coder product, lets you download, run and debug generated code.

Follow the instructions of the Wind River Systems SingleStep installer.

To resolve questions or difficulties with Wind River Systems SingleStep, refer to the documentation, or contact Wind River Systems.

You will need to note the path to the installed Wind River Systems SingleStep debugger in order to configure your target preferences (see "Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep" on page 49-126).

Setting Target Preferences for Wind River Compiler and Wind River Systems SingleStep. After installing your development tools, the next step is to configure your target preferences for the Wind River Compiler and Wind River Systems SingleStep debugger (read "Setting Target Preferences for MPC5xx" on page 49-11, if you have not yet done so).

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window.

2. Select `Target Preferences` from the drop-down list, and click **OK**. This opens the Target Preferences dialog box, which allows you to edit settings for your cross-development environment.

3. Select `Diab` from the **Toolchain** menu (the Wind River Compiler was formerly known as Diab).

4. Expand the `ToolChainOptions` by clicking the plus sign, and type the correct path into **CompilerPath**. For example `"d:\applications\WindRiver\4.3g"`.

5. For Wind River Systems SingleStep you must also type the correct path into **DebuggerPath**. For example `"d:\applications\sds"`.

6. The defaults for **DebuggerSwitches** and **DebuggerExecutable** are set up for use of Wind River Systems SingleStep (using a visionPROBE BDM connection). You may need to change `LPT1` to whatever port you connect to.

Note, once you have set target preferences, you must initialize the device. See "Initialize visionPROBE" on page 49-128.

7 To use any other BDM device than the visionPROBE (such as the Wiggler, Raven/Blackbird or OnBoard BDM with Wind River Systems SingleStep), you must change two target preferences from the defaults:

   a Change the **DebuggerSwitches** target preference to the following:

   ```
   -g -V mpc555 -r - -p LPT1=1
   ```

   If necessary you can change `LPT1` to whatever port you connect the probe to.

   b Change the **DebuggerExecutable** from the default to:

   ```
   bdmp58.exe
   ```

ToolChainOptions	mpc555.DiabOptions
CompilerOptimizationSwitches	mpc555.CompilerOptimizationSwitches
Debug	-g
Size	-XO -Xsize-opt
Speed	-XO
CompilerPath	d:\applications\diab\4.3g
DebuggerExecutable	bdmp58.exe
DebuggerPath	d:\applications\sds
DebuggerSwitches	-g -V mpc555 -r - -p LPT1=1

The **DebuggerSwitches** target preference is specific to Wind River Systems SingleStep. If you want to change the default debug settings, type

```
help debug
```

at the Wind River Systems SingleStep command line to see the options available. For example you can change parallel port here. The default is `-p LPT1=1` which specifies port 1 on your host PC at speed 1. You could change it to `-p LPT2=2` to specify port 2 at speed 2.

Other debugger executables are supplied with Wind River Systems SingleStep — if you want to change the defaults to use a different connection device and different debug settings, consult the Wind River Systems SingleStep documentation.

Note that the path to the Wind River Systems SingleStep debugger, specified in `DebuggerPath` in the Target Preference GUI, is the root folder of your Wind River Systems SingleStep installation, on either an actual hard drive on your PC, or a mapped drive. Do not use a Universal Naming Convention (UNC) path. For most purposes, the other target preferences fields can be left at their defaults. Once you have set these target preferences, the build process will automatically invoke your compiler and debugger when required for downloading code.

Initialize visionPROBE. Before using the visionPROBE (and after setting target preferences), you must initialize the device:

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window.

2. Select `Initialize visionPROBE for Selected Target Board (WindRiver Only)` from the drop-down list, and click **OK**.

You mustcarry out the initialization procedure again if you change the target processor..

Note that for the visionPROBE, you must configure the parallel port BIOS settings as follows:

- ECP mode
- Enabled (as opposed to Auto)
- IRQ and address of the parallel port specified in the BIOS must match that in the visionPROBE `comdll.cfg` file - edit the `cfg` file if necessary. Default parallel port I/O address = 0x378; IRQ=7, communicating over PAR1 (LPT1).

Configure MPC5xx Jumpers. Make sure that the jumpers on the MPC5xx board are set as described in "Jumper Settings" on page 49-134. The correct jumper configuration is required when downloading to flash memory. Any other jumper settings may cause downloading to flash memory to fail, or cause other problems when operating with the coder product. For additional information on jumper settings, consult the MPC5xx documentation and the Wind River Systems SingleStep manual.

The next step is to verify your installation:

1. You can download and run the test program supplied. See "Run Test Program" on page 49-15.

2. You must then follow the instructions to download boot code ("Download Boot Code to Flash Memory" on page 49-16). Once you have completed these steps, you can begin working with the coder product.

Setting Up Your Installation with Freescale CodeWarrior

- "Required Hardware and Software" on page 49-129
- "Procedure" on page 49-129
- "Limitations" on page 49-132

Required Hardware and Software

To use the coder product with Freescale CodeWarrior, you need the following:

- An MPC5xx development board (such as the phyCORE-MPC555 development board) and a debugger connector (such as the BDM Wiggler from Macraigor Systems). Note the phyCORE-MPC555 board comes with built-in debugger connector which you can plug a parallel port connector into directly, in which case you may not require a BDM connector.
- Freescale CodeWarrior Development Studio, MPC5xx Edition, as detailed in "Supported Cross-Development Tools for Freescale MPC5xx" on page 49-9.

Procedure

- "Install Freescale CodeWarrior IDE" on page 49-130
- "Configure Freescale CodeWarrior Debugger" on page 49-130
- "Set Target Preferences for CodeWarrior" on page 49-131
- "Configure MPC5xx Jumpers" on page 49-132

Install Freescale CodeWarrior IDE. The first step is to install the Freescale CodeWarrior IDE:

1 If you have previously installed an older version of Freescale CodeWarrior for Embedded PowerPC , uninstall it.

2 Install Freescale CodeWarrior Development Studio, MPC5xx Edition, v8.7 using the setup program provided on your Freescale CodeWarrior CD (or on your network). Run `Setup.exe` and follow the prompts.

3 Open CodeWarrior IDE. You can use the Windows Start menu (**Start > Programs > CodeWarrior > CodeWarrior IDE**).

4 Select **Edit > Preferences > Build Settings > Build Before Running**

5 Select the option **Never** and click **Apply**.

   It is vital you set this to avoid errors when building and automatically downloading code with the coder product.

Configure Freescale CodeWarrior Debugger. The next step is to configure the CodeWarrior debugger to communicate with the MPC5xx board over the parallel port:

1 From the Freescale CodeWarrior IDE, select the **Edit** menu, and open the **IDE Preferences** dialog box. In the **IDE Preference Panels** pane, click on the plus sign next to **Debugger**.

2 A list of choices opens below **Debugger**. Select `Remote Connections`. The **Remote Connections** panel is displayed on the right.

3 If no `MPC555DK Wiggler` configuration exists, create one as follows:

   a Click the **Add**... button. The **New Connection** configuration dialog box opens. Set the **Name** property to `MPC555DK Wiggler`.

   b If you are using a Raven or Blackbird BDM device, set the **Debugger** property to `EPPC - MSI BDM Raven`.

   c If you are using a Wiggler or On-Board BDM, set the **Debugger** property to `EPPC MSI Wiggler`.

   d Set the **Connection Type** property to `Parallel`.

e. Set the **Connection Port** property to match the port to which you have connected your MPC5xx board (the default is `LPT1`).

f. Set the **Speed** property to `1`.

g. Set the **FPU Buffer Address** property to 0x3f9800.

h. Click **OK** and skip to step 5.

4. If a `MPC555DK Wiggler` exists, click the **Change** button. The `MPC555DK Wiggler` configuration dialog box opens. By default, the **Parallel Port** property is set to `LPT1`. If you have connected your MPC5xx board to a different port, change the **Parallel Port** setting accordingly. Then click **OK** to close the **MPC555DK Wiggler** configuration dialog box.

5. Click **Apply** and close the **IDE Preferences** dialog box.

Set Target Preferences for CodeWarrior. The next step is to configure your target preferences for Freescale CodeWarrior (read "Setting Target Preferences for MPC5xx" on page 49-11 if you have not yet done so). Follow these steps:

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window.

2. Select `Target Preferences` from the drop-down list, and click **OK**.

   This opens the Target Preferences dialog box, which allows you to edit settings for your cross-development environment.

3. Select `CodeWarrior` from the **Toolchain** menu.

4. Expand the `ToolChainOptions` by clicking the plus sign, and type the correct path into **CompilerPath**. Do not use a UNC path, use only local or mapped drives.

Note that when using CodeWarrior, you do not also have to specify the DebuggerPath, as the compiler and debugger are integrated. When required, the build process will automatically invoke the CodeWarrior debugger.

For most purposes, the other target preferences fields can be left at their defaults.

Note If you have multiple versions of the CodeWarrior IDE installed, the version launched may not be the version you expect. The CodeWarrior IDE is launched using the CodeWarrior COM API, and depends on installation order, not your setting in the Target Preferences. You can correct this problem by running the `regservers.bat` script, which is located in the `bin` folder of your CodeWarrior installation. This registers the correct CodeWarrior application to be launched.

Configure MPC5xx Jumpers. Make sure that the jumpers on the MPC5xx board are set as described in "Jumper Settings" on page 49-134. The correct jumper configuration is required.

The next step is to verify your installation.

1. You can download and run the test program supplied. See "Run Test Program" on page 49-15.

2. You must then follow the instructions to download boot code ("Download Boot Code to Flash Memory" on page 49-16). Once you have completed these steps, you can begin working with the coder product.

Limitations

Limitation with S-functions and the CodeWarrior Compiler. If you try to build a model including a non-inlined generated S-function and are using the CodeWarrior compiler, you see an error like the following during compilation.

```
#   Error:
#   the file 'Subsystem_sf_types.h' cannot be opened
#   (included from:
#       C:\Work\Subsystem_sf.h:12
#       C:\Work\finalSubsystem_mpc555rt\finalSubsystem.h:33
#       C:\Work\finalSubsystem_mpc555rt\mpc555_main.c:17)

Errors caused tool to abort.
matlab\bin\win32\gmake.exe: ***
[bin/CODE_WARRIOR/555/mpc555_main.o] Error 1
```

You can work around this problem by editing the model's configuration parameters. In the Custom Code settings, add the path of the S-function build folder to the list of additional include folders.

Alternatively you can avoid this problem by compiling with the Wind River Systems Wind River Compiler.

Setting Up Your Target Hardware

- "Communications Ports" on page 49-133
- "Jumper Settings" on page 49-134

This section describes the required connections and jumper settings for the following development boards: Phytec phyCORE-MPC555, the Phytec MPC565 and the Axiom MPC555, MPC564, and MPC566.

If you are using other development boards you may need to see "Configuration for Nondefault Hardware" on page 49-141.

Communications Ports

Before you begin working with the coder product, you should set up your target board and connect it to your host computer. For example, the hardware setup is described in the *phyCORE-MPC555 Quickstart Instructions* manual on the Phytec Spectrum CD. See the "Interfacing the phyCORE-MPC555 to a Host PC" section of the "Getting Started" chapter.

In this document, we assume that you have connected your board to the same serial (COM1) and parallel (LPT1) ports described in the *phyCORE-MPC555 Quickstart Instructions*. Note that you must ensure your computer's LPT parallel port for BDM interface is set to EPP mode and Auto (as opposed to Enabled). This is generally a BIOS level configuration. If you are using a visionPROBE you must configure the parallel port as detailed in "Initialize visionPROBE" on page 49-128.

Jumper Settings

Note You MUST check your jumper settings. Do not assume hardware is supplied with jumpers set as documented by the manufacturer. Incorrect operation or even hardware damage may occur if you do not check jumper settings.

Use the following settings for these boards:

- "Phytec MPC555 Jumper Settings" on page 49-134
- "Phytec MPC565 Jumper Settings" on page 49-137
- "Axiom MPC555 Jumper Settings" on page 49-138
- "Axiom MPC564EVB Jumper Settings" on page 49-139
- "Axiom MPC566EVB Jumper Settings" on page 49-139

Phytec MPC555 Jumper Settings. The coder product has been tested by the MathWorks with the Phytec phyCORE-MPC555 board, using the jumper settings indicated in the table below. MathWorks has tested the PCB 1174.0 board. If you are using a PCB 1174.2 board you may also have to alter settings such as jumper 19. Please see your Phytec development board manual for details.

For jumper locations and pin numbers, see Jumper Layout section of the *Development Board for phyCORE-MPC555 Hardware Manual*, `"L-525E.pdf"`.

The following table summarizes the correct jumper settings to use when your host PC is connected to the on-board BDM port, or via visionPROBE, Wiggler, Raven, or Blackbird devices.

Jumper	Description	visionPROBE, Raven or Blackbird	Wiggler	On-Board BDM
JP1	On-board BDM reset signal connection	Open	as Raven	3+4 closed
JP2	Power supply for external BDM	Open (unless BDM device requires supply voltage from development board)	1+2 closed	1+2 closed
JP3	Connect push button to different reset signals	1+2 (/HRESIN connected to push button)	as Raven	as Raven
JP4	Programming of Internal MPC555 Flash internal memory enabled	Closed	as Raven	as Raven
JP5,JP7, JP8,JP9	Jumpers relating to on-board BDM	Open	as Raven	All closed
JP6	See note below.	Open	as Raven	Closed
JP10	Connect one of the LEDs to supply voltage	Closed	as Raven	as Raven
JP11	Connect 5V supply voltage	Closed	as Raven	as Raven
JP12	Connect 3V3 supply voltage	Closed	as Raven	as Raven
JP13	CAN A bus termination	Closed (apply 120 Ohm termination)	as Raven	as Raven
JP14	CAN B bus termination	Closed (apply 120 Ohm termination)	as Raven	as Raven

Jumper	Description	visionPROBE, Raven or Blackbird	Wiggler	On-Board BDM
JP15	Select boot memory	1+2 (boot from internal flash memory)	as Raven	as Raven
JP16	Use J5 as source of Hard-Reset-Configuration	Open	as Raven	as Raven
JP17	Connect /HRESET or /SRESET to external BDM interface logic	1+2 (/HRESET connected to BDM interface logic)	as Raven	as Raven
JP18	Connect interrupt to push button	Default 1+2	as Raven	as Raven
JP19	See note below.			

Note Jumper 6 must be open unless using an On-Board BDM.

When using the On-Board BDM connection, if you then want to run the target stand-alone (disconnected from the debugger) you must also disconnect (open) jumper 6. This only affects the on-Board BDM, all other configurations *always have jumper 6 open*. Use the On-Board BDM settings in the table if you are using the BDM connection for debugging, but remember you must make this change to run stand-alone:

For debugging: Jumper 6 must be closed (target stops while debugging after reset); connect parallel cable to target.

For stand-alone: Jumper 6 must be open (target runs in normal mode after reset); disconnect parallel cable from target.

Note The jumper 19 setting may need to be altered if you are using a PCB 1174.2 board. See the Phytec documentation for more information.

Phytec MPC565 Jumper Settings. These settings are for EXTERNAL BDM device only, NOT On-Board BDM.

Make sure you use the default Phytec documented MPC565 jumper settings and the following additional changes :

General:

JP28: 2-3

JP29: Closed

BDM Related:

JP32: Open

JP33: Open

JP34: Open

JP35: Open

JP36: Open

JP37: Open

JP38: Open

Additional warning - JP20:

Closing JP20 on the Phytec MPC565 development board connects the MPC565 MDA27 to the ZZ - "Snooze Enable of the burst-RAM". If you wish to use MDA27 on the MPC565 development board then JP20 must be left open.

Please either do not use module 27 with the MIOS Waveform Measurement block or open JP20 on your development board.

Jumper 33 warning: When using the onboard BDM and the download control panel of the coder product, you might see the download timeout as the target is halted, if Jumper 33 is not correctly set. Perform these debugging steps:

- If you download code for debugging: Connect parallel cable to target and make sure Jumper 33 is closed (target stops always while debugging after reset).
- If you download code for stand-alone execution: Disconnect the parallel cable from the target and make sure Jumper 33 is open (target runs in normal mode after reset).
- If you cannot download code, but can debug:

  1. Check if you are using the onboard BDM.
  2. Check the setting of Jumper 33 and state of parallel cable, as stated above.
  3. If this does not resolve the issue, check the other jumper settings.

Axiom MPC555 Jumper Settings. These jumper settings work with an external BDM device.

Make sure you use the default Axiom documented jumper settings and the following additional changes:

Config Switch	Mode Switch 1	Mode Switch 2	Other
1 : Off 2 : On 3 : Off 4 : On 5 : On 6 : On	1 : Off 2 : Off 3 : Off 4 : Off 5 : Off 6 : Off 7 : Off 8 : Off	1 : On 2 : Off 3 : Off 4 : Off 5 : Off 6 : Off 7 : Off 8 : Off	M-SEL Jumper - Open FLSH-SEL Jumper - Open RAM-SEL Jumper - 2 Closed MEM-OPT Jumper - 5, 7 Closed

Axiom MPC564EVB Jumper Settings. These settings work with an external BDM device.

Make sure you use the default Axiom documented jumper settings and the following additional changes:

MAP_SW	CONFIG_SW
1: off 2: on 3: on 4: off 5: on 6: on 7: on 8: on	1: on 2: off 3: on 4: off 5: off 6: off 7: on 8: on

Axiom MPC566EVB Jumper Settings. Make sure you use the default Axiom documented jumper settings and the following additional changes:

MAP_SW	CONFIG_SW
5: off 8: on	7: on 8: on

CS0 > Ext_Flash

CS1 > Ext_SRAM

IP bit off (execute from 0x0000_0100 on reset)

Internal chip Flash enabled

Check the oscillator frequency Target Preference is configured correctly to 4MHz for the 566 board (check the board manual).

CAN Hardware and Drivers

- "Configuring CAN Channels" on page 49-140

- "Creating and Assigning Application Channels" on page 49-140

Configuring CAN Channels

Similarly to the Vector CAN blocks, the Download Control Panel is based on the Vector CAN Driver Programming Library. The Download Control Panel uses the Application Channel mechanism used by the Vector CAN Blocks.

You can use the **CAN Driver Configuration Tool** from Vector to select a CAN channel (installed CAN hardware or a virtual CAN channel) and set the speed of the connection. You can access this tool by clicking **Configure** on the Communication Options tab of the Download Control Panel. Also this tool is opened automatically when you open the Vector CAN Configuration block if you have installed Vector drivers.

Creating and Assigning Application Channels

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window.

2. Select `Download RAM / FLASH Based Application (via CAN / Serial)` from the drop-down list, and click **OK**. This opens the Download Control Panel.

3. On the **Communications Options** tab, select `CAN` from the **Connection type** drop-down menu.

4. Select from the drop-down menu one of the MATLAB application channels (1-10).

   Use the Vector CAN Driver Configuration Tool to create and assign the selected MATLAB application channel to the required CAN hardware device or virtual channel as follows.

5. Click **Configure** on the Download Control Panel to open the Vector CAN Driver Configuration Tool.

6. Click **App. Settings** in the Vector CAN Driver Configuration Tool.

7. Click **Add** in the **Application Settings** dialog that appears.

8. Enter `MATLAB` in the edit box for the new application name and click **OK**.

9. Click **Done** to leave the **Application Settings** dialog.

10. Click to select the CAN hardware device or virtual channel you want to use (for example, Channel 1 of a CAN-AC2-PCI card).

11. Click **Assign to application** (or right-click on the required channel).

12. Select `MATLAB 1` or `MATLAB 2` from the list.

Make sure you select the same MATLAB application channel in the Vector CAN Configuration block. If your model requires more than one application channel take care to assign a different channel to each Vector CAN Configuration block.

See the Vector Help for the CAN Driver Configuration Tool to find out more about how to select the CAN channel, bit rate, synchronization jump width, sample point and number of samples per bit.

Refer also to "Vector CAN Blocks Hardware and Drivers" for more information on configuring your hardware and software drivers.

Configuration for Nondefault Hardware

- "Hardware Clock Configuration" on page 49-142
- "Other Configuration Changes for Nondefault Hardware" on page 49-143

The coder product has been developed and fully tested using the development boards described in "Setting Up Your Target Hardware" on page 49-133. We recommend the use of these boards for getting started. If you are using different MPC5xx hardware, it may be necessary to perform some additional manual configuration.

The following sections provide information about where to make changes for hardware clock configuration and other hardware-specific configurations.

Hardware Clock Configuration

The coder product uses the Periodic Interrupt Timer (PIT) to support a range of sample times. Note that the PIT is driven by the crystal frequency. This results in the following possible sample time ranges:

For a crystal frequency of 20Mhz:

- Fastest sample time = 1.28e-5 seconds.
- Slowest sample time = 0.8388 s.

For a crystal frequency of 4 MHz:

- Fastest sample time = 6.4e-5 s.
- Slowest sample time = 4.1942 s.

Note that if you select a sample time slower than the slowest possible for your clock frequency, Simulink issues a warning message.

Also note that the fastest sample time may not be achievable because timer overruns may occur, depending on your model.

The coder product uses the main system oscillator (OSCM) to provide the system clock. The OSCM uses either a 4-MHz or 20-MHz crystal to generate the PLL reference clock. The next section describes how to configure the real-time target for use on hardware with 4MHz crystal frequency (the default is 20 MHz).

Note External clock inputs are not supported.

Configuring for a Crystal Frequency Other Than 20 MHz. The MPC555 can operate with a crystal frequency of either 4 MHz or 20 MHz. By default, the coder product is configured for a crystal frequency of 20 MHz.

You can use the Target Preferences dialog box to change to a 4MHz oscillator frequency.

1. Open the Utilities for Use with MPC555 dialog box by entering `mpc555utils` in the Command Window.

2. Select `Target Preferences` from the drop-down list, and click **OK**. This opens the Target Preferences dialog box.

3. Use the drop-down menu for `OscillatorFrequency` to change from `20` (the default) to `4`.

4. Now install the appropriate bootcode for your hardware. Open the Utilities for Use with MPC555 dialog box. Select `Install MPC5xx Bootcode` from the drop-down list, and click **OK**.

The correct bootcode is installed for the oscillator frequency and processor variant that you have selected in the Target Preferences. See the Target Preferences section "Target Board" on page 49-14.

Note that you must also change the oscillator frequency and processor variant in your models. Use the Resource Configuration block. The oscillator frequency and processor set here must match the Target Preferences, or you will see warnings.

The default value for `Oscillator_Frequency` is `20`. If you are using 4MHz hardware, you must change the value for `Oscillator_Frequency` to `4` in every model.

See also "**System Clock and Related Parameters** " for information on changing the system clock speed, and the block MPC5xx Switch Target Configuration to easily switch between a selection of preset target configurations with different processors and system frequencies.

Other Configuration Changes for Nondefault Hardware

Depending on your target hardware, it may be necessary to make changes to configure settings such as the size and type of external memory.

If you are downloading using the Freescale CodeWarrior development environment, the relevant hardware configuration settings are contained in *`matlabroot`*`\toolbox\rtw\targets\mpc555dk\mpc555dk\:`

```
@codewarrior_tgtaction\mpc5xx_osc20.cfg
```

```
@codewarrior_tgtaction\mpc5xx_osc4.cfg
```

If you are downloading using the Wind River Compiler and Wind River Systems SingleStep development environment, the configuration settings are contained in *matlabroot*\toolbox\rtw\targets\mpc555dk\mpc555dk\:

```
@diab_tgtaction\mpc5xx_osc20.cfg
@diab_tgtaction\mpc5xx_osc4.cfg
@diab_tgtaction\mpc555.wsp
```

Note that there is now only one Wind River Systems SingleStep workspace file for RAM and flash memory.

The necessary changes to these files depend on the hardware that you are using. Depending on your hardware, you may also need to configure switches and jumper settings. Consult the documentation for your development board.

If you are generating stand-alone real-time applications, you may also need to make changes to settings that are contained in the startup code. These are contained in

```
matlabroot\toolbox\rtw\targets\mpc555dk\drivers\src\applications
\bootcode\bootcode_init.s.t
```

Note that after making any changes to `bootcode_init.s.t`, you must recompile the boot code as described in "Rebuilding the Boot Code and Device Driver Libraries" on page 49-51.

Integrating External Blocksets

- "Introduction" on page 49-144
- "Example External Blockset Folder Structure and rtwmakecfg.m" on page 49-145

Introduction

You can configure a `rtwmakecfg.m` file to seamlessly integrate custom third-party Simulink blocks with the coder product. You must provide the `rtwmakecfg.m` file along with the third party S-function block DLLs and

associated files. `rtwmakecfg.m` files are widely used throughout the coder product and they allow you to:

- Specify include paths to add to the list of includes used in the generated makefiles.
- Specify precompiled libraries to add to the list of libraries used in the generated makefiles.
- Specify TLC include paths to be searched for block TLC files during code generation.

For a general explanation of how to use `rtwmakecfg.m` files, please see the section "Customizing and Creating Template Makefiles" in the Simulink Coder documentation."

For a detailed explanation of using the `rtwmakecfg.m` file please consult the section on "Using the rtwmakecfg.m API" in the Simulink Coder documentation.

The next section contains a detailed explanatory example for the MPC5xx build process.

These steps are required:

- Add the location of the `rtwmakecfg.m` file to the MATLAB path.
- Make sure this file is located in the same folder as the S-function DLLs.

Example External Blockset Folder Structure and rtwmakecfg.m

To understand how the `rtwmakecfg.m` file works, imagine a set of S-functions, comprising a Simulink library, provided by an external supplier, and how they can be integrated into the MPC5xx build process.

Example folder structure for an external (plugin) blockset:

```
C:\externalblocks
C:\externalblocks\tlc_c
C:\externalblocks\include
C:\externalblocks\lib
```

Note: Only the root folder `C:\externalblocks` needs to be on the MATLAB path.

`C:\externalblocks` will contain files such as:

- `Rtwmakecfg.m` — `rtwmakecfg.m` defining MPC5xx Plugins
- `Blocklibrary.mdl` — Simulink block library containing `Sfun_a` and `Sun_b`
- `Sfun_a.mexw32` — S-function member of `Blocklibrary.mdl`
- `Sfun_b.mexw32` — S-function member of `Blocklibrary.mdl`

`C:\externalblocks\tlc_c` will contain files such as:

- `Sfun_a.c` — S-function source for simulation.
- `Sfun_b.c` — S-function source for simulation.
- `Sfun_a.tlc` — S-function TLC for code generation
- `Sfun_b.tlc` — S-function TLC for code generation

Note: `tlc_c` folders in the same folder as the S-function DLLs are automatically added to the TLC include path.

`C:\externalblocks\include` will contain files such as:

- `Blocksetheader.h` — Header file used in the generated code

`C:\externalblocks\lib` will contain files such as:

- Blocksetlibrary_5xx_CODEWARRIOR.a and `Blocksetlibrary_5xx_DIAB.a` — Different versions of the library are required depending on which toolchain is being used. The variable `mpc5xx_tool_chain` (see example `rtwmakecfg.m` below) enables different versions of the library to be selected during the build process, based on the target toolchain.

An example `rtwmakecfg.m` that will add the `Blocksetheader.h` parent folder to the list of include paths and Blocksetlibrary_*`ToolChain`*.a to the list of libraries follows:

```
% RTWMAKECFG adds include and source folders to rtw make files.
% makeInfo=RTWMAKECFG returns a structured array containing build info.
% Please refer to the rtwmakecfg API section in the Simulink Coder
% Documentation for details on the format of this structure.

% Get hold of the fullpath to this file, without the filename itself
rootpath = fileparts(mfilename('fullpath'));

% Get hold of the toolchain token to uniquely indentify libraries
prefs = RTW.TargetPrefs.load('mpc555.prefs');
mpc5xx_tool_chain = upper(prefs.ToolChain);

% External blocks need the following include path added
% Add the header file
makeInfo.includePath = { fullfile(rootpath, 'include') };

% External blocks reference the following precompiled library
% Add the precompiled libraries
makeInfo.linkLibsObjs = { fullfile(rootpath, 'lib',...
['Blocksetlibrary_' mpc5xx_tool_chain '.a']) };
```

Working with Green Hills MULTI IDE

- “Getting Started” on page 50-2
- “Automation Interface” on page 50-10
- “Project Generator” on page 50-33
- “Breakpoints and PIL” on page 50-44

Getting Started

In this section...

"Overview" on page 50-2

"Software Structure and Components" on page 50-3

Overview

Embedded Coder software provides an interface between MATLAB and the Green Hills MULTI IDE software. The software enables you to

- Access the processor
- Manipulate data on the processor
- Manage projects within the IDE

while using the MATLAB numerical analysis and simulation functions.

Embedded Coder software connects MATLAB and Simulink with Green Hills MULTI integrated development and debugging environment from Green Hills®. The software enables you to use MATLAB and Simulink to debug and verify embedded code running on many microprocessors that Green Hills MULTI software supports, such as the ARM, Freescale MPC5500 and MPC7400, Blackfin, and NEC® V850 families.

Using the software, you can perform the following tasks and others related to Model-Based Design:

- Function calls — Write scripts in MATLAB to execute any function in the Green Hills MULTI IDE
- Automation — Write automated tests in MATLAB to execute on your processor, including control and verification operations
- Host-Processor Communication — Communicate with the processor directly from MATLAB, without going to the IDE
- Verification and Validation

  - Load and execute projects into the Green Hills MULTI IDE software from the MATLAB command line
  - Build and compile code, and then use vectors of test data and parameters to test the code
  - Build and compile your code, and then download the code to the processor and execute it
- Design models — Design models and algorithms in MATLAB and Simulink and run them on the processor
- Generate code — Generate executable code for your processor directly from the models designed in Simulink, and execute it

Embedded Coder software includes a project generator component. With the project generator component, you can generate a complete project file for Green Hills MULTI software from Simulink models, using C code generated with Embedded Coder software. Thus, you can use both Simulink Coder and Embedded Coder software to generate generic ANSI C code projects for Green Hills MULTI from Simulink models. You can then build and run the code on supported processors.

The following list suggests some of the uses for Embedded Coder software:

- Create test benches in MATLAB and Simulink for testing your manually written or automatically generated code running on a variety of DSPs
- Generate code and project files for Green Hills MULTI software from Simulink models using both Simulink Coder and Embedded Coder software for rapid prototyping or deployment of a system or application
- Build, debug, and verify embedded code on supported processors with MATLAB, Simulink, and Green Hills MULTI software
- Perform processor-in-the-loop (PIL) testing of embedded code

Software Structure and Components

- “Components” on page 50-4
- “Automation Interface” on page 50-4
- “Project Generator” on page 50-5

- “Verification” on page 50-5
- “Configuring Your Software” on page 50-5
- “Configuring Green Hills® MULTI to use Full Folder Paths” on page 50-8

Components

Embedded Coder software comprises these components

- Automation Interface — Enables communication between MATLAB and Green Hills MULTI software.
- Project Generation — Uses Simulink to let you build models, simulate them, and generate code from the models directly to the processor.
- Verification — Validate and verify your projects. You can simulate algorithms and processes in Simulink models and concurrently on your processor. Comparing the concurrent simulation results helps verify the fidelity of your model or algorithm code.

Automation Interface

The Automation Interface component enables you to use MATLAB functions and methods to communicate with the Green Hills MULTI IDE software. With the MATLAB functions, you can perform the following program development tasks:

- Automate project management.
- Debug projects by manipulating the data in the processor memory (internal and external) and registers.
- Exercise functions from your project on the processor.
- Communicate between the host and processor applications.

The Automation Interface component provides the following functionality in the Debug component—methods and functions for project automation, debugging, and data manipulation.

Project Generator

The Project Generator component is a collection of methods that use the Green Hills MULTI API to create projects in Green Hills MULTI and generate code. With the interface, you can do the following:

- Automatic project-based build process — Automatically create and build projects for code generated by Simulink Coder or Embedded Coder.
- Custom code generation — Use System Target Files (STF) to generate both processor-specific and optimized code.
- Automatic downloading and debugging — Debug generated code in the Green Hills MULTI debugger, using either the instruction set simulator or real hardware.
- Create and build projects for Green Hills MULTI from Simulink models — Project Generator uses Simulink Coder or Embedded Coder to build projects that work with supported processors.
- Generate custom code using the Configuration Parameters in your model with the system target files `multilink_ert.tlc` and `multilink_grt.tlc`.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded Coder software provide the following verification tools.

- **Processor in the loop (PIL) simulation** — Use simulation techniques to verify generated code running in an instruction set simulator or real hardware environment.
- **Execution profiling** — Gather execution profiling measurements with Green Hills MULTI instruction set simulator to establish the timing requirements of your algorithm.

Configuring Your Software

Embedded Coder software requires some information about your MULTI installation before you can use the software to develop projects in MULTI from MATLAB. To configure the interface between MATLAB and MULTI, provide the information in the following table. Embedded Coder software

provides a GUI-based configuration utility to help you configure the software and interface.

GUI Parameter	Configuration Information	Description
Directory	MULTI installation folder	Identifies the path to your Green Hills software.
Configuration	Primary processor	Identifies the processor on which you are developing.
Debug server	Debug server type	Specifies the type of debug server to use.
Host name	Host name	Specifies the name of the machine that runs your IDE Link service.
Port number	Port number	Specifies the port for communicating with the host and IDE Link service. The service listens on this port.

Configuring Embedded Coder Software. You must configure your installation before you start working with the software and MULTI.

To generate code for Blackfin processors, the software supports only the Green Hills version of the Blackfin compiler.

Note The software does not support using Analog Devices Blackfin compiler. When you select your configuration during the configuration process, do not select `bfadi_standalone.tgt` from the **Configuration** list. `bfadi_standalone.tgt` uses the ADI compiler.

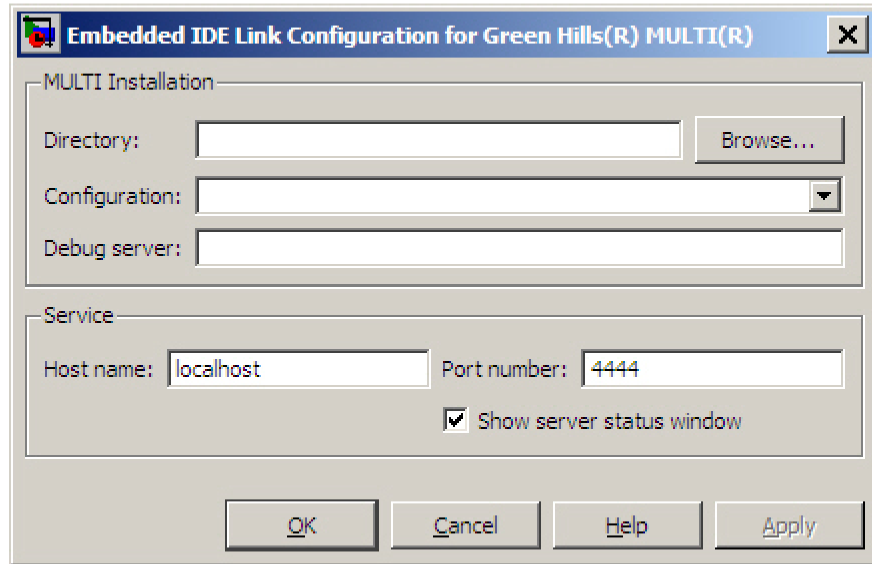
Follow these steps to open the Embedded Coder configuration utility:

Note You must perform this configuration process before using Embedded Coder software.

---

1 Enter `ghsmulticonfig` at the MATLAB prompt.

The Embedded Coder Configuration dialog box opens, as shown in the following figure.



2 In the **Directory** field, enter the path to the executable file `multi.exe` for your Green Hills MULTI installation. Click **Browse** to search for the file if necessary.

3 From the **Configuration** list, select your primary processor. Embedded Coder software supports a variety of processors. Choose one that matches your development platform. In many cases, the *`processor`*`_standalone.tgt` variants, such as `ppc_standalone.tgt`, work well. Refer to your Green Hills MULTI documentation for more information about the configuration options for processors.

4 Enter the debug server string in **Debug server**. The string you enter sets specific values for processors, such as the board support library and whether the processor is big or little endian.

The standard input string is `debugconnection`. To use a processor simulator, such as the MPC5554 simulator, enter the string

```
simppc -cpu=ppc5554 -fast -dec-rom_use_entry
```

Your MULTI documentation provides more information about the debug server options and how to use them. You can find more debug server string for simulators in the reference material for `ghsmulticonfig`.

Note If you use a custom board, add the `-bsp` option to the **Debug server** string to specify your processor. For example, add `-bsp=mpc5554` if you use the MPC5554 EVB.

---

5. In **Host name**, enter the name of the machine that is going to run the IDE Link service. When you construct a `ghsmulti` object, the `ghsmulti` function starts the IDE Link service. To launch the service, the function needs to know where the service will run. The **Host name** string identifies that location. The default value is `localhost`, meaning the service runs on the local machine. No other input is valid.

6. Enter the port number for the service in **Port number**.

   Port number 4444 is the default port value. To change the port used, enter a different value in this field. Verify that the port you enter is available. If the port number you enter is not available, the IDE Link service does not start. Thus, you get an error message in MATLAB when you try to construct a `ghsmulti` object.

7. Select or clear **Show server status window** to specify whether the IDE Link service status appears in the task bar. The default value is to show the service status. Clearing **Show server status window** hides the status in the task bar. Select this option as a best practice. Keeping this option selected enables the software to shut down the communication services for Green Hills MULTI completely.

8. Click **OK** to complete the configuration process and close the dialog box.

Configuring Green Hills MULTI to use Full Folder Paths

When you install MULTI to use with the software, MULTI sets the **Show Paths** option to use relative file paths. To ensure that projects and programs

build correctly, configure MULTI to use full folder paths. Follow these steps to change the configuration in MULTI.

1 Start MULTI from your desktop.

2 Switch to the Project Manager tool.

3 Select **View > Show Paths > Full Paths**.

Automation Interface

In this section...

"Getting Started with Automation Interface" on page 50-10

"Constructing Objects" on page 50-26

"Properties and Property Values" on page 50-27

"ghsmulti Object Properties" on page 50-30

Getting Started with Automation Interface

- "Introducing the Automation Interface Tutorial" on page 50-10
- "Starting and Stopping Green Hills MULTI From the MATLAB Desktop" on page 50-12
- "Running the Interactive Tutorial" on page 50-16
- "Querying Objects for Green Hills MULTI Software" on page 50-16
- "Loading Files into Green Hills MULTI Software" on page 50-17
- "Running the Project" on page 50-19
- "Working With Data in Memory" on page 50-20
- "More Memory Data Manipulation" on page 50-22
- "Closing the Connections to Green Hills MULTI Software" on page 50-25
- "Tasks Performed During the Tutorial" on page 50-25

Introducing the Automation Interface Tutorial

Embedded Coder software provides a connection between MATLAB software and a processor in Green Hills MULTI development environment. You use MATLAB objects as a mechanism to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you while you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Note Before using the functions available with the objects, you must designate a server and processor in Green Hills MULTI software. The object you create is specific to the server and processor you specify.

To help you start using objects in the software, Embedded Coder software includes a tutorial—`multilinkautointtutorial.m`. As you work through this tutorial, you perform the following tasks that step you through creating and using objects to interact with the Green Hills MULTI IDE:

1. Select your primary server and port.
2. Create and query objects to Green Hills MULTI IDE.
3. Use MATLAB to load files into Green Hills MULTI IDE.
4. Work with your Green Hills MULTI IDE project from MATLAB.
5. Close the connections you opened to Green Hills MULTI IDE.

The tutorial covers some methods and functions for the software. The following tables show functions and methods for the software. The functions do not require an object. The methods require an existing `ghsmulti` object to use as an input argument for the method.

Functions for Working with Green Hills MULTI. The following table shows functions that do not require an object.

Function	Description
`ghsmulti`	Construct an object that refers to a Green Hills MULTI IDE instance. When you construct the object you specify the IDE instance by host and port.
`ghsmulticonfig`	Set Embedded Coder software preferences.

Methods for Working with ghsmulti Objects in Green Hills MULTI. The following table presents some of the methods that require a `ghsmulti` object.

Methods	Description
add	Add file to project
address	Return address and page for entry in symbol table in Green Hills MULTI IDE
build	Build project in Green Hills MULTI
cd	Change working folder
connect	Connect IDE to processor
display	Display properties of object that references Green Hills MULTI IDE
halt	Terminate execution of process running on processor
isrunning	Test whether processor is executing process
load	Load built project to processor
open	Open file in project
read	Retrieve data from memory on processor
regread	Read values from processor registers
regwrite	Write data values to registers on processor
reset	Restore program counter (PC) to entry point for current program.
restart	Restore processor to program entry point
run	Execute program loaded on processor
write	Write data to memory on processor

Starting and Stopping Green Hills MULTI From the MATLAB Desktop

Embedded Coder software provides you the ability to control MULTI software from the MATLAB command window. When you create a `ghsmulti` object, MATLAB starts the services shown in the following table to enable MATLAB to communicate with the Green Hills MULTI IDE:

Service Type for Each Port	Process Name	Description
Python Service	`mpythonrun.exe`	Python is a programming language the software uses to establish a connection between MATLAB and MULTI.
Python Service	`svc_python.exe`	Connection to IDE.
Python Service	`svc_router.exe`	Connection to IDE.
Python Service	`svc_statemgr.exe`	Connection to IDE
Python Service	`svc_window.exe`	Connection to IDE.
IDE Link service	Not applicable	Enables MATLAB to send commands to the Green Hills MULTI development environment. This is a child process of the python services.

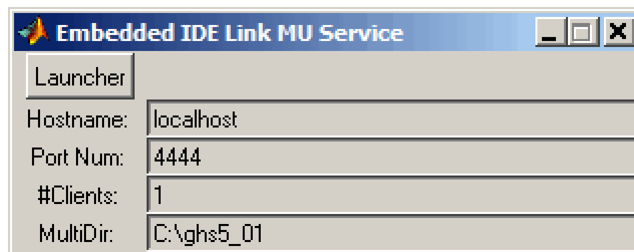
Each time you create a `ghsmulti` object, the software starts another set of the python services shown in the table.

Starting Green Hills MULTI From MATLAB. When you use the `ghsmulti` function, the software starts two classes of services—python services and the IDE Link service for each new port. The entries in the following table describe how the software controls the IDE when you create a `ghsmulti` object:

Create ghsmulti Object with ghsmulti Function	Status of IDE	Result
`id=ghsmulti`	Not running	The software starts the IDE Link service and the IDE connects to the default host name and port number—`localhost` and `4444` as set in the configuration options.

Create ghsmulti Object with ghsmulti Function	Status of IDE	Result
`id=ghsmulti('hostname','localhost','portnum',4444)`	Not running	The software starts the IDE Link service and the IDE and connects to the specified host name and port number—`localhost` and `4444`.
`id2=ghsmulti`	Running	The software connects to the existing IDE Link service connected to the default host name and port.
`id2=ghsmulti('hostname','localhost','portnum',4446)`	Running	The software starts a new the IDE Link service connected to the specified host name and port number.

When the software starts the IDE Link service, the following service dialog box appears on your desktop:



Information in the window provides details about the service. Clicking **Launcher** opens the MULTI Launcher utility.

Stopping Green Hills MULTI From MATLAB. After you complete your development work with the software, best practice suggests that you close the IDE from MATLAB. Two conditions control how you close the IDE, as shown in the following table:

The IDE Link Service State	To Close the IDE
One or more services appear in the task bar and the IDE Link service dialog boxes are visible.	Perform these steps: 1 Enter `clear all` in MATLAB to remove the `ghsmulti` objects from your workspace. 2 Verify that the MULTI clients are no longer connected by checking that **#Clients** in each service dialog box is `0`. 3 Close the service dialog boxes.
Services appear in the task bar but the service dialog boxes are not visible.	Perform these steps: 1 Enter `clear all` inMATLAB to remove the `ghsmulti` objects from your workspace. 2 Open the MicrosoftWindows Task Manager. 3 Click **Processes**. 4 Select `svc_router.exe` from the list. Closing this service stops `mpythonrun.exe`, `svc_window.exe`, and `svc_statemgr.exe`. 5 Click **End Now**. 6 Select `svc_python.exe` from the list. 7 Click **End Now**.

Note Clicking the task bar icon for the service and selecting close does not close the IDE correctly.

Running the Interactive Tutorial

You have the option of running this tutorial from the MATLAB command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB, click `run multilinkautointtutorial`. This command launches the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next section. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial MATLAB file used here by clicking `multilinkautointtutorial.m`.

Querying Objects for Green Hills MULTI Software

In this tutorial section you create the connection between MATLAB and Green Hills MULTI IDE. This connection, or `ghsmulti` object, is a MATLAB object that you save as variable `id`. You use function `ghsmulti` to create `ghsmulti` objects. `ghsmulti` supports input arguments that let you specify values for `ghsmulti` object properties, such as the global timeout. Refer to the `ghsmulti` reference information for more about the input arguments.

Use the generated object `id` to direct actions to your project and processor. In the following tasks, `id` appears in all method syntax that interact with the IDE primary target and the processor: The object `id` identifies and refers to a specific instance of the IDE.

You must include the object in any method syntax you use to access and manipulate a project or files in a session in Green Hills MULTI software:

1. Create an object that refers to your selected service and port. Enter the following command at the prompt.

   ```
   id = ghsmulti('hostname','localhost','portnum',4444)
   ```

2. Next, enter `display(id)` at the prompt to see the status information.

```
MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\ppc\
```

Embedded Coder software provides three methods to read the status of a processor:

- `info` — Return a structure of testable session conditions.
- `display` — Print information about the session and processor.
- `isrunning` — Return the state (running or halted) of the processor.

3 Verify that the processor is running by entering

```
runstatus = isrunning(id)
```

The MATLAB prompt responds with message that indicates the processor is stopped:

```
runstatus =

     0
```

Loading Files into Green Hills MULTI Software

You have established the connection to a processor and board. Using three methods you learned about the hardware, and whether it was running. Next, give the processor something to do.

In this part of the tutorial, you load the executable code for the CPU in the IDE. Embedded Coder software includes a tutorial project file for Green Hills MULTI. Through the next commands in the tutorial, you locate the tutorial project file and load it into Green Hills MULTI. The `open` method directs Green Hills MULTI to load a project file or workspace file.

Note To continue the tutorial, you must identify or create a folder to which you have write access. Embedded Coder software cannot create a folder for you. Create one in the Microsoft Windows folder structure before you proceed with the this tutorial.

Green Hills MULTI has its own workspace and workspace files that are quite different from MATLAB workspace files and the MATLAB workspace. Remember to monitor both workspaces. To change the working folder to your writable folder:

1. Use `cd` to switch to the writable folder

   ```
   prj_dir=cd('C:\ide_link_mu_demo')
   ```

   where the name and path to the writable folder is a string, such as `C:\ide_link_mu_demo` as used in the example. Replace `C:\ide_link_mu_demo` with the full path to your writable folder.

2. Change your working folder to the new folder by entering the following command:

   ```
   cd(id,prj_dir)
   ```

3. Use the following command to create a new Green Hills MULTI project named `debug_demo.gpj` in the new folder:

   ```
   new(id,'debug_demo.gpj')
   ```

   Switch to the IDE to verify that your new project exists. Next, add source files to your project.

4. Add the provided source file—`multilinkautointtutorial.c` to the project `debug_demo.gpj` using the following command:

   ```
   add(id,'multilinkautointtutorial.c')
   ```

5. Save your project.

   ```
   save(id,'my_debug_demo.gpj','project')
   ```

Your IDE project is saved with the name my_debug_demo.gpj in your writable folder. The input string 'project' specifies that you are saving a project file.

6 Next, set the build options for your project. Use the following command to set the compiler build options to use and specify a processor (optional).

```
setbuildopt(id,'Compiler','-G -cpu=V850')
```

The input argument -cpu=V850 is optional to specify the processor. Change to processor designation to match your processor if necessary.

Running the Project

After you create dot_project_c.gpj in the IDE, you can use Embedded Coder software functions to create executable code from the project and load the code to the processor.

To build the executable and download and run it on your processor:

1 Use the following build command to build an executable module from the project debug_demo.gpj.

```
build(id,'all',20) % Set optional time-out period to 20 seconds.
```

2 To load the new executable to the processor, use load with the project file name and the object name. The name of the executable is debug_demo.

```
load(id,'debug_demo',30); % Set time-out value to 30 seconds.
```

Embedded Coder software provides methods to control processor execution—run, halt, and reset. To demonstrate these methods, use run to start the program you just loaded on to the processor, and then use halt to stop the processor.

1 Enter the following methods at the command prompt and review the response in the MATLAB command window.

```
run(id)      % Start the program running on the processor.
halt(id)     % Halt the processor.
reset(id)    % Reset the program counter to start of program.
```

Use `isrunning` after the `run` method to verify that the processor is running. After you stop the processor, `isrunning` can verify that the processor has stopped.

Working With Data in Memory

Embedded Coder software provides methods that enable you to read and write data to memory on the processor. Reading and writing data depends on the symbol table for your project. The symbol table is available only after you load the executable into the debugger. This sections introduces `address` and `dec2hex`. Use them to read the addresses of two global variables—`ddat` and `idat`.

1 After you load `debug_demo` into the debugger, enter the following commands to read the addresses of `ddat` and `idat`:

```
ddatA=address(id,'ddat')
ddatA =
     3145744        0

ddatI=address(id,'idat')

ddatI =

     3145728           0
```

2 Review the results in hexadecimal representation.

```
dec2hex(ddatA)

ans =

300010
000000

dec2hex(ddatI)

ans =

300000
000000
```

After you load the target code to the processor, you can examine and modify data values in memory, as the previous `read` function examples demonstrated.

For non-changing data values in memory (static values), the values are available immediately after you load the program file.

A more interesting case is looking at variable values that change during program execution. Manipulating changing data values at intermediate points during execution can provide helpful analysis and verification information.

To enable you to read and write data while your program is running, the software provides methods to insert and delete breakpoints in the source programs. Inserting breakpoints lets you pause program execution to read or change variable data values. You cannot change values while your program is running.

The method `insert` creates a new breakpoint at either a source file locations, such as a line number, or at a physical memory address. `insert` takes either the line number or the address as an input argument.

To read the values in the next section of this tutorial, use the following methods to insert breakpoints at lines 24 and 29 in the source file `multilinkautointtutorial.c`

1 Change folders to your original working folder.

```
cd(id,proj_dir);
```

2 (Optional for convenience) Create variables for the line numbers in the source file.

```
brkpt24 = 24;
brtpt29 = 29;
```

3 Use the following commands to insert breakpoints on line 24 and line 29 of the source file:

```
insert(id,'multilinkautointtutorial',brkpt24); % Insert breakpoint on line 24.
insert(id,'multilinkautointtutorial',brkpt29); % Insert breakpoint on line 29.
```

4 Open and activate the file in the IDE from the MATLAB command window by issuing the following commands:

```
open(id,'multilinkautointtutorial');
activate(id,'multilinkautointtutorial');
```

Activating `multilinkautointtutorial.c` transfers focus in the IDE to the activated file. Switch to the IDE to verify that the file is in your project and open.

When you look in the IDE debugger window, the breakpoints you added to `multilinkautointtutorial.c` are marked by a STOP sign icon on lines 24 and 29.

A similar method, `remove`, deletes breakpoints.

To help you inspect the source file in the IDE and verify the breakpoints, the `open` and `activate` methods display the file `multilinkautointtutorial.c` in the IDE and force the source file to the front.

One final method actually connects the IDE to your hardware or simulator. `connect` takes a `ghsmulti` object as an input argument to connect the specific IDE primary target referenced by `id` to the associated processor.

More Memory Data Manipulation

The source file `multilinkaautointtutorial.c` defines two 1-by-4 global data arrays—`ddat` and `idat`. You can locate the declaration in the file. Embedded Coder software provides the read and write methods so you can access the arrays from MATLAB. Find the declaration and note the initialization values.

This tutorial section demonstrates reading and writing data in memory, and controlling the processor.

1 Get the address of the symbols `ddat` and `idat`. Enter the following commands at the prompt.

```
ddat_addr=address(id,'ddat'); % Get address from symbol table.
idat_addr=address(id,'idat');
```

2 Create two MATLAB variables to specify the data types for `ddat` and `idat`.

```
ddat_type-'double';
idat_type='int32';
```

3 Declare some values in two MATLAB variables.

```
ddat_value=double([pi 12.3 exp(-1) sin(pi/4)]);
idat_value=int32(1:4);
```

4 Stop the processor.

```
halt(id)
```

5 Reload the project. If you did not save the source file in the project, reloading the project removes the breakpoints you added and move the program counter (PC) to the start of the program.

```
% Reload program file (.gpj). Reset PC to program start.
reload(id,100);
```

6 Use the following commands to restore the breakpoints on line 24 and 29.

```
insert(id,'multilinkautointtutorial.c',brkpt24);
insert(id,'multilinkautointtutorial.c',brkpt29);
```

7 Use the following method to connect the IDE to the processor:

```
connect(id);
```

8 With the breakpoints in the code, run the program until it stops at the first breakpoint on line 24.

```
run(id,'runtohalt',30); % Set time-out to 30 seconds.
```

9 Check the current values stored in `ddat` and `idat`. Later in this tutorial you change these values from MATLAB.

```
% Do ddat values match initialization values in the source?
ddatV=read(id,address(id,'ddat',ddat_type,4)
idatV=read(id,address(id,'idat',idat_type,4)
```

MMATLAB displays the values of `ddatV` and `idatV`.

```
ddatV=
```

```
   16.300   -2.1300   5.1000   11.8000

idatV=

        1 508   646   7000
```

10 Change the values in ddat and idat by writing new values to the memory addresses.

```
% Write pi, 12.3, exp(-1), and .7070 to memory.
write(id,address(id,'ddata'),ddat_value)
% Write vector [1:4] to memory.
write(id,address(id,'idat'),idat_value)
```

11 Resume the program execution from the breakpoint and run until the program stops.

```
run(id,'runtohalt','30); % Stop at next breakpoint (line 29).
```

12 Read the values in memory for ddat and idat to verify the changes.

```
% Read the data as double data type.
ddatV = read(id,address(id(id,'ddat'),ddat_type,4)

ddatV=

3.1416   12.3000   0.3679   0.7071

% Read the data as int32 data type.
idatV = read(id,address(id,'idat'),idat_type,4)

idatV=

       1    2    3    4
```

The data stored in ddat and idat are what you wrote to memory.

13 After you review the data, restart the processor to run to return the PC to the program start.

```
restart(id);
```

Closing the Connections to Green Hills MULTI Software

Objects that you create in Embedded Coder software have connections to Green Hills MULTI IDE. Until you delete these objects, the Green Hills MULTI process (`Idde.exe` in the Windows Task Manager) remains in memory. Closing MATLAB removes these objects automatically, but there may be times when it helps to delete the handles manually, without quitting MATLAB.

Note When you clear the last `ghsmulti` object, the software does not close the running IDE Link service. When it does close the IDE, it does not save current projects or files in the IDE, and it does not prompt you to save them.

A best practice is to save your projects and files before you clear `ghsmulti` objects from your MATLAB workspace.

Use the following commands to close the project files in Green Hills MULTI IDE and remove the breakpoints you added to the source file.

```
close(id,'debug_demo.gpj','project') % Close the project file.
remove(id,'multilinkautointtutorial.c',brkpt24);

remove(id,'multilinkautointtutorial.c',brkpt29);
```

Finally, to delete your link to Green Hills MULTI use `clear id`.

You have completed the Automation Interface tutorial using Embedded Coder software.

Tasks Performed During the Tutorial

During the tutorial you performed the following tasks:

1. Created and queried objects that refer to a session in Embedded Coder software to get information about the session and processor.

2. Used MATLAB software to load files into the Green Hills MULTI IDE and used methods in MATLAB software to run that file.

3. Closed the links you opened to Green Hills MULTI software.

This set of tasks is used in any development work you do with signal processing applications. Thus, the tutorial gives you a working process for using Embedded Coder software and your signal processing programs to develop programs for a range of processors.

Constructing Objects

When you create a connection to a session in Green Hills MULTI using the `ghsmulti` function, you create a `ghsmulti` object (in object-oriented design terms, you *instantiate* the `ghsmulti` object). The object implementation relies on MATLAB object-oriented programming capabilities like the objects in MATLAB or DSP System Toolbox software.

The discussions in this section apply to the objects in Embedded Coder software. Because `ghsmulti` objects use the MATLAB software techniques, the information about working with the objects, such as how you get or set object properties or use methods, apply to the `ghsmulti` objects in Embedded Coder software.

Like other MATLAB structures, ghsmulti objects have predefined fields referred to as *object properties*.

You specify object property values by the following methods:

- Specifying the property values when you create the object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting link properties, refer to "Setting Property Values with set".

Example — Constructor for ghsmulti Objects

The easiest way to create an object is to use the function `ghsmulti` to create an object with the default properties. Create an object named `id` referring to a session in Green Hills MULTI by entering the following syntax:

```
id = ghsmulti
```

MATLAB responds with a list of the properties of the object `id` you created along with the associated default property values.

```
MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\ppc\
```

The object properties are described in the `ghsmulti` documentation.

Note These properties are set to default values when you construct links.

Properties and Property Values

- "Working with Properties" on page 50-27
- "Setting and Retrieving Property Values" on page 50-28
- "Setting Property Values Directly at Construction" on page 50-28
- "Setting Property Values with set" on page 50-29
- "Retrieving Properties with get" on page 50-29
- "Direct Property Referencing to Set and Get Values" on page 50-30
- "Overloaded Functions for ghsmulti Objects" on page 50-30

Working with Properties

Links (or objects) in this Embedded Coder software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. Also, a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

Setting and Retrieving Property Values

You can set `ghsmulti` object property values by either of the following methods:

- Directly when you create the link — see "Setting Property Values Directly at Construction"
- By using the `set` function with an existing link — see "Setting Property Values with set"

Retrieve `ghsmulti` object property values with the `get` function.

Direct property referencing lets you either set or retrieve property values for `ghsmulti` objects.

Setting Property Values Directly at Construction

To set property values directly when you construct an object, include the following entries in the input argument list for the constructor method `ghsmulti`:

- A string for the property name to set, followed by a comma. Enclose the string in single quotation marks as you do any string in MATLAB.
- The property value to associate with the named property. Sometimes this value is also a string.

You can include as many property names in the argument list for the object construction command as there are properties to set directly.

Example — Setting Link Property Values at Construction. Create a connection to an instance of the IDE in Green Hills MULTI software and set the following object properties:

- Link to the specified MULTI instance and host.
- Specify the communication port on the host.
- Set the global timeout to 5 s. The default is 10 s.

Set these properties when you construct the object by entering

```
id = ghsmulti('hostname','localhost','portnum',4444,'timeout',5);
```

The `localhost`, `portnum`, and `timeout` properties are described in Link Properties, as are the other properties for links.

Setting Property Values with set

After you construct an object, the `set` function lets you modify its property values.

Using the `set` function, you can Set link property values.

Example — Setting Link Property Values Using set. To set the timeout specification for the link `id` from the previous section, enter the following syntax:

```
set(id,'timeout',8);

get(id,'timeout');
ans=

     8
```

The display reflects the changes in the property values.

Retrieving Properties with get

You can use the `get` command to retrieve the value of an object property.

Example — Retrieving Link Property Values Using get. To retrieve the value of the `hostname`property for `id`, and assign it to a variable, enter the following syntax:

```
host=get(id,'hostname')

host =

localhost
```

Direct Property Referencing to Set and Get Values

You can directly set or get property values using MATLAB structure-like referencing. Do this by using a period to access an object property by name, as shown in the following example.

Example — Direct Property Referencing in Links. To reference an object property value directly, perform the following steps:

1. Create a link with default values.

2. Change its time out and number of open channels.

```
id = ghsmulti;
id.time = 6;
```

Overloaded Functions for ghsmulti Objects

Several methods and functions in Embedded Coder software have the same name as functions in other MathWorks products. These functions behave similarly to their original counterparts, but you apply them to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for objects. After you specify your object by assigning values to its properties, you can apply the methods in this toolbox (such as `address` for reading an address in memory) directly to the variable name you assign to your object. You do not have to specify your object parameters again.

For a list of the methods that act on `ghsmulti` objects, refer to the Green Hills MULTI in the function reference pages.

ghsmulti Object Properties

- "Quick Reference to ghsmulti Properties" on page 50-31
- "Details About ghsmulti Object Properties" on page 50-31

Quick Reference to ghsmulti Properties

The following table lists the properties for the links in Embedded Coder software. The second column indicates to which object the property belongs. Knowing which property belongs to each object in an interface tells you how to access the property.

Property Name	User Settable?	Description
`hostname`	At construction only	Reports the name of the host the IDE Link service in Green Hills MULTI that the object references.
`portnum`	At construction only	Stores the number of the port to communicate with MULTI.
`timeout`	Yes/default	Contains the global timeout setting for the link.

Some properties are read only. Thus, you cannot set the property value. Other properties you can change at any time. If the entry in the User Settable column is “At construction only,” you can set the property value only when you create the object. Thereafter, it is read only.

Details About ghsmulti Object Properties

To use the objects for Green Hills MULTI interface, set values for the following:

- `hostname` — Specify the session with which the object interacts.
- `portnum` — Specify the processor in the session. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — Specify the global timeout value. (Optional. Default is 10 s.)

Details of the properties associated with `ghsmulti` objects appear in the following sections, listed in alphabetical order by property name.

hostname. Property `hostname` identifies the host that is running the IDE Link service. Use `hostname` to specify the machine to host your service.

To work with a service, you need the `hostname` and `portnum` values. `Hostname` supports the string `localhost` only.

portnum. Property `portnum` specifies the port for communicating with the IDE Link service. MATLAB uses sockets to communicate with Green Hills MULTI. The `portnum` property value specifies the port, with a default value of 4444. When you create a new `ghsmulti` object, Embedded Coder software assumes the port value is 4444 unless you enter a different value when you configure the software or use the `portnum` input argument with `ghsmulti`.

timeout. Property `timeout` specifies how long Green Hills MULTI waits for any process to finish. You set the global timeout when you create an object for a session in Green Hills MULTI. The default global timeout value 10 s. The following example shows the `timeout` value for object `id2`.

```
display(id2)

MULTI Object:
  Host Name        : localhost
  Port Num         : 4444
  Default timeout  : 10.00 secs
  MULTI Dir        : C:\ghs\multi500\ppc\
```

Project Generator

In this section...

"Introducing Project Generator" on page 50-33

"Project Generator Tutorial" on page 50-34

"Model Reference" on page 50-39

Introducing Project Generator

Project generator provides the following features for developing projects and generating code:

- Automated project building for Green Hills MULTI that lets you create MULTI projects from code generated by Simulink Coder and Embedded Coder. Project generator populates projects in the MULTI development environment.
- Blocks in the library `idelinklib_ghsmulti` for controlling the scheduling and timing in generated code.
- Highly configurable code generation using model configuration parameters and Target Preferences block options.
- Ability to use one of two system target files to generate code specific to your processor.
- Highly configurable project build process.
- Automatic downloading and running of your generated projects on your processor.

To configure your Simulink models to use the Project Generator component, do one or both of the following tasks:

- Add a Target Preferences block to your model.
- To use the asynchronous scheduler capability in Embedded Coder software, add a hardware interrupt block or idle task block.

The following sections describe the blockset and the blocks in it, the scheduler, and the Project Generator component.

Project Generator Tutorial

- “Process for Building and Generating a Project” on page 50-34
- “Create the Model” on page 50-35
- “Adding the Target Preferences Block to Your Model” on page 50-36
- “Specifying Simulink Configuration Parameters for Your Model” on page 50-36
- “Creating Your Project” on page 50-39

Process for Building and Generating a Project

In this tutorial, you build a model and generate a project from the model into Green Hills MULTI.

Note The model shows project generation only. You cannot build and run the model on your processor without additional blocks.

This is an overview of the process for generating a project from a model:

1. Use Simulink blocks, DSP System Toolbox blocks, and blocks from other blocksets to create the model application.
2. Add the Target Preferences block from the `idelinklib_common` library to your model, as described in “Target Preferences” on page 43-4..
3. Set the configuration parameters for your model, including the following parameters:
   - Solver parameters such as simulation start and solver options. Choose the discrete solver when you generate executables. If you are using PIL, select any setting from the **Type** and **Solver** lists.
   - Simulink Coder options such as processor configuration and processor compiler selection
4. Generate your project.
5. Review your project in MULTI.

Create the Model

To build the model for this tutorial, follow these steps:

1. Use Simulink blocks, DSP System Toolbox blocks, and blocks from other blocksets to create the model application.

2. Add the Target Preferences block from the `idelinklib_common` library to your model, as described in “Target Preferences” on page 43-4..

3. Set the configuration parameters for your model, including the following parameters:

   - Solver parameters such as simulation start and solver options. Choose the discrete solver when you generate executables. If you are using PIL, select any setting from the **Type** and **Solver** lists.
   - Simulink Coder options such as processor configuration and processor compiler selection

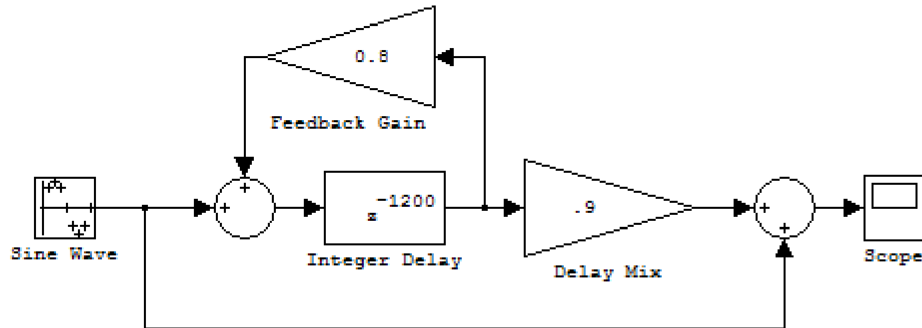
4. Generate your project.

5. Review your project in MULTI.

1. Start Simulink.

2. Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.

3. Use Simulink blocks and DSP System Toolbox blocks to create the following model.



Look for the Integer Delay block in the Discrete library of Simulink and the Gain block in the Commonly Used Blocks library. This model implements an audio signal reverberation scheme. Part of the input audio signal passes directly to the output. A delayed version passes through a feedback loop before reaching the output. The result is an echo, or reverberation, added to the audio output.

4. Save your model with a suitable name before continuing.

Adding the Target Preferences Block to Your Model

To configure your model to work with the processors your IDE supports, add a Target Preferences block to your model, as described in "Target Preferences" on page 43-4.

You have completed the model. Next, configure the model configuration parameters to generate a project in Green Hills MULTI from your model.

Specifying Simulink Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink.

Setting Solver Options. After you have designed and implemented your digital signal processing model in Simulink, complete the following steps to set the configuration parameters for the model:

1 Open the Configuration Parameters dialog box and set the appropriate options on the `Solver` category for your model and for Embedded Coder software.

  - Set **Start time** to `0.0` and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this parameter to `inf` for completeness.
  - Under **Solver options**, select the `fixed-step` and `discrete` settings from the lists. When you use PIL, select any setting on the **Type** and **Solver** lists.
  - Set the **Fixed step size** to `Auto` and the **Tasking Mode** to `Single Tasking`.

Note Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

Ignore the `Data Import/Export`, `Diagnostics`, and `Optimization` categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Simulink Coder Code Generation Options. To configure Simulink Coder software to use the correct processor files, compile your model, and run your model executable file, set the options in the `Code Generation` category of the model Configuration Parameters. Follow these steps to set the Simulink Coder software options to generate code tailored for your processor:

1 Select `Code Generation` on the **Select** tree.

2 In **Target selection**, click **Browse** to select the appropriate system target file for code generation—`multilink_grt.tlc` or `multilink_ert.tlc` (if you use Embedded Coder software). The correct target file might already be selected.

Clicking **Browse** opens the **System Target File Browser** to allow you to change the system target file.

3 On the **System Target File Browser**, select the proper system target file `multilink_grt.tlc` or `multilink_ert.tlc`, and click **OK** to close the browser.

Setting Embedded Coder Code Generation Options. After you set the Simulink Coder options for code generation, set the options that apply to your Embedded Coder software run-time and build processes.

1 From the **Select** tree, choose `IDE Link` to specify code generation options that apply to the processor.

2 Set the following **Runtime** options:

   - **Build action**: `Create_project`.
   - **Interrupt overrun notification method**: `Print_message`.

3 (optional) Under **Link Automation**, verify that **Export MULTI link handle to base workspace** is selected and provide a name for the handle in **MULTI link handle name**.

4 If you are using an actual board, identify a Board Support Package (BSP) in the **Compiler options string** (under **Project options**). For example, enter "`-bsp=at91rm9200`". If you do not provide this type of information, the software can generate errors that do not identify the absence of linker directives as the cause.

5 Under **Code Generation**, clear all of the options.

6 Change the category on the **Select** tree to `Hardware Implementation`.

7 Verify that the Device type is the correct value for your processor—`Analog Devices`, `NEC`, or `Freescale`.

You have configured the Simulink Coder options that let you generate a project for your processor. A few Simulink Coder categories on the **Select** tree, such as `Comments`, `Symbols`, and `Optimization` do not require configuration for use with Embedded Coder software. In some cases, set options in the other categories to configure other code generation features.

For your new model, the default values for the options in these categories are correct. For other models you develop, setting the options in these categories provides more information during the build process. Some of the options configure the model to run TLC debugging when you generate code. Refer to your Simulink and Simulink Coder documentation for more information about setting the configuration parameters.

Creating Your Project

After you set the configuration parameters and configure Simulink Coder to create the files you need, you direct Simulink Coder to create your project:

1. Click **OK** to close the Configuration Parameters dialog box.

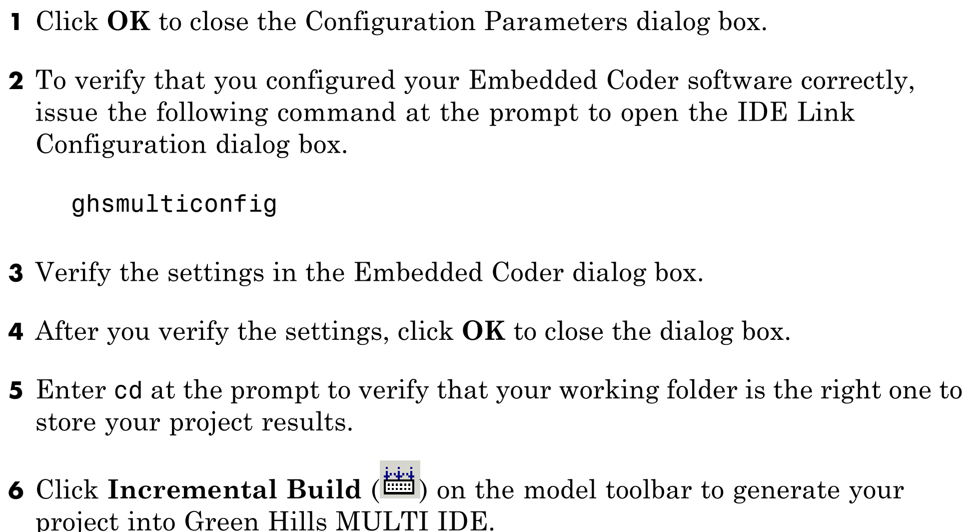
2. To verify that you configured your Embedded Coder software correctly, issue the following command at the prompt to open the IDE Link Configuration dialog box.

   ```
   ghsmulticonfig
   ```

3. Verify the settings in the Embedded Coder dialog box.

4. After you verify the settings, click **OK** to close the dialog box.

5. Enter `cd` at the prompt to verify that your working folder is the right one to store your project results.

6. Click **Incremental Build** () on the model toolbar to generate your project into Green Hills MULTI IDE.

   When you press with `Create_project` selected for **Build action**, the build process starts the Green Hills MULTI application and populates a new project.

Model Reference

- "About Model Reference" on page 50-40
- "How Model Reference Works" on page 50-40
- "Using Model Reference" on page 50-41

- "Configuring Targets to Use Model Reference" on page 50-42

About Model Reference

Model reference lets your model include other models as modular components. This technique is useful because it provides the following capabilities:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and then only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Simulink Coder documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. This discussion uses the following terms:

- The *Top model* is the root model block or model. It refers to other blocks or models. In the model hierarchy, this model is the topmost model.
- *Referenced models* are blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Simulink Coder documentation in the online Help system.

Model Reference in Simulation. When you simulate the top model, Simulink Coder detects that your model contains referenced models. Simulink generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (`.mex` file) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these settings through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation. Simulink Coder requires executables to generate code from models. If you have not simulated your model at least once, Simulink Coder creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Simulink Coder calls `make_rtw` on the top model. The call to `make_rtw` links to the library files Simulink Coder created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded Coder software provides full support for generating code from models that use model reference.

Build Action Setting. The most important requirement for using model reference with the Green Hills MULTI software supported processors is you must set the **Build action** (select **Configuration Parameters > IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action, perform the following steps:

1. Open your model.
2. Select **Simulation > Configuration Parameters** from the model menus.

The Configuration Parameters dialog box opens.

3 From the **Select** tree, choose IDE Link.

4 In the right pane, under **Runtime**, select set `Archive_library` from the **Build action** list.

If your top model uses a reference model that does not have the build action set to `Archive_library`, the build process automatically changes the build action to `Archive_library` and issues a warning about the change.

Selecting `Archive_library` disables the **Interrupt overrun notification method**, **Export MULTI link handle to the base workspace**, and **System stack size** options for the referenced models.

Target Preferences Blocks in Reference Models. Each referenced model and the top model must include a Target Preferences block for the correct processor. Configure all the Target Preferences blocks for the same processor.

The referenced models need Target Preferences blocks to provide information about which compiler and which archiver to use. Without these blocks, the compile and archive processes do not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations. Model reference with Embedded Coder software code generation options does not allow you to use noninlined S-functions in reference models. Verify that the blocks in your model do not use noninlined S-functions.

Configuring Targets to Use Model Reference

When you create models to use in Model Referencing, keep in mind the following considerations:

- Your model must use a system target file derived from the ERT or GRT target files.

- When you generate code from a model that references other models, configure the top-level model and the referenced models for the same system target file.
- Simulink Coder builds and Embedded Coder software projects do not support external mode in model reference. If you select the external mode option, it is ignored during code generation.
- Your TMF must support use of the shared utilities folder, as described in Supporting Shared Utility Directories in the Build Process in the Simulink Coder documentation.

To use an existing processor, or a new processor, with Model Reference, set the `ModelReferenceCompliant` flag for the processor. For information about setting this option, refer to `ModelReferenceCompliant` in the online Help system.

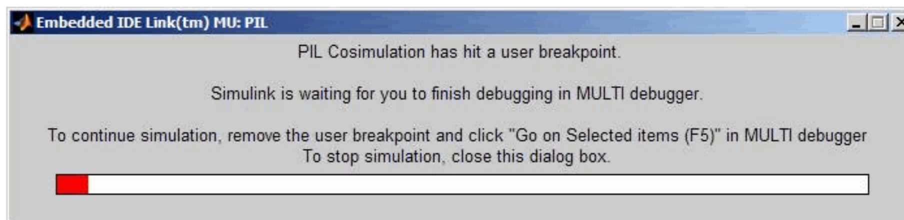
If you start with a model that was created before MATLAB release R14SP3, use the following command to make your model compatible with model reference :

```
% Set the Model Reference Compliant flag to on.
set_param(bdroot,'ModelReferenceCompliant','on')
```

Code that you generate from Simulink models by using Embedded Coder software includes the model reference capability. You do not need to set the flag.

Breakpoints and PIL

Green Hills MULTI debugger allows you to add breakpoints to your projects. When you run a PIL simulation that includes added breakpoints, the following dialog box appears:



The dialog box gives you two options:

- Stop the running simulation by closing the dialog box.
- Go to MULTI, remove the breakpoint you added, and press **F5** to continue running your simulation.

Working with Infineon C166 Processors

- "Getting Started" on page 51-2
- "Tutorial: Simple Example Applications for C166 Microcontrollers" on page 51-22
- "Integrating Your Own Device Drivers" on page 51-38
- "Custom Storage Class for C166 Microcontroller Bit-Addressable Memory" on page 51-49
- "Execution Profiling" on page 51-56
- "Configuration Parameters" on page 51-71

Note Support for the Infineon C166 processor will be removed in a future release of your MathWorks software.

Note The information in this chapter describes Embedded Coder features and user interfaces that are unique to the Infineon C166 processor. Do not generalize this information to Embedded Coder support for other IDEs.

Getting Started

In this section...

"Overview" on page 51-2

"Using This Guide" on page 51-4

"Supported Hardware for Infineon C166" on page 51-5

"Supported Cross-Development Tools for Infineon C166" on page 51-7

"Switching Between Hardware Variants" on page 51-7

"Setting Up and Verifying Your Installation" on page 51-8

"Setting Up Your Target Hardware" on page 51-12

"Setting C166 Target Preferences" on page 51-14

"Code Generation Configuration for Nondefault Processors" on page 51-15

"Supported Blocks and Data Types" on page 51-18

"Accessing Utilities for Infineon® C166" on page 51-20

"Overview of C166 Options in the Configuration Parameters Dialog Box" on page 51-20

Overview

- "Introduction" on page 51-2
- "Feature Summary" on page 51-3

Introduction

Note Support for the Infineon C166 processor family will be removed in a future release of your MathWorks software.

The coder product is provides a set of tools for developing embedded applications for the C166 family of processors from Infineon (`http://www.infineon.com/`). This includes derivatives such as Infineon C167 and XC16x, and ST Microelectronics ST10 (`http://www.us.st.com`).

Used in conjunction with the Simulink, Stateflow, the coder product lets you

- Design and model your system and algorithms.
- Compile, download, run and debug generated code on the target hardware, seamlessly integrating with industry-standard compilers and development tools for the C166 microcontroller.
- Deploy production code on the target hardware.

Feature Summary

The coder product provides the following capabilities:

- A flexible build process, which allows you to automatically create and build projects in the TASKING EDE using code generated by the coder product.
- Customizable project templates for targeting embedded hardware or instruction set simulator.
- Processor-in-the-Loop (PIL) simulation techniques to verify generated code running in an instruction set simulator or real embedded hardware environment. You can set breakpoints, step through the code, and watch variables during simulation.
- MATLAB commands to rapidly and easily interact with projects in the TASKING EDE or debug generated code in the CrossView Pro debugger.
- Execution profiling and code coverage reports are returned from the TASKING EDE to MATLAB for your review.

The coder product also provides these features:

- Automatic generation of the `main` program including singletasking or preemptive multitasking scheduler
- Scheduler is configurable to allow temporary overruns
- Automated build procedure including starting debugger or download utility
- Support for integer, floating-point, or fixed-point code
- Driver blocks for serial transmit and receive
- Driver blocks for CAN message transmit and receive

- Examples to show you how to integrate your own driver code
- Enhanced HTML report generation provides analysis of RAM/ROM usage; this is in addition to the standard HTML report generation that shows optimization settings and hyperlinks to generated code files
- Support for CAN Calibration Protocol
- External mode for parameter tuning and signal logging

Using This Guide

Follow this path to get acquainted with the coder product and gain hands-on experience with the features most relevant to your interests:

- Read in its entirety, paying particular attention to “Setting Up and Verifying Your Installation” on page 51-8.
- If you are interested in using the device driver blocks supplied with the coder product, and in deploying stand-alone, real-time applications on the C166 microcontroller, read “Tutorial: Simple Example Applications for C166 Microcontrollers” on page 51-22 Work through the “Tutorial: Creating a New Application” on page 51-22.
- Then, if you are interested in using the coder product for integrating automatically generated code with your own manually written device driver code, see “Integrating Manually Coded Device Drivers with a Simulink Model” on page 51-38. Work though the example provided in “Tutorial: Using the Example Driver Functions” on page 51-43.
- See “Custom Storage Class for C166 Microcontroller Bit-Addressable Memory” on page 51-49 to find out how to use the coder product to take advantage of C166 bit-addressable memory. This can significantly reduce code size and increase execution speed. There are examples provided in “Using the Bitfield Example Model” on page 51-50.
- It is particularly important to read C166 Resource Configuration, as the C166 Resource Configuration block is required to use the device driver blocks.

We recommend you work through the tutorials in this User’s Guide with step-by-step instructions for using and understanding these demos.

Supported Hardware for Infineon C166

The coder product may be used to generate programs that can run on any development board or Electronic Control Unit (ECU) that is based on the C166 microcontroller.

The coder product is supplied with default configurations that have been tested on hardware listed in the following table:

Supplier	Board	Processor	Other Information
Phytec	phyCORE®-167 Rapid Development Kit	SAF-C167CS-LM or SAK-C167CS-LM	Product code KPCM-009-C1U: C167CS. For other details, see phyCORE-167
Phytec	phyCORE®-ST10 Rapid Development Kit	ST10F269Z2Q3	Supported, but no longer available commercially
Phytec	kitCON-167 C167CR	SAB-C167CR-LM	A newer board kitCON-16x (product code KC-167-KSM04) is available. See Phytec products. The coder product has not been tested on the newer board.
Infineon	XC167CI Starter Kit	SAK-XC167CI	Supported, but no longer available commercially

Supplier	Board	Processor	Other Information
Infineon	XC164CM U CAN Start Kit	SAK-XC164	Supported, but no longer available commercially
STMicroelectronics	MB449 ST10F25x EVA Board	ST10F252-ABG	Supported, but no longer available commercially

You can switch easily between these configurations. For other hardware variants, you will need to change the default configuration settings. For details on this and other requirements, see “Switching Between Hardware Variants” on page 51-7.

This guide assumes that you are working with the Phytec phyCORE-167CS development board, and documents specific settings and procedures for use with the Phytec phyCORE-167CS board, in conjunction with specific cross-development environments.

If you use a different development board, you may need to adapt these settings and procedures for your development board.

CAN Hardware

If you want to use CAN to transmit or receive CAN messages between your host PC and your target, you require Vector-Informatik CAN hardware supported by the Vector CAN Driver Library. You must install the correct driver libraries to support profiling, downloading, and external mode.

Note For CANcaseXL, you must install *both* the Vector XL-driver library and Vector CAN Driver Library `vcand32.dll`.

For older CAN hardware, you must install the Vector CAN Driver Library `vcand32.dll`.

Make sure that the library, `vcand32.dll`, is placed in the Windows `system32` folder.

Supported Cross-Development Tools for Infineon C166

In addition to the required MathWorks software, a supported cross-development environment is required.

- MiniMon freeware download and monitor utility

Before using the coder product with the above cross-development tools, please be sure to read and follow the instructions in "Setting Up and Verifying Your Installation" on page 51-8.

Switching Between Hardware Variants

There are many different members of the C166 microcontroller family, e.g., C167CS, ST10, XC167CI. For each of these processors, it is appropriate to use different compiler switches and link libraries. Even if you are working with a single processor variant, you may need to build for different memory configurations, for example, depending on whether the application will run from RAM or flash memory. The compilation settings are captured in the project file.

The coder product is supplied with preconfigured projects for targeting the hardware and simulator for a set of processor variants — see "Supported Hardware for Infineon C166" on page 51-5. If your hardware variant is not in this set, then you need to create a new template project (see "Tutorial: Creating New Template Projects" on page 46-76) and set the C166 code generation options (see "Code Generation Configuration for Nondefault Processors" on page 51-15).

When switching between target configurations, you should review your option set and ensure that options are set appropriately for the new configuration.

Additionally, for each model that you build, you must check, and, if necessary, change the following settings in the C166 Resource Configuration block:

- `System_frequency`
- `External_oscillator_frequency`

To determine the correct value of these parameters, consult your hardware documentation.

It is possible to make all the required changes programmatically: a convenience function `c166switchconfig` is provided for this purpose. This function can be run by double-clicking the block Switch Target Processor Variant inside any of the demo models.

Setting Up and Verifying Your Installation

- "Setting Up Software" on page 51-8
- "Verifying MiniMon Settings" on page 51-9

Setting Up Software

Install the Tasking C Cross-Compiler and CrossView Pro Debugger by following the instructions provided by Altium Limited.

If the CrossView connection to your target hardware requires a serial connection, install the MiniMon download utility. By using MiniMon instead of CrossView to launch your application, the serial connection will be available for other purposes, if required. If your CrossView connection is via a debug interface (for example, on XC16x hardware) then it is not necessary to install MiniMon.

You can obtain the MiniMon download utility for monitoring the serial interface from the Infineon Web site at this URL:

```
http://www.infineon.com/
```

To download the MiniMon utility:

1. Go to the Infineon Web site, and click the `sitemap`.

2. Select Product Categories > Microcontrollers > Development Tools, Software and Training -> C166/XC166 Development Tools and Software > Software Downloads.

   Find MiniMon in the table, and download and install Minimon. Version 2.2.33 has been verified with this product.

Minimon may need to be configured for your target processor.

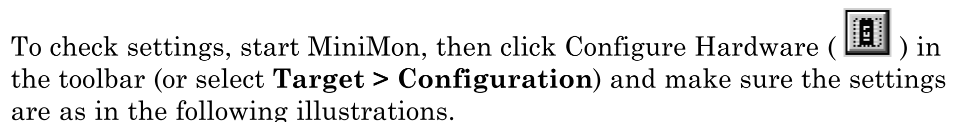
After you install, you must specify the location of MiniMon in the `BootstrapLoaderExe` target preference, as detailed in "Setting C166 Target Preferences" on page 51-14. Check that MiniMon is correctly configured for your target, as detailed in the next section, "Verifying MiniMon Settings" on page 51-9.

The next sections describe how to configure your development environment (compiler, debugger, etc.) for use with the coder product, and how to verify correct operation. The initial configuration steps are described in the following sections:

- "Setting Up Your Target Hardware" on page 51-12
- "Setting C166 Target Preferences" on page 51-14

Verifying MiniMon Settings

You must check that MiniMon has the correct target settings. This section describes MiniMon configuration settings that work for the C167CR processor and for the C167CS processor. Settings for the C167CS board also work successfully with the ST10F269. You may be able to use MiniMon to download onto other processors, however, you must establish a corresponding MiniMon configuration.



To check settings, start MiniMon, then click Configure Hardware ( ) in the toolbar (or select **Target > Configuration**) and make sure the settings are as in the following illustrations.

In general, you should choose configuration settings that are consistent with the values specified in the Tasking EDE project.

Select **Settings > Interface** and ensure that the settings for the serial interface match those in the Resource Configuration block of your model.

To set up a configuration for a C167CR:

1. Select `C167CR` from the **Controller type** drop-down list.

2 Click **Yes** three times when prompted by the dialog boxes asking the following questions:

   a Do you want to load default memory units for this Type?

   b Do you want to activate the default kernel for this Type?

   c Do you want to load default initialization registers of this Type?

3 Perform the following steps on the Initialize register settings:

   a Set SYSCON to 0085.

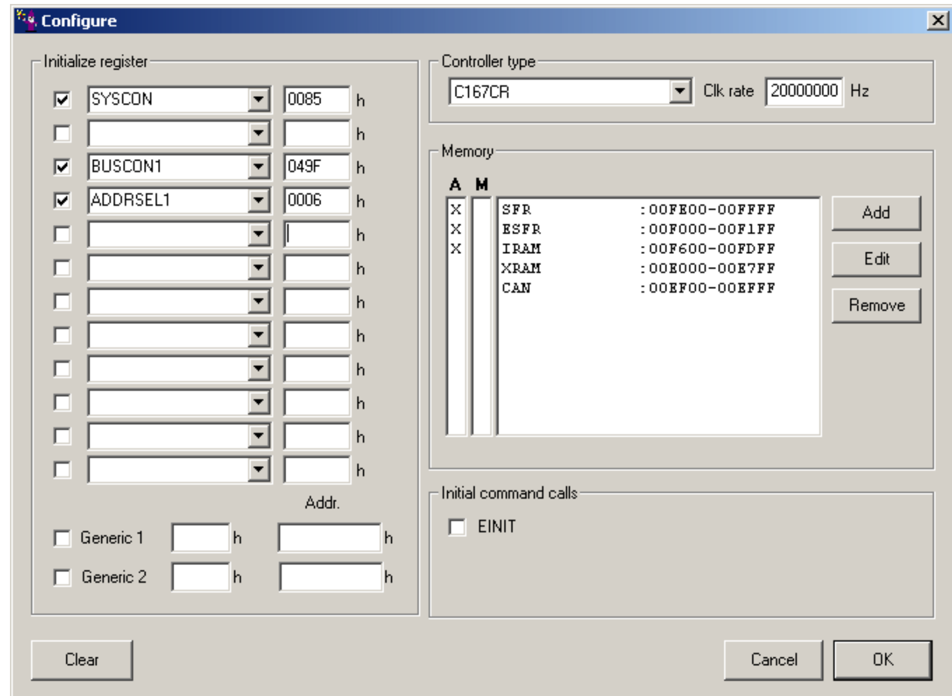
   b Set BUSCON1 to 049F.

   c Set ADDRSEL1 to 0006.

   d Clear all the other check boxes.

   The register settings should look as shown.

This configuration has been verified with a Phytec kc167 (C167CR).



To set up a configuration for a C167CS or ST10F269:

1 Select `C167CS-4RM` from the **Controller type** drop-down list.

2 Click **Yes** three times when prompted by the dialog boxes asking the following questions:

   a Do you want to load default memory units for this Type?

   b Do you want to activate the default kernel for this Type?

   c Do you want to load default initialization registers of this Type?

3 Perform the following steps on the Initialize register settings:

   a Change SYSCON to XPERCON, and set the value to 0403.

   b Change SYSCON1 to SYSCON, and set the value to 0085.

   The order is important: XPERCON must be above SYSCON.

c Set BUSCON1 to 049F.

d Set ADDRSEL1 to 0006.

e Clear all the other check boxes.

The register settings should look as shown.

This second configuration has been verified with phyCORE-C167CS and on phyCORE-ST10F269 hardware.

Setting Up Your Target Hardware

- "Jumper Settings for the phyCore-167 Development Board" on page 51-13
- "Setting Up XC164CM Hardware" on page 51-13
- "Jumper Settings for the STMicrolectronics MB449 ST10F25x EVA Board" on page 51-14

Jumper Settings for the phyCore-167 Development Board

This section describes the required connections and jumper settings for the phyCORE-167CS module with HD200 development board.

After setting up your board, you must configure target settings associated with the coder product, as described in the next section.

1. Configure jumpers as detailed in the instructions found in the phyCORE QuickStart documentation. Note that these settings can be markedly different from the configuration fresh out of the box.

2. If you are running applications from RAM only, it is useful if the board starts up in bootloader rather than execution mode. There is one jumper setting that needs to be changed to achieve this: close pins 1 and 2 on JP10. This is optional; if you do not close this jumper, then when you download to the target, you need to keep the Boot switch depressed while pressing the Reset button.

Connect the supplied power cable to the board, and use the serial cable to connect the serial port P1 on the board to the serial port of your PC.

Setting Up XC164CM Hardware

See the product help for information on software installation.

If you need to profile on XC164CM hardware via the serial port, this is possible when using CrossView. Check which COM port is assigned to USB COM Port. To access the Device Manager where you can see this information, select Windows **Start > Settings > Control Panel**, double-click `System`, select the **Hardware** tab, and click **Device Manager**.

This information can be passed to the `profile_c166` command as follows:

```
% assuming that it was assigned to COM4
profile_c166('serial','SerialPort','COM4')
```

The MiniMon hyperlink may not be provided at the end of builds for hardware (e.g. `XC164CM U CAN` and `xc167ci_hw_usb`) that has a JTAG interface available. This is because this hardware uses the JTAG debug interface instead of a serial connection to ASC0. For hardware with a JTAG interface

there is no conflict between using the CrossView debugger simultaneously with the ASC0 serial interface: in these cases it is recommended always to use CrossView for downloading and running applications.

Jumper Settings for the STMicroelectronics MB449 ST10F25x EVA Board

Settings not listed here should be as default, as specified by the board manual.

Type	Jumper Settings
Boot / Configuration Mode > SW4	Switch 4-5-6: CLKCFGSwitch State: on-on-off. Note: With fCPU = 5* fXTAL and oscillator frequency of 8MHz, the system frequency, fCPU, is 40MHz
	Switch 2-3: SALSEL Switch State: off-off
	Switch 1: WRCSwitch State: off
Boot / Configuration Mode > SW3	Switch 7-8: BUSTYPSwitch State: off-off
	Switch 5-6: BSLSwitch State: on-off
	Switch 2: ADPSwitch State: off
External Memory	J1, 1-2: ClosedNote: Enables external memory
Reset and Vstby EA jumpers	J4, 2-3: Closed Note: Forces EA pin to Vcc level
CAN	J11, 1-2: ClosedNote: Connects onboard CAN transceiver
	J11, 3-4: Closed Note: Connects onboard CAN transceiver

Setting C166 Target Preferences

This section describes configuration settings associated with the coder product. These settings, which persist across MATLAB sessions and different

models, are referred to as *target preferences*. Target preferences let you specify the location of your cross-compiler and other parameters affecting the generation, building, and downloading of code.

1. First you must set up your target preferences to specify the location of your cross-compiler and other settings. See "Setting Target Preferences for Altium TASKING" on page 46-8.

2. Enter `c166utils` in the Command Window to open the Utilities for Use with C166 dialog box.

3. Select `Target Preferences`, and click **OK**. This opens the Target Preferences dialog box.

4. Edit the settings for your cross-development environment:

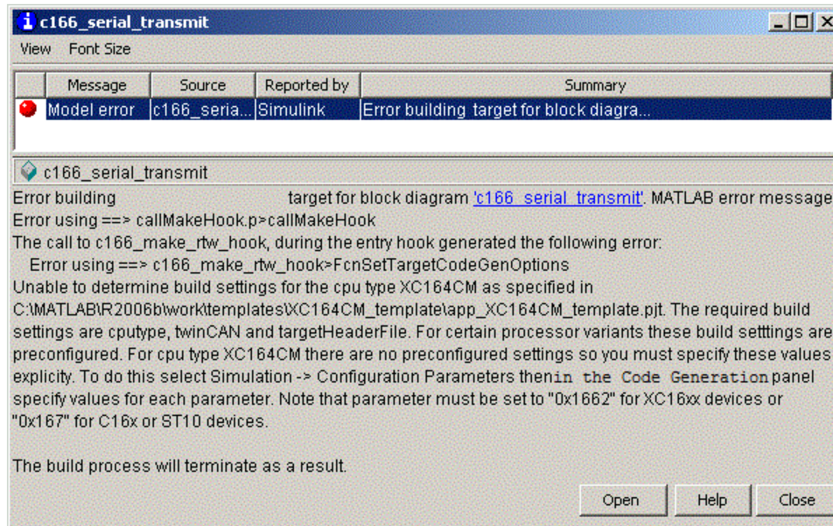
   - `BootstrapLoaderExe` specifies the path to your download utility (MiniMon).

You must check this path and also verify that the Target Preferences are correct for your machine. You may need to localize these paths to suit your PC. You can edit a path by clicking on it. The drive designated in the path must be either an actual hard drive on your PC, or a mapped drive. Do not use a Universal Naming Convention (UNC).

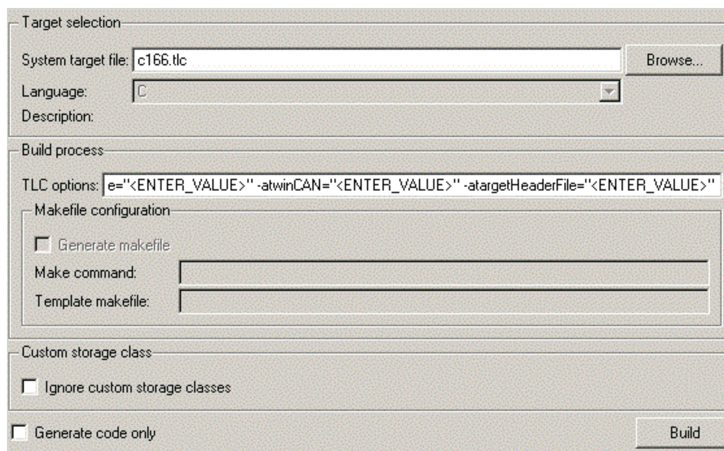
Code Generation Configuration for Nondefault Processors

If you wish to target nondefault processor types, then you need to set some code generation options in the **TLC Options** of your model's Configuration Parameters.

If you are using a template that specifies a nondefault processor type (see "Tutorial: Creating New Template Projects" on page 46-76 in the product help), when you try to build the model, you see a build error message similar to the one in the following figure.



When you open the Configuration Parameters dialog box, the parameters you need to set now appear in the TLC Options field. You must replace the string `<ENTER VALUE>` for each of the parameters `cpuType`, `twinCAN`, and `targetHeaderFile`. The following example shows these parameters before the strings `<ENTER VALUE>` are replaced.



An example configured for an XC164CM is shown in the following figure.

Summary of Parameters

The **TLC Options** edit box includes the following parameters:

`cpuType`

- 0x167, for C16x and ST10 type processors
- 0x1662, for XC16x type processors

`twinCAN`

- 0 – disabled, for use with processors without TwinCAN support
- 1 – enabled, for use with processors with TwinCAN support

`targetHeaderFile` — The file name of the header file for your processor type. These are found in *`TASKING ROOT`*`\include` folder.

Typical Parameter Configuration

The following table shows a configuration matrix for the parameters `cpuType`, `twinCAN` and the typical configurations used for the processor variants supported by the product.

Processor Type	CPU type	TwinCAN
16x, ST	0x167	0 - disabled
XC	0x1662	1 – enabled

Note Driver blocks may not work on unsupported processors.

Supported Blocks and Data Types

The coder product supports the same blocks and data types as the coder product.

Note however

- You should not use IEEE values `Inf` or `NaN` in your model: these are not supported and result in an error.
- Floating point support is implemented in the software; if speed and ROM usage are of concern, you should select the option for integer code and avoid the use of floating-point values in your model. This is detailed in step 9 of "Tutorial: Using the Example Driver Functions" on page 51-43.

The coder product provides one block library, containing seven sublibraries that support different functions, as follows:

- C166 Drivers Library
  - Asynchronous/Synchronous Serial Interface Sublibrary
  - CAN Interface Sublibrary
  - Execution Profiling Sublibrary
  - TwinCAN Interface Sublibrary
  - Interrupts Sublibrary
  - Utilities Sublibrary
  - Digital Input/Output Sublibrary

You can click **Help** on the Block Parameters dialog box for the block or access the block reference page through Help.

The top-level C166 Drivers library contains the C166 Resource Configuration block. This block supports driver configuration for C166 microcontrollers and is required if there are device driver blocks in the model. See C166 Resource Configuration.

The C166 Resource Configuration block provides information required for generating timer interrupt code. If you do not include a C166 Resource Configuration block in your model, the code simply executes as fast as possible. That is, it is not synchronized to real time. This behavior may be desirable if you are running code on the debugger or hardware simulator.

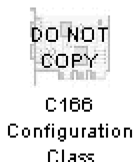
Caution When using device driver blocks from the Embedded Targets libraries with the C166 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the C166 Resource Configuration block operates incorrectly. See the "Top-Level Blocks"reference page for further information.

Model Reference and Driver Blocks

Referenced sub-models that contain driver blocks (including the C166 Resource Configuration block) cause build failures. All driver blocks from the coder product must be placed in the top level model. It is not possible to include driver blocks in any of the referenced sub-models.

Configuration Class Blocks

Each sublibrary of Embedded Targets library contains a *configuration class block* that has an icon similar to the one shown in this picture.



Caution Configuration class blocks exist only to provide information to other blocks. *Do not copy these objects into a model.* If you do you see an error dialog box to warn you. This causes build failures.

Accessing Utilities for Infineon C166

You can open the Utilities for Use with C166 dialog box by entering `c166utils` in the Command Window or double-clicking Launch C166 Utilities in the Simulink block library.

You will see the following options:

- **Target Preferences**. Select this to open the Target Preferences dialog box.
- **Download via Minimon**. Select this to use the Minimon utility to download your application to your target hardware.

Overview of C166 Options in the Configuration Parameters Dialog Box

When you select a C166 system target file in the Configuration Parameters dialog box, additional options appear in the tree: `C166 Options`, and `IDE Link`.

Select **C166 Options** under **Code Generation** in the Configuration Parameters dialog box.

Include input/output driver function hooks
Use this option to integrate your own device driver code. This is described in "Calling the Device Driver Functions from c166_main.c" on page 51-41.

The following are all execution profiling controls. See "Overview of Execution Profiling" on page 51-56.

Maximum number of concurrent base-rate overruns
Option for task execution profiling. See "Task Scheduler Overrun Options" on page 51-61.

Maximum number of concurrent sub-rate overruns
Option for task execution profiling. See "Task Scheduler Overrun Options" on page 51-61.

Execution profiling
Option for task execution profiling. See "Options for Execution Profiling" on page 51-60.

Number of data points
Option for task execution profiling. See "Options for Execution Profiling" on page 51-60.

When you select a C166 system target file, the Configuration Parameters component is automatically added to the model.

The **Target Preferences Configuration** description is automatically set to `C166` to use the predefined C166 project templates. The IDE Link options contain settings for configuring the build process.

Tutorial: Simple Example Applications for C166 Microcontrollers

This section includes the following topics:

In this section...
“Introduction” on page 51-22
“Tutorial: Creating a New Application” on page 51-22
“Debugging and Using The Code Profile Report” on page 51-30
“Parameter Tuning and Signal Logging” on page 51-34

Introduction

This section describes how to use two example models to generate, download and run stand-alone real-time applications for the C166 microcontroller. The components required to generate stand-alone code are

- The embedded target
- The example models provided: `c166_serial_transmit` and `c166_serial_io`
- The Tasking C Cross-Compiler and Tasking CrossView Pro Debugger for compiling and downloading generated code to the target hardware

  As an alternative to CrossView, you can use the MiniMon utility for downloading an application to your target hardware.

Using these, you can build the complete application. You do not need to manually write any C code to integrate the generated code into a final application.

The tutorial “Tutorial: Creating a New Application” on page 51-22 uses two blocks from the Embedded Targets library.

Tutorial: Creating a New Application

- “Tutorial Overview” on page 51-23

- “Before You Begin” on page 51-23
- “Example Model 1: c166_serial_transmit” on page 51-24
- “Generating and Downloading Code” on page 51-26
- “Example 2: c166_serial_io” on page 51-28

Tutorial Overview

In this tutorial, you build stand-alone real-time applications from models incorporating blocks from the Embedded Targets library.

In the following sections, you will

- Examine two models
- Generate code from the models
- Download and run the code automatically as part of the build process
- Use MiniMon to monitor the code executing on the target

Before You Begin

We assume that you are already familiar with Simulink software and with the code generation and build process. This tutorial requires the following specific hardware and software in addition to the coder product:

- Phytec phyCORE-167CS development board, connected via serial port to your PC
- Tasking C Cross-Compiler and CrossView Pro Debugger
- MiniMon download utility

You must make sure the target preferences have been set correctly. See “Setting C166 Target Preferences” on page 51-14.

Note Make sure the `default.ini` file in the MiniMon folder is not read only. This can cause errors.

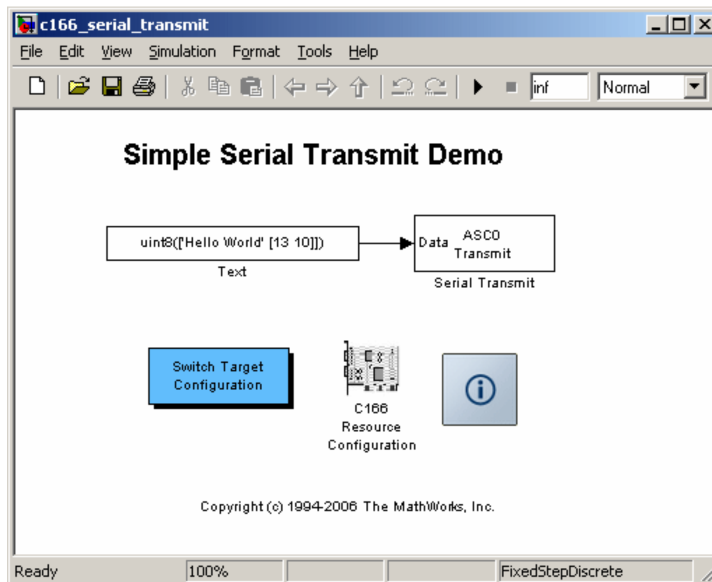
Example Model 1: c166_serial_transmit

In this tutorial you start with a simple example model, `c166_serial_transmit`, from the folder *`matlabroot`*`/toolbox/rtw/targets/c166/c166demos`.

This folder is on the default MATLAB path.

1 Open the model by typing `c166_serial_transmit` at the command line.

   This example shows the tutorial model `c166_serial_transmit` at the root level.



   The model contains a C166 Resource Configuration object. When building a model with driver blocks from the Embedded Targets library, you must always place a C166 Resource Configuration object into the model (or the subsystem from which you want to generate code) first.

   The purpose of the C166 Resource Configuration object is to provide information to other blocks in the model. Unlike conventional blocks, the C166 Resource Configuration object is not connected to other blocks via input or output ports. Instead, driver blocks (such as the ASC0

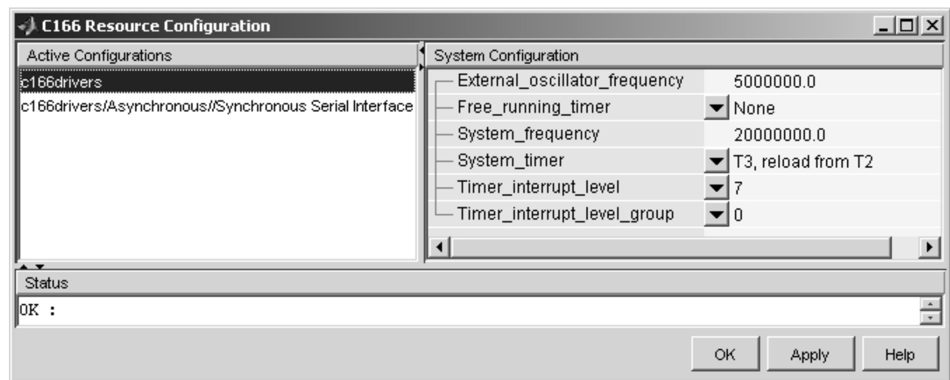
Serial Transmit block in the example model) query the C166 Resource Configuration object for required information.

For example, a driver block may need to find the system clock speed that is configured in the C166 Resource Configuration object. The C166 microcontroller has a number of clocked subsystems; to generate correct code, driver blocks need to know the speeds at which these clock busses will run.

The C166 Resource Configuration window lets you examine and edit the C166 Resource Configuration settings.

2 Double click the switch target configuration block, and then select `c167cs_hw`. This selection sets the appropriate `System_frequency` and `External_oscillator_frequency` in the Resource Configuration block and options set. See "Option Sets" on page 46-24 in the product help for more information.

3 To open the C166 Resource Configuration window, double-click the C166 Resource Configuration icon. The picture following shows the C166 Resource Configuration window for the `c166_serial_transmit` model.

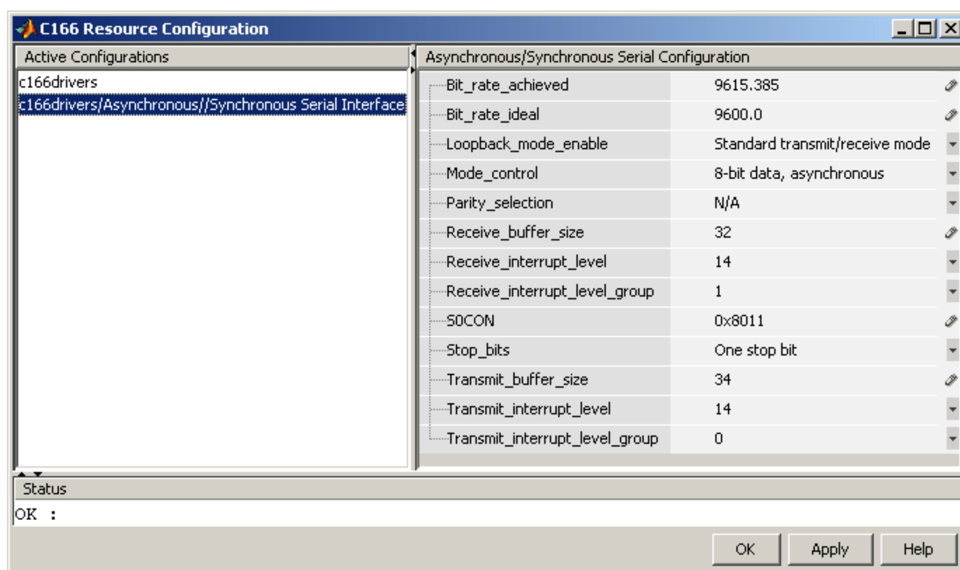


In this tutorial, use the default C166 Resource Configuration settings.

Note If hardware is running at a system frequency other than 20 MHz, you must change this parameter appropriately.

Otherwise, observe, but do not change, the parameters in the **C166 Resource Configuration** window. By default, the **c166drivers** configuration is selected. This shows parameters for the C166 microcontroller CPU in the **System Configuration** pane on the right.

4 View the settings for the serial driver block by clicking the `c166drivers/Asynchronous/Synchronous Serial Interface` option in the **Active Configurations** pane. These settings are shown in the following illustration.



The settings appear in the **Asynchronous/Synchronous Serial Configuration** pane on the right. Do not edit any of these parameters for this tutorial. To learn more about the C166 Resource Configuration object, see C166 Resource Configuration.

5 Close the **C166 Resource Configuration** window before proceeding.

Generating and Downloading Code

To generate code for the model:

1 Select **Simulation > Configuration Parameters**.

The Configuration Parameters dialog box opens.

2 Select **Code Generation** in the tree.

3 Click **Build**.

Alternately, you can go straight to building the model by selecting **Tools > Code Generation > Build Model** or using the shortcut **Ctrl+B**.

Watch the progress messages in the command window as code is generated.

4 Enter `c166utils` in the Command Window to open the Utilities for Use with C166 dialog box. Select **Download via Minimon** to download your application via the MiniMon link. If the Minimon link has not been generated then your C166 option set is not compatible with MiniMon downloads. This failure to generate could be because you are targeting one of the simulator configurations or your board is using OCDS (on-board wiggler) to connect to the target. The Minimon option should appear when:

- The Build Action is set to **Create and Build Application Project**, **Create, Build and Execute Application Project**, or **Create, Build and Debug Application Project**.
- The option set is for hardware (rather than simulator).
- You are using a serial connection to connect to your target.
- If you have created your own template projects, the option to generate a hex file must be selected.

Caution You must ensure the option to generate a hex file is turned on. If you do not you will see the following warning:

```
It was not possible to generate a minimon script for this
 build. This is because your EDE project template is not
configured to generate a .hex file which is required by
Minimon. To generate a .hex file as part of the build
you need to check the box 'Intel HEX records' in your
EDE project template.
You can change this option via Project -> Project Options
 -> Linker/Locator -> Output Format.
```

When MiniMon is started, a dialog box appears asking you to reset your hardware.

5. Press the Reset button on your phyCORE-167CS board or cycle the power, and then click **OK**.

   You can see progress messages in the MiniMon window as it connects and then downloads to the target. MiniMon then disappears and the code begins executing on the target.

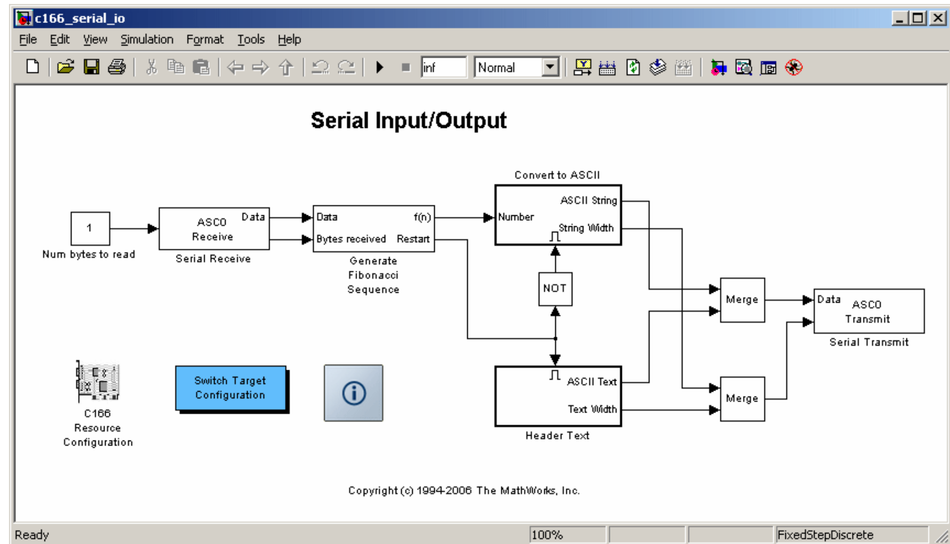
Verifying Execution on the Target.

1. Start MiniMon (navigate to `MiniMon.exe` and double-click).
2. Watch the model output in the MiniMon window. When the application is running, it sends the text "Hello World" plus a carriage return and a linefeed over the serial interface.

Example 2: c166_serial_io

This example model demonstrates how to use both serial transmit and receive blocks for the C166 microcontroller. You could use these blocks in this way with your own Simulink models.

1. Open the model by typing `c166_serial_io` at the command line.



2 Double click on the switch target configuration block, then select `c167cs_hw`. This will set the `System_frequency` and `External_oscillator_frequency` in the Resource Configuration block and the option set.

3 Press **Ctrl+B** or select **Tools > Code Generation > Build Model**.

Watch the progress messages as code is generated from the model.

4 You can download the application by clicking on the link at the end of the build log. This link launches Minimon.

MiniMon is started to download the code to the target over the serial connection. The MiniMon dialog box appears asking you to reset your hardware.

5 Press the Reset button on your phyCORE-167CS board or cycle the power, and then click **OK**.

You can see progress messages in the MiniMon window as it connects and then downloads to the target. MiniMon then disappears and the code begins executing on the target.

You can restart MiniMon to monitor the serial interface.

Verifying Execution on the Target.

1 Start MiniMon (select **Start > Programs > MiniMon > MiniMon** in Windows, or navigate to `MiniMon.exe` and double-click).

2 Watch the model output in the MiniMon window. When the application is running, it generates a sequence of 16–bit numbers, converts them to ASCII characters, and transmits them over the serial interface.

3 If you enter the character `r` in the MiniMon command line field, the application restarts at the beginning of the sequence. Examine the model to see how this works: the Serial Receive block passes the restart command through to the Generate Fibonacci Sequence subsystem. This subsystem checks for the restart command.

Debugging and Using The Code Profile Report

- "Starting the Debugger on Completion of the Build Process" on page 51-30
- "RAM / ROM Code Profile Report" on page 51-31

Starting the Debugger on Completion of the Build Process

As an alternative to downloading with MiniMon at the end of the build process, you can start your debugger. Depending on the features provided by your debugger, you can debug the application either on-chip or on a hardware simulator.

For this example, you use another demo model, `c166_user_io`. This model is designed to show you how to integrate your own manually coded device drivers with automatically generated code using the coder product. This model is covered in detail in "Integrating Your Own Device Drivers" on page 51-38. You use it as an example here because you will typically need to use the debugger in cases where you are integrating your own code.

Also, note that running the debugger on-chip over the serial interface conflicts with the serial transmit and receive blocks. The `c166_user_io` model does not use serial blocks, so this avoids serial conflicts for this example. If you need to debug an application that includes the serial transmit and receive blocks, you must run the debugger using a hardware simulator; alternatively, it may be

possible to run your debugger on-chip without using the serial interface, for example, if debugging over CAN or JTAG is available.

1. Open the model `c166_user_io`.
2. Select **Simulation > Configuration Parameters**.
3. Select **IDE Link** in the tree.
4. Select the **Build action** `Create, Build and Debug Application Project`.
5. Before generating code, check that your target preferences related to the debugger are correctly configured. See “Setting C166 Target Preferences” on page 51-14.
6. Click **OK**.
7. Right-click the controller subsystem and select **Code Generation > Build Subsystem**.
8. Click **Build** in the next dialog box.

   Watch the progress messages in the command window as code is generated. At the end of the build process, your debugger launches automatically with the application ready to run. You may now debug the application.

Note If your model contains a serial transmit or receive block, it is not possible to perform on-chip debugging over the same serial interface. If you attempt to use the debugger once your application is running, you will no longer be able to control the application from the debugger, because the application is using the serial channel.

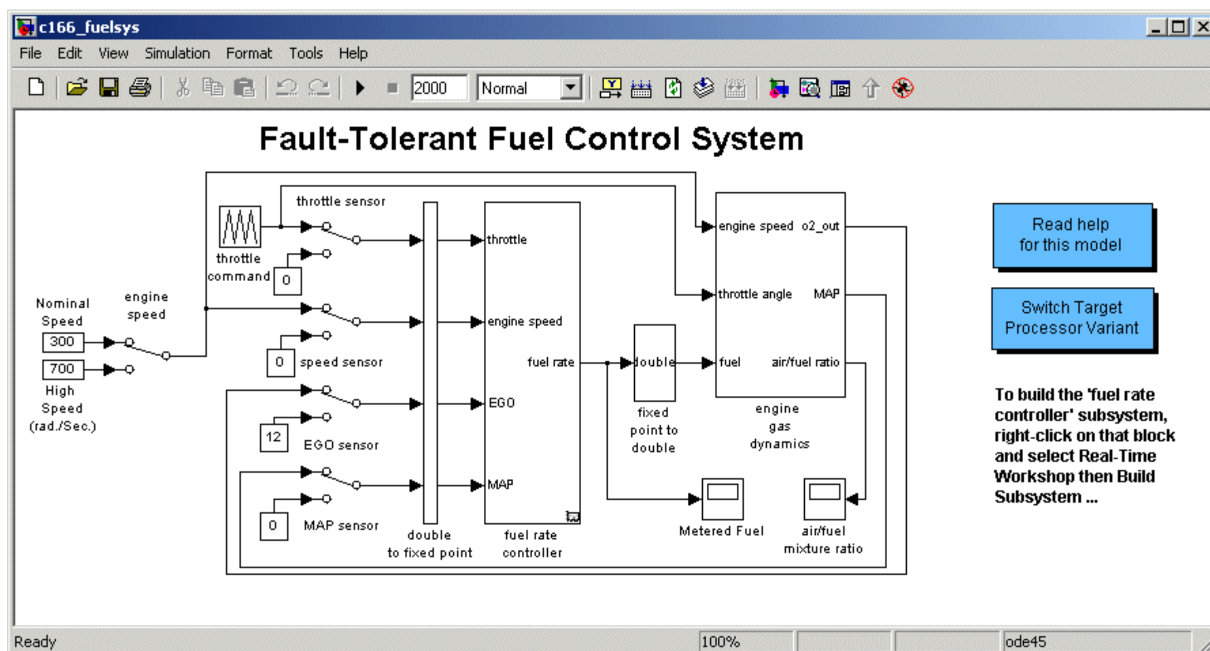
RAM / ROM Code Profile Report

The `c166_fuelsys` model is derived from the `fuelsys` demo model. The floating point control algorithm from the original model has been converted to fixed point to allow efficient code generation for the Infineon C166 microcontroller.

Note This demo requires the Simulink Fixed Point product.

The complete model includes a plant simulation as well as a fixed-point implementation of the control algorithm. When you generate code for this example, be sure to generate code for the control algorithm subsystem only:

1 Open the model `c166_fuelsys`.



2 Select **Simulation > Configuration Parameters**.

3 Select **Code Generation** in the tree. Note that the **Generate code only** option is not selected. The reason for this step is that the code generation report obtains information from MAP files that are created by your cross-compiler during the build process. If the **Generate code only** option is on, these files are not generated, which prevents the generation of the code generation report.

4. Select **Report** in the tree, and then observe the selected check box **Create code generation report**.

   Select the **Launch report automatically** check box.

5. Select **IDE Link** in the tree, and observe the **Build Action** is **Create and Build Application Project**. You must have one of the Build options selected to get the code profile report (with RAM/ROM usage):

   - **Create and Build Application Project**
   - **Create, Build and Execute Application Project**
   - **Create, Build and Debug Application Project**

6. Close the Configuration Parameters dialog box.

7. Right-click the fuel rate controller block.

8. From the pop-up menu, select **Code Generation > Build Subsystem**.

9. On the following dialog box, click **Build**.

When code generation is complete, the Code Generation Report appears in your Help browser. Here you can review the RAM and ROM requirements of the model. To do this, left-click the link `Code profile report` in the left list. If you compared with the original floating-point version of the `fuelsys` control algorithm: you would find that using the fixed-point implementation results in a considerable reduction in both RAM and ROM. An example report is shown following.

Back Forward

Contents

Summary
Traceability Report
Subsystem Report
Code profile report

Generated Source Files

asc_serial_pec.c
c166_main.c
fuel.c
fuel_data.c
fuel.h
fuel_private.h
fuel_types.h
profile_vars.h

Code Profile Report

Compiler: Tasking

Compiler Options: Not Available

- Entire Code Summary
- Entire Code Detail
- Memory MAP

Entire Code Summary

Module	Size [in bytes]
RAM	2331
ROM	10420

Entire Code Detail

RAM	File	Section	Size [in bytes]
ASC_SERIAL_PEC_2_IR........		ASC_SERIAL_PEC_2_IR........	65
C166_US...................		C166_US...................	192
DISPATCH_BITDATA_SCT.......		DISPATCH_BITDATA_SCT.......	4
DISPATCH_RAM_DATA_SCT......		DISPATCH_RAM_DATA_SCT......	4
FUEL_6_NB.................		FUEL_6_NB.................	44
FUEL_ID_BA................		FUEL_ID_BA................	2
FUEL_ID_NB................		FUEL_ID_NB................	2
PROFILE_5_NB..............		PROFILE_5_NB..............	2012
PROFILE_ID_NB.............		PROFILE_ID_NB.............	6
ROM	**File**	**Section**	**Size [in bytes]**
?INTVECT..................		?INTVECT..................	512
ASC_SERIAL_PEC_1_PR........		ASC_SERIAL_PEC_1_PR........	368
BINARYSEARCH_S16_1_PR......		BINARYSEARCH_S16_1_PR......	74
C166_BSS..................		C166_BSS..................	20
C166_INIT.................		C166_INIT.................	38
C166_MAIN_1_PR............		C166_MAIN_1_PR............	356
C166_MAIN_2_CO............		C166_MAIN_2_CO............	8
DISPATCH_SCT..............		DISPATCH_SCT..............	52
DIV_S32_SAT_FLOOR_1_PR.....		DIV_S32_SAT_FLOOR_1_PR.....	262
DOTPRODUCT_S32S16_1_PR.....		DOTPRODUCT_S32S16_1_PR.....	60
FUEL_5_PR.................		FUEL_5_PR.................	3854
FUEL_DATA_1_NC............		FUEL_DATA_1_NC............	2810
FUEL_IR_BA................		FUEL_IR_BA................	2
FUEL_IR_NB................		FUEL_IR_NB................	2
INTERPOLATE_EVEN_S16_S16_SA		INTERPOLATE_EVEN_S16_S16_SA	192

Parameter Tuning and Signal Logging

- "Methods For Parameter Tuning and Signal Logging" on page 51-35
- "Using a Third Party Calibration Tool" on page 51-35

Methods For Parameter Tuning and Signal Logging

The coder product supports parameter tuning and signal logging either using Simulink external mode or with a third party calibration tool. In both cases the model must include a special block, the CAN Calibration Protocol block.

Using a Third Party Calibration Tool

The coder product allows an ASAP2 data definition file to be generated during the code generation process. This file can be used by a third party tool to access data from the real-time application while it is executing.

ASAP2 is a data definition standard by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description for data measurement, calibration, and diagnostic systems. The coder product lets you export an ASAP2 file containing information about your model during the code generation process. See also "Compatibility with Calibration Packages".

Before you begin generating ASAP2 filesSelect **Interface** (under **Code Generation)** in the tree. with the coder product, you should read the "Generating an ASAP2 File" section of the Simulink Coder documentation. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

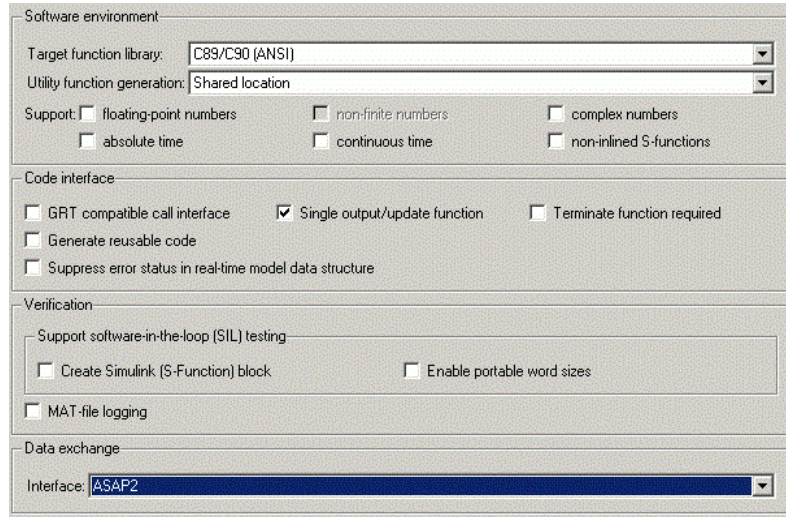
Select the ASAP2 option before the build process as follows:

1. Select **Simulation > Configuration Parameters**.

   The Configuration Parameters dialog box appears.

2. Select **Interface** (under **Code Generation)** in the tree.

3. Select the `ASAP2` option from the **Interface** drop-down menu, in the **Data exchange** frame, as shown.



4 Click **Apply**.

The build process creates an ASAM-compliant ASAP2 data definition file for the generated C code.

- The standard ASAP2 file generation does not include the memory address attributes in the generated file. Instead, it leaves a placeholder that must be replaced with the actual address by postprocessing the generated file.
- The map file options in the template project need to be set up a certain way for this procedure to work. If you have created your own template projects, and you do not have the correct settings, you see the following instructions:

```
Warning: It was not possible to do ASAP2 processing on your
.map file.This is because your EDE project template is not
configured to generate a .map file in the correct format.
To generate a .map file in the correct format you need to
setup the following options in your EDE project template:
Generate section map should be checked on
Generate register map should be checked off
Generate symbol table should be checked on
Format list file into pages should be checked off
Generate summary should be checked off
```

```
Page width should be equal to 132 characters
Symbol colums should be 1
You can change these options via Project -> Project Options
 -> Linker/Locator -> Map File -> Map File Format.
```

The coder product performs this postprocessing for you. To do this, it first extracts the memory address information from the map file generated during the link process. Secondly, it replaces the placeholders in the ASAP2 file with the actual memory addresses. This postprocessing is performed automatically and requires no additional input from you.

For an example of a model that is configured to generate an ASAP2 file, see `c166_ccp`.

Integrating Your Own Device Drivers

This section includes the following topics:

In this section...
“Integrating Manually Coded Device Drivers with a Simulink Model” on page 51-38
“Preparing Input and Output Signals to the Device Driver Functions” on page 51-39
“Calling the Device Driver Functions from c166_main.c” on page 51-41
“Adding the I/O Driver Source to the List of Files to Build” on page 51-41
“Tutorial: Using the Example Driver Functions” on page 51-43

Integrating Manually Coded Device Drivers with a Simulink Model

The coder product has a limited set of I/O device driver blocks. This means that, for most applications, it is necessary to manually write some device driver code.

This approach requires the following steps:

1. Identify the model inputs/outputs that must be read from/written to device driver functions.
2. Set the data type and storage class for each input or output signal so that it is compatible with your device driver code.
3. Use the hooks provided in the automatically generated `c166_main.c` to call your device driver initialization, input, and output functions.
4. Add your device driver source code to the list of files that must be included in the build process.

Each of these steps is described in the following sections. An example model is provided: `c166_user_io`.

An alternative approach is to create Simulink I/O blocks that automatically generate the device driver code. This approach may be worth considering if you need to reconfigure the I/O behavior frequently. If you want to take this alternative approach, you should consult the documentation on S-functions and TLC. See the section Developing Device Drivers for Embedded Targets in the document Developing Embedded Targets in the product help.

A useful tool for creating C166 device drivers is the freeware Digital Application Engineer DAvE from Infineon. You can find this at the following URL:

```
http://www.infineon.com/dave
```

Using this package along with the hardware User's Manual greatly eases the task of developing your own device driver code.

Preparing Input and Output Signals to the Device Driver Functions

Structure your model similarly to `c166_user_io`. Place the control algorithm that will be targeted onto the C166 microcontroller hardware in a separate subsystem. Before generating code, you can run this model in closed-loop simulation; this allows you to validate the correct behavior of your control algorithm before running it in real time.

When structuring your model in this way, you should make sure that all the input and output signals to the control algorithm are channeled through top-level input or output ports in the control algorithm subsystem.

By default, when you generate code for the control algorithm subsystem, the build process chooses variable names and data structures for each of the top-level input and output signals. However, in this case, you must ensure that the variables are global, and that their names and data structures match those that are required by the manually written device driver functions.

The example model `c166_user_io` illustrates some alternative ways to achieve this. The simplest method is to

1. Select one of the signals in your model connected to a top-level output port in the control algorithm subsystem. As an example, open the demo `c166_user_io`.

2. Open the `controller` subsystem.

3. Click the `output_PWM0` signal.

4. Select the menu item **Edit > Signal Properties**. The **Signal Properties** dialog box appears

5. Enter the required variable name for your signal in the **Signal name** edit box. This must match the variable name required by your manually written device driver functions.

6. Click the **Code Generation** tab and select `ExportedGlobal` from the **Storage class** drop-down menu.

When you generate code for this model, the build process uses the variable name that you have specified and creates an `extern` declaration in the model header file. By using a `#include` directive to include this model header file in your device driver source code, it is possible for the device driver functions to read or write this variable that is defined in the generated code.

A more sophisticated approach is to use custom storage classes. By using custom storage classes, you can collect a number of input or output variables together into a C struct, resulting in more readable code. The LED output signal in the `c166_user_io` uses a custom storage class, which uses a single bit in a bitfield variable. See "Tutorial: Using the Example Driver Functions" on page 51-43 for details about the different ways the model variables are defined and referenced to interface the manually coded driver functions and the automatically generated code.

By defining your own custom storage classes, you have complete control over the data structures that are used for any signal in the model. See the custom storage class documentation in the product help for more details.

Calling the Device Driver Functions from c166_main.c

You should check the option to include I/O driver function hooks. When you use code generation for this model, it includes some extra calls to user-supplied I/O device driver functions:

1. Select **Simulation > Configuration Parameters**.

   The Configuration Parameters dialog box appears.

2. Select **C166 Options (1)**, under **Code Generation** in the tree.

3. Select the check box option for including I/O driver function hooks.

These functions are

`user_io_initialize` — called following model initialization

`base_rate_model_inputs` — read model inputs, called at the base sample rate

`base_rate_model_outputs` — write model outputs, called at the base sample rate

`sub_rate_i_model_inputs` — read model inputs, called at the start of sub-rate i, where i=1, 2, ...

`sub_rate_i_model_outputs` — write model outputs, called at the start of sub-rate i, where i=1, 2, ...

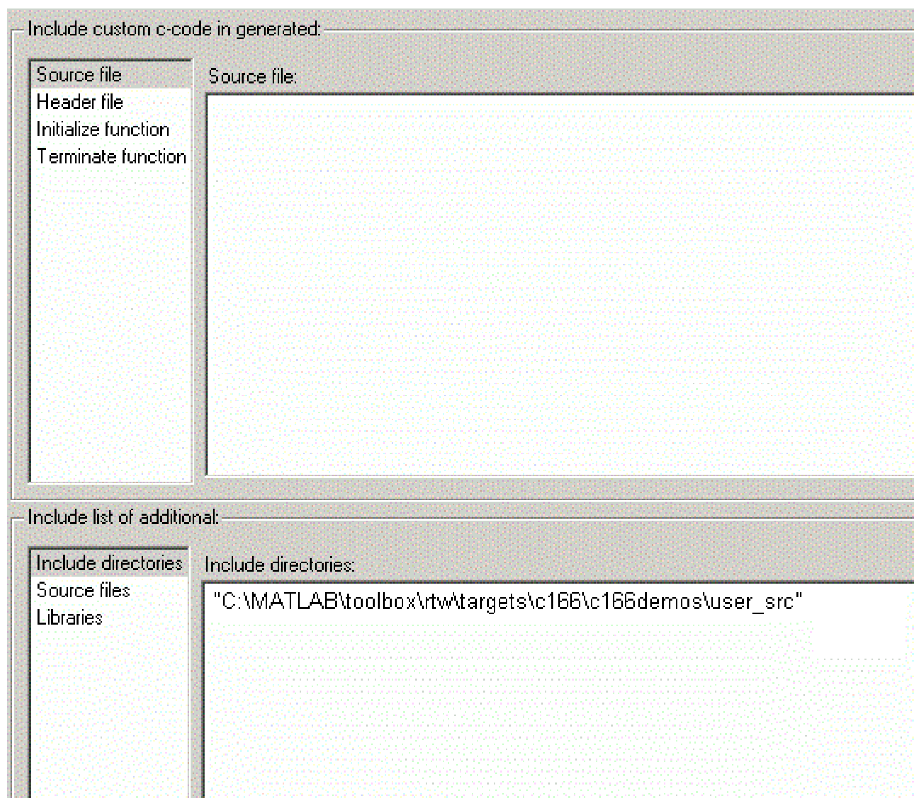
If you are using the automatically generated `c166_main.c`, then these function names are fixed.

For an example implementation of these functions, open the model `c166_user_io` and follow the link to open the I/O driver source files. These are described in “Tutorial: Using the Example Driver Functions” on page 51-43.

Adding the I/O Driver Source to the List of Files to Build

You must tell the build process to compile and link the I/O driver source files that you have written. You do so by adding the files to the custom code dialog box. Access the Configuration Parameters dialog box, look under **Code**

Generation > **Custom Code**, and add the necessary `Include Directories` and `Source Files`.



You are now ready to build your model and run it in real time.

You can examine an example of this in the example model `c166_user_io`. See the instructions in "Tutorial: Using the Example Driver Functions" on page 51-43. Step 8 shows you how to specify the location of your own manually coded drivers.

Tutorial: Using the Example Driver Functions

The example model c166_user_io demonstrates how to integrate user-defined device driver code. In this tutorial, you generate code from the controller subsystem, which automatically downloads and runs on the target.

The model c166_user_io illustrates three alternative methods for using global variables to interface the manually written driver functions with the automatically generated code. The three different methods are illustrated by these signals:

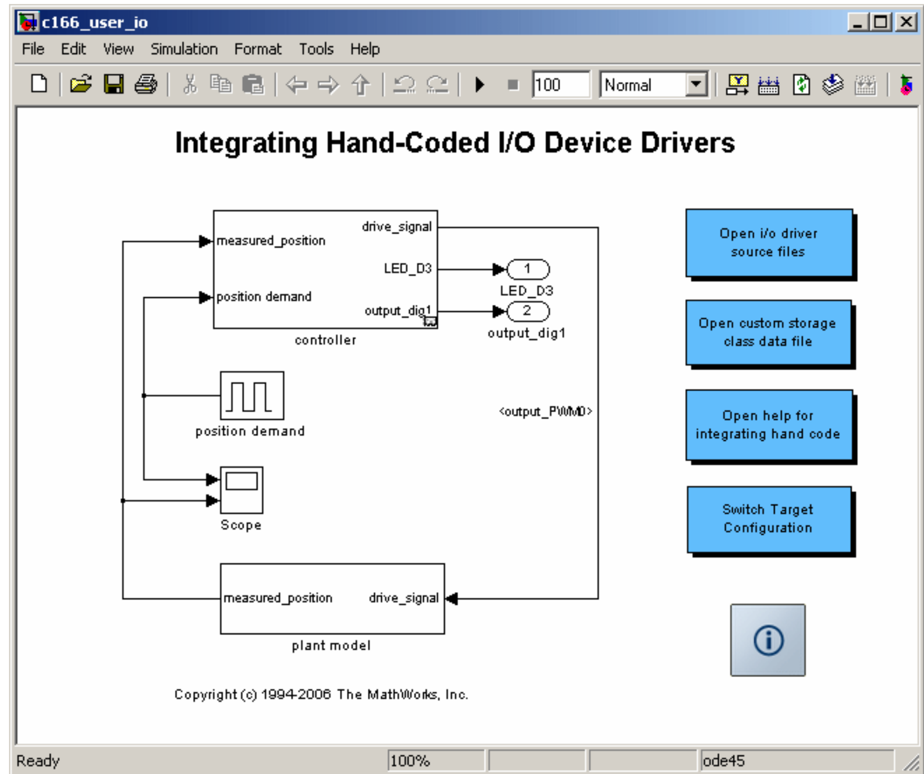
- input_adc0
- output_PWM0
- output_led_D3

For input_adc0, the variable is defined in the manually written code and referenced in the generated code.

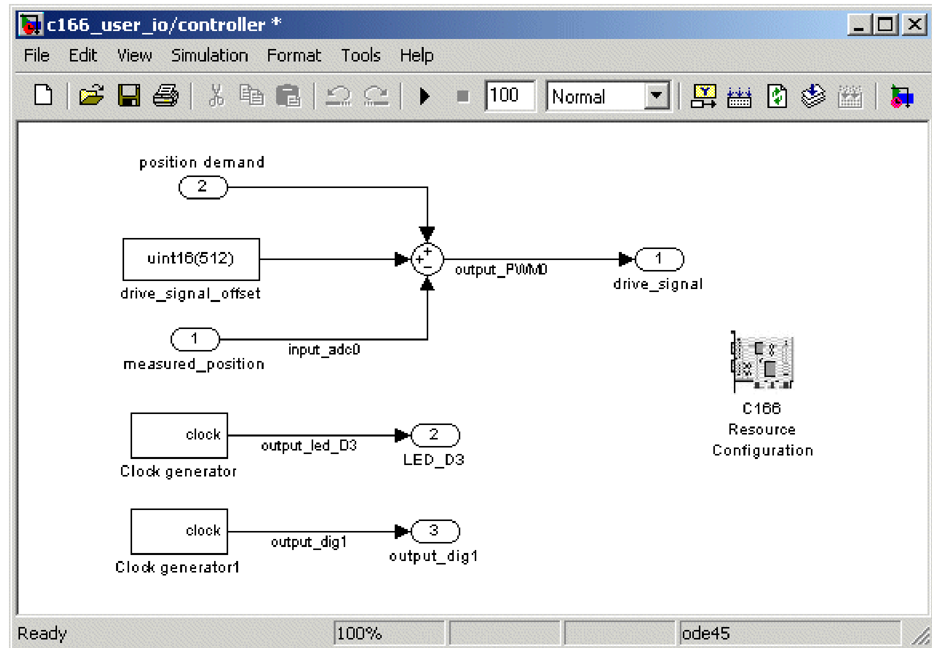
For output_PWM0, the variable is defined in the generated code and referenced in the manually written code.

For output_led_D3, a more sophisticated approach is used, involving custom storage classes. In this case, the variable is again defined in the generated code and referenced by the manually written code; the difference is that the variable is defined and referenced as a bitfield using C166 microcontroller bit-addressable memory:

1 Open the model c166_user_io.



2 Open the `controller` subsystem by double-clicking and select the signal `input_adc0`.



3 Select the menu item **Edit > Signal Properties**. The Signal Properties dialog box appears.

Click the **Code Generation** tab and observe that the **Storage class** is `ImportedExtern`. When you generate code for this model, the specified variable name `input_adc0` is used, and an `extern` declaration is created in the model header file. Since the storage class is `ImportedExtern`, this variable must be defined in the manually written driver code. When you open the file `user_io.c` in the next step, you will find the line `uint16_T input_adc0` that provides this definition.

4 In the top level model, double-click the link **Open the i/o driver source files**.

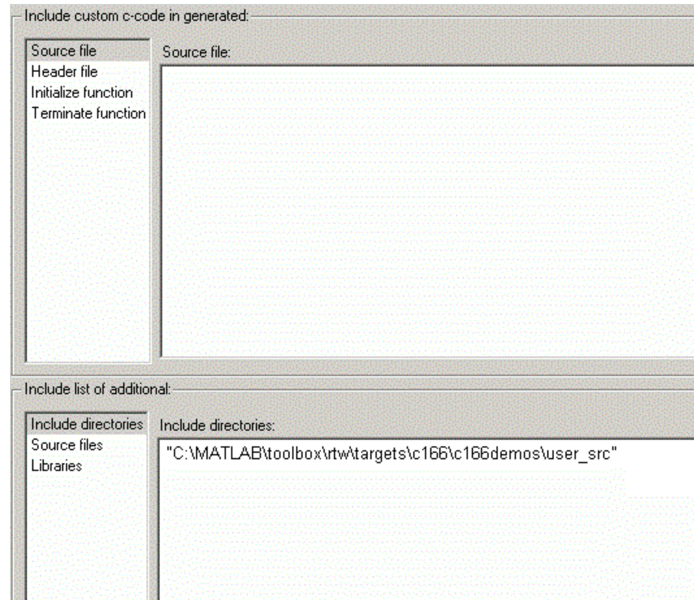
Two source files open in the MATLAB editor, `user_io.h` and `user_io.c`.

```
/*
 * File: user_io.h
 *
 * Abstract:
 *   Example file showing how to integrate hand-code input/output driver
 *   functions with Embedded Target for Infineon C166.
 *
 * $Revision: 1.1 $
 * $Date: 2002/10/03 09:45:27 $
 *
 */

#include "tmwtypes.h"

/*================================================*
 * Declare variables that are imported by the model
 *================================================*/
extern uint16_T input_adc0;

/*====================*
```

user_io.h user_io.c

5 Click the `user_io.h` tab, as shown above. Here you can see `extern uint16_T input_adc0` under the heading `Declare variables that are imported by the model`. Also look at the `#include` directive in `user_io.c`. The `extern` declaration and incorporating the header file into the build makes it possible for the device driver functions to read or write this variable that is defined in the generated code.

6 You need to instruct the coder product to compile and link the manually coded I/O driver source files in the build process. You do so by adding the files to the custom code dialog box. Access the Configuration Parameters dialog box, select **Code Generation** > **Custom Code** in the tree, and review the Include Directories and Source Files.



7 Select **C166 Options (1)** (under **Code Generation** in the tree). Notice that **Include input/output driver function hooks** is selected.

This instructs the build process to include extra calls to the user-supplied I/O device driver functions when code is generated for this model.

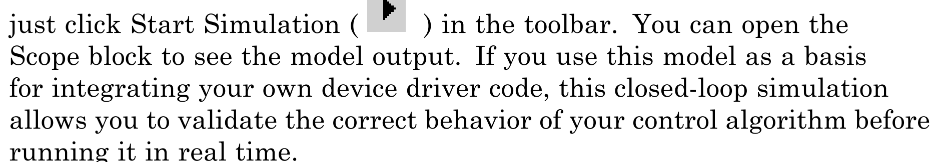
8 Select **Interface** in the tree. Observe the option **Floating-point numbers** is *not* selected.

If your model does not use floating point, you should make sure this option is not checked to use integer code only. Using only integer code results in smaller code size and faster real-time execution. It also speeds up the build process because libraries that are used only by floating-point applications are not included.

Explore the `user_io.c` file. This example file is intended to show you some manually coded input/output driver functions and how they can be integrated with the coder product.

You can see sections for initializing these input/output drivers: ADC, digital I/O, and Pulse Width Modulation (PWM).

9 Close the Signal Properties dialog box and Configuration Parameters dialog box if they are still open.

   Prior to generating code, you can run the model in closed-loop simulation; just click Start Simulation () in the toolbar. You can open the Scope block to see the model output. If you use this model as a basis for integrating your own device driver code, this closed-loop simulation allows you to validate the correct behavior of your control algorithm before running it in real time.

10 Generate code by right-clicking the `controller` subsystem and selecting **Code Generation > Build Subsystem**.

11 Click **Build** in the Build code for Subsystem: Controller dialog box that appears. Watch the messages as the process proceeds and code is generated.

   If you are using a Phytec phyCORE module with HD200 development board, the digital output is connected to the LED D3. You can see successful execution of the code when the LED blinks.

Custom Storage Class for C166 Microcontroller Bit-Addressable Memory

This section contains the following topics:

In this section...
"Specifying C166 Microcontroller Bit-Addressable Memory" on page 51-49
"Using the Bitfield Example Model" on page 51-50

Specifying C166 Microcontroller Bit-Addressable Memory

The coder product allows you to take advantage of Infineon C166 microcontroller bit-addressable memory. The example model `c166_bitfields` demonstrates this. By using bit-addressable memory, the compiler is able to use special assembler instructions that significantly reduce code size and increase execution speed.

Note This feature requires the coder product.

This is done by using the custom storage class `SimulinkC166.Signal`. To specify that a signal in the model should use bit-addressable memory, you must perform the following steps:

1. Ensure that the signal has the Simulink data type `'boolean'`.

2. Attach a label to the signal, either by using **Edit > Signal Properties** or by double-clicking the signal and typing in the name directly; this label will be used as the bitfield variable name in the generated code.

3. Create a new Simulink data object of type `SimulinkC166.Signal` with the same name as the signal label. See the file `c166bitfielddata.m` for an example.

4. Select **View > Model Explorer** and click the base workspace to inspect all the Simulink data objects that are available to the model.

5 Build the model.

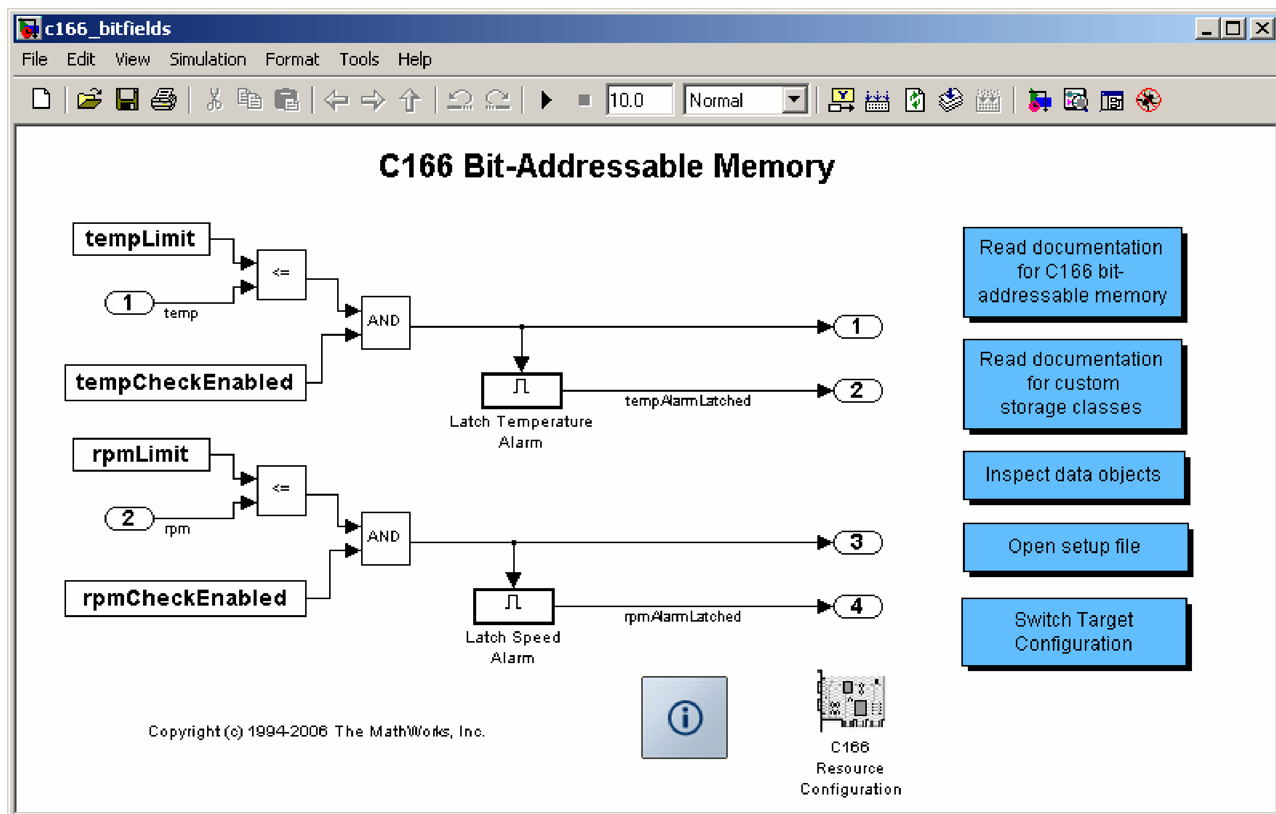
One of the signals in the demo model `c166_user_io` also uses the custom storage class `SimulinkC166.Signal` to specify that the signal uses bit-addressable memory. You can compare this with the `c166_bitfields` example; it is included in the steps in "Using the Bitfield Example Model" on page 51-50.

Using the Bitfield Example Model

You can use the example model `c166_bitfields` to see the automatic debugger start at the end of the build.

Follow these steps:

1 Open `c166_bitfields`.



2 Press **Ctrl+B** to build the model.

3 Examine the project generated code in the TASKING EDE:

a Select **Search > Multiple Sources**.

b In the dialog box, select Project Space under Multiple Sources, and enter `_bita` for the search string.

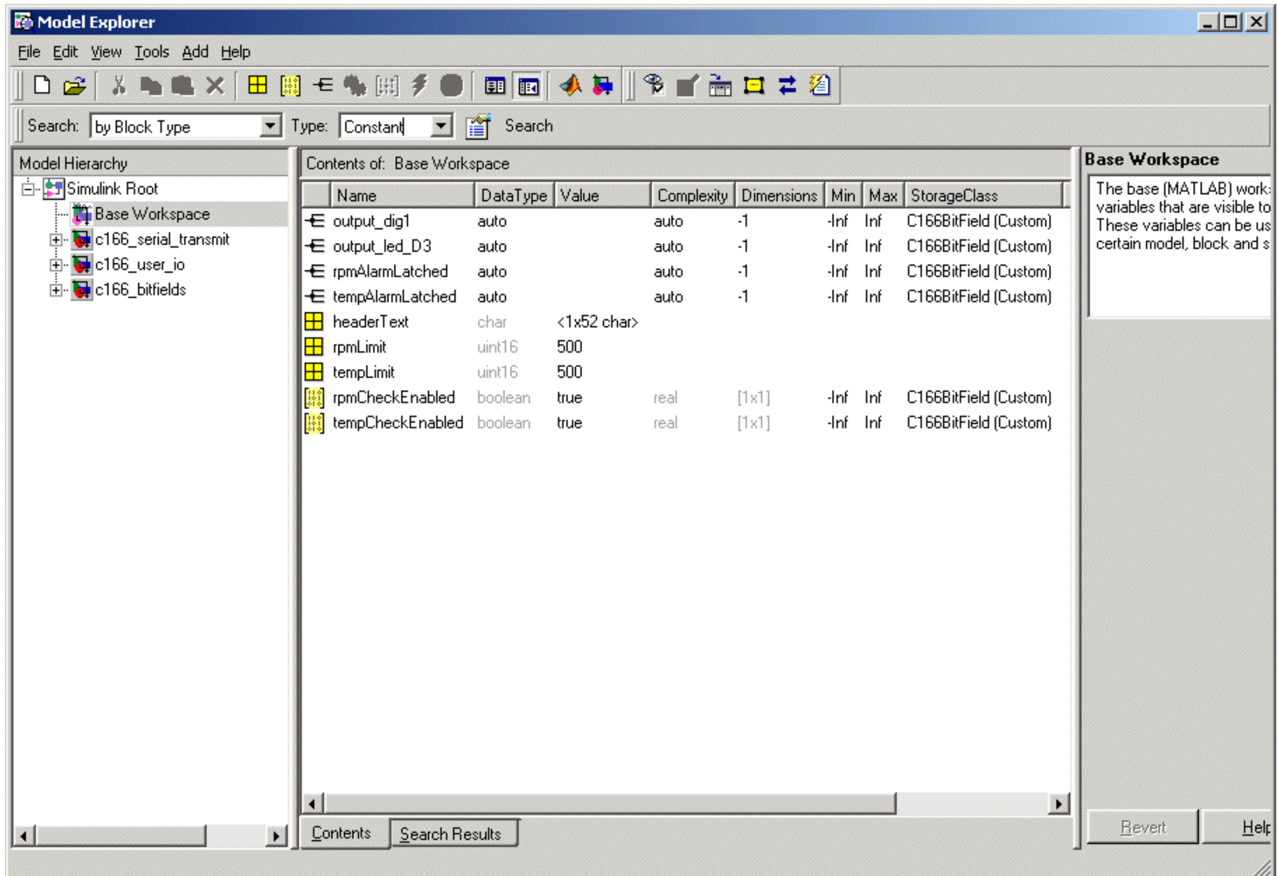
```
/* user code (bottom of header file) */
/* Bitfield data */
typedef _bita struct alarms_tag {
  unsigned int tempAlarmLatched:1;
  unsigned int rpmAlarmLatched:1;
} alarms_bitfield;

extern alarms_bitfield alarms;

/* Bitfield data */
typedef _bita struct control_tag {
  unsigned int rpmCheckEnabled:1;
  unsigned int tempCheckEnabled:1;
} control_bitfield;

extern control_bitfield control;

#endif                    /* _RTW_HEADER_c166_bitfields_h_ */
```

4 You can double-click **Open setup file** in the model to open the file `c166bitfielddata.m` in the MATLAB editor.

```
1   % C166BITFIELDDATA create data for C166 bitfield demo model
2
3   %   Copyright 2002 The MathWorks, Inc.
4   %   $Revision: 1.1 $
5   %   $Date: 2002/10/03 09:45:24 $
6
7   cscdemoclearws
8
9   tempAlarm = SimulinkC166.Signal;
10  tempAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
11
12  tempAlarmLatched = SimulinkC166.Signal;
13  tempAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
14
15  rpmAlarm = SimulinkC166.Signal;
16  rpmAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
17
18  rpmAlarmLatched = SimulinkC166.Signal;
19  rpmAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
20
21  tempLimit = uint16(500);
```

This file creates a new Simulink data object using the custom storage class `SimulinkC166.Signal`. By using custom storage classes, you can collect a number of input or output variables together into a C struct, resulting in more readable code. By defining your own custom storage classes, you have complete control over the data structures that are used for any signal in the model. See the custom storage class topic in the *Embedded Coder User's Guide* for more details. You can double-click **Read documentation for custom storage classes** in the model to go directly to the relevant product help section.

5. You can double-click **Inspect data objects** to inspect all the Simulink data objects that are available to the model.

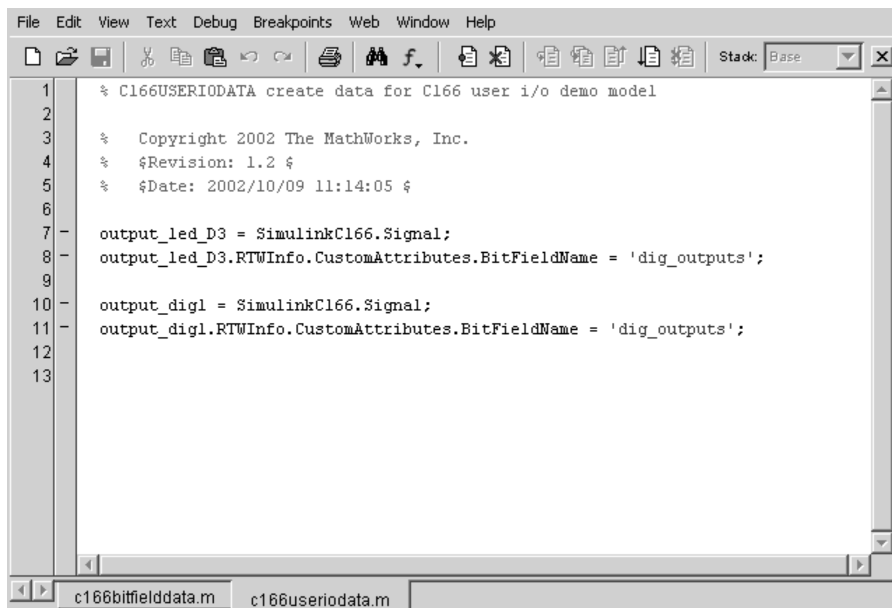


Here you can see the `SimulinkC166.Signal` data object and you can click on each object to inspect the properties.

6 One of the signals in the demo model `c166_user_io` also uses the custom storage class `SimulinkC166.Signal` to specify that the signal uses bit-addressable memory. Open `c166_user_io`.

7 Double-click **Open custom storage class data file**.

The file `c166useriodata.m` opens in the MATLAB editor.



Compare with `c166bitfielddata.m`.

For more details on the variables in this model, see “Tutorial: Using the Example Driver Functions” on page 51-43.

Execution Profiling

This section contains the following topics:

In this section...
"Overview of Execution Profiling" on page 51-56
"Options for Execution Profiling" on page 51-60
"Multitasking Demo Model" on page 51-62

Overview of Execution Profiling

- "Introducing Execution Profiling" on page 51-56
- "The Profiling Command" on page 51-57
- "Definitions" on page 51-59
- "Execution Profiling Blocks" on page 51-59

Introducing Execution Profiling

The coder product provides a set of utilities for recording, uploading, and analyzing execution profile data for timer-based tasks and asynchronous Interrupt Service Routines (ISRs). With these utilities, you can

- Generate a graphical display that shows when timer-based tasks and interrupt service routines are activated, preempted, resumed, and completed.
- Generate a report with information on
  - Maximum number of overruns for each timer-based task since model execution began
  - Maximum turnaround time for each timer-based task since model execution began
  - Analysis of profiling data for timer-based tasks and asynchronous interrupts over a period of time

To perform execution-profiling analysis on a model, you must perform the following steps:

1. Place a copy of the appropriate execution profiling block in your model:
   - Execution Profiling via ASC0 if using a serial connection
   - Execution Profiling via CAN A if using CAN with a C166 processor
   - Execution Profiling via TwinCAN A if using CAN with an XC16x processor variant
2. Select the **Execution profiling** option under **Code Generation** options in the Configuration Parameters dialog box. See "Options for Execution Profiling" on page 51-60.
3. Connect the target processor to your host PC (with a serial or CAN cable).
4. Build, download, and run the model.
5. Initiate execution profiling by running the command `profile_c166`. See below for more information on the profiling command.

Two forms of execution profiling are provided:

1. The worst-case values for task turnaround times and number of concurrent task overruns since model execution began are updated whenever a previous worst-case value is exceeded.
2. A snapshot of task and ISR activity may be recorded over a period of time; the length of this period depends on how much memory is reserved to log the data.

The Profiling Command

Use the profiling command as follows:

```
profile_c166(connection)
```

Specify your `connection` as `'can'` or `'serial'`, to collect data via a CAN or serial connection between the target and the host computer. Make sure the model includes the appropriate C166 execution profiling block (CAN or

ASC0), to provide an interface between the target-side profiling engine and the host-side computer from which this command is run.

`PROFDATA = profile_c166(connection)` collects and displays execution profiling data from a C166 target microcontroller that is running a suitably configured application generated by the coder product. `PROFDATA` contains the execution profiling data in the format documented by `exprofile_unpack`.

The data collected is unpacked then displayed in a summary HTML report and as a MATLAB graphic.

To use the serial connection, the C166 board must be connected via a serial cable to one of the host computer's serial ports. This function defaults to port ASC0 on the C166 and port COM1 on the host computer. If the `'BitRate'` argument is not provided, the default of 57600 baud is used.

```
PROFDATA = PROFILE_C166('serial','SerialPort',serialport)
```

sets the serial port to the specified `serialport`, which should be one of COM1, COM2, etc.

Optionally, you can specify the bit rate as follows:

```
PROFDATA = PROFILE_C166('serial', 'BitRate', bitrate)
```

This specification sets the `bitrate` for serial connection to the target. `bitrate` must be the same as the bit rate specified for the application that is running on the target.

Alternatively, you can set the bitrate for the serial connection to the target automatically as follows:

```
profdata = profile_c166('serial', 'ModelName', modelname)
```

This specification automatically sets the bit rate by analyzing *`modelname`* and extracting the correct serial connection bit rate setting from the model. *`modelname`* should be set to the name of a model which is currently open and running on the target.

To use the CAN connection, you must have suitable CAN hardware installed. If no Application Channel is specified, this function will use the channel

`'MATLAB 1'`. The bit rate is a property of the Application Channel; to change the bit rate, you must use a different Application Channel, or change the bit rate by running the Vector Informatik configuration utility. To run this utility, make sure that `vcanconf.exe` is on your System Path, then type `vcanconf` from a Windows command prompt.

You can specify the Application Channel as follows:

```
profdata = profile_c166('can', 'CANChannel', canchannel)
```

`canchannel` specifies the Vector Informatik CAN Application Channel, and must be of the form `'MATLAB 1'`, `'MATLAB 2'` etc.

Definitions

Task turnaround time is the elapsed time between start and finish of a task. If the task is not preempted, then the task turnaround time is equal to the task execution time.

Task execution time is that part of the time between task start and finish when the task is actually running and not preempted by another task. Note that the task execution time cannot be measured directly, but is inferred from the task start and finish time and the intervening periods during which it was preempted by another task. Note that, in performing these calculations, no account is taken of processor time consumed by the scheduler while switching tasks: this means that, in cases where preemption has occurred, the reported task execution times will overestimate the true values.

Concurrent task overruns occur when a timer task does not complete before that same task is next scheduled to run. Depending on how the real-time scheduler is configured, a task overrun may be handled as a real-time failure. Alternatively, a small number of concurrent task overruns may be allowed to accommodate cases where a task occasionally takes longer than normal to complete.

Execution Profiling Blocks

See the block reference sections:

- C166 Execution Profiling via ASC0

- C166 Execution Profiling via CAN A
- C166 Execution Profiling via TwinCAN A

Options for Execution Profiling

- "Execution Profiling" on page 51-60
- "Number of Data Points" on page 51-60
- "Task Scheduler Overrun Options" on page 51-61

Execution Profiling

You can see the options for execution profiling by selecting **C166 Options (1)** (under **Code Generation** in the tree) in the Configuration Parameters dialog box.

If the **Execution Profiling** option is selected, then the generated code for the model will be "instrumented" with function calls at the beginning and end of each task or ISR to be profiled. These function calls read a timer (on C166 a free running timer is selected from the options in the C166 Resource Configuration block) and log this reading along with a task identifier.

When code for the model is generated, these functions will update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous worst-case value is exceeded. Additionally, when a trigger is provided, data will be logged over a period of time to record all task start and finish times. The trigger signal can be supplied, for example, by the block C166 Execution Profiling via CAN A.

Number of Data Points

When a snapshot of task and ISR activity is logged, this data is stored in memory that is statically allocated at build time. Each data point requires 4 bytes on C166. The larger the number of data points to be stored, the more RAM that must be reserved for this purpose. At the end of a logging run, the data must be uploaded to the host computer for analysis; this is typically achieved by using one of the C166 execution profiling blocks — via ASCO, CAN A, or TwinCAN A. For more information, see:

- C166 Execution Profiling via ASC0
- C166 Execution Profiling via CAN A
- C166 Execution Profiling via TwinCAN A

Task Scheduler Overrun Options

These scheduler options configure the allowable number of concurrent task overruns. You can see these options on the **C166 Options (1)** section in the Configuration Parameters dialog box.

You can use the options **Maximum number of concurrent base-rate overruns** and **Maximum number of concurrent sub-rate overruns** to configure the behavior of the scheduler when any of the timer based tasks do not complete within their allowed sample time. It is useful to allow task overruns in the case where a task may occasionally take longer than usual to complete (e.g., if extra processing is required when a special event occurs); if the task overrun is only occasional, then it is possible for the scheduler to catch up after the extra processing has been completed.

If the maximum number of concurrent overruns for any task is exceeded, this is deemed to be a failure and the real-time application is stopped.

As an example, if the base rate is 1 ms and the maximum number of concurrent base-rate overruns is set to 5 then it is possible for the base rate task to run for almost 6 ms before failure occurs. Once the overrun has occurred, it is necessary for subsequent executions of the base rate to complete in less than 1 ms in order that the lost time is recovered.

The occurrence of base-rate overruns does not affect the numerical behavior of the algorithm (although reading/writing external devices will of course be delayed).

If sub-rate overruns are allowed, then the transfer of data between different rates (via rate-transition blocks) in the model may be affected; this causes the numerical behavior in real time to differ from the behavior in simulation. To see an illustration of this effect, try running the demo model `c166_multitasking`, described in the next section. To disallow sub-rate overruns and ensure that this effect does not occur, you should set **Maximum number of concurrent sub-rate overruns** to zero.

Note Allowing sub-rate overruns may cause non-determinism and loss of integrity for data transferred between different rates in the model. Set this value to zero if you require sub-rate overruns to be handled as a failure (recommended).

If you allow sub-rate overruns, then the behavior of any Rate-Transition blocks may be affected. Specifically, if the model contains a Rate Transition block where the option "Ensure deterministic data transfer (maximum delay)" is selected, then this setting may not be honored.

Multitasking Demo Model

- "Introducing the Multitasking Demo" on page 51-62
- "Running the Multitasking Demo" on page 51-63
- "Interpreting the MATLAB Graphic" on page 51-65
- "The Generated HTML Report" on page 51-67

Introducing the Multitasking Demo

The demo model `c166_multitasking` illustrates both execution profiling and the preemptive multitasking scheduler with configurable overrun handling.

The model is multirate, having tasks running at 1 ms, 4 ms, and 16 ms. It is configured to use the preemptive multitasking scheduler.

A special feature of this model is that each task is designed to perform an increasing number of calculations to increase the processor loading until that task reaches a target turnaround time. This behavior ensures that task overruns occur to demonstrate the behavior of the model in this situation.

Each block in the model, labeled `Load base rate`, `Load sub-rate 1`, `Load sub-rate 2` performs calculations, the result of which should always be 1 both in simulation and in real time. Any other result is a failure and should never occur.

The `Test Rate Interaction` blocks are designed to test whether data is transferred between tasks in a deterministic manner. In simulation, the output of each of these blocks is always zero, indicating that there is no drift between tasks running at different rates. When running in real time, under normal circumstances, the output is also zero; in this case the real-time behavior is deterministic and exactly matches the results in simulation. Even if task preemption and base-rate overruns occur, the output of these blocks will be zero so that the real-time behavior faithfully reproduces the results in simulation. The circumstance under which drift occurs is if sub-rate overruns occur during execution in real time; if this behavior is not desired, you should disallow sub-rate overruns by setting the maximum allowed number of sub-rate overruns to zero in the **C166 Options (1)** section in the Configuration Parameters dialog box (see "Task Scheduler Overrun Options" on page 51-61).

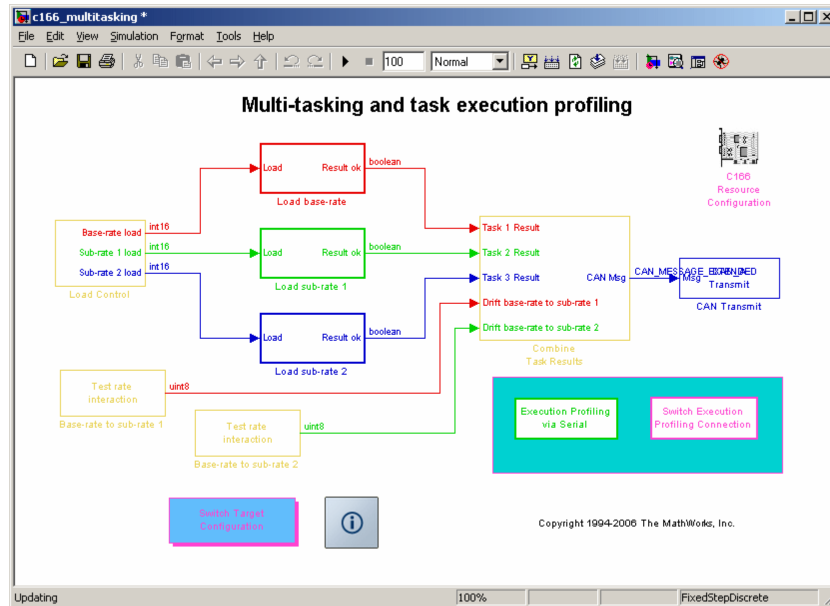
You can double-click the block provided in the model to switch between profiling over serial or CAN connections.

Running the Multitasking Demo

1. Open the model by typing at the command line

   ```
   c166_multitasking
   ```

   If viewing in the Help browser, you can click the link to open the model. If you update the diagram you can see the sample-time colors.



2 Select **Simulation > Configuration Parameters**.

The Configuration Parameters dialog box appears.

3 Select **IDE Link** in the tree and change the **Build action** to `Create, Build and Execute Application Project`. Click **OK** to dismiss the dialog box.

4 Make sure the target is connected to the host PC via serial or CAN cable. The default setting in this demo model is serial. You can double-click the Switch Execution Profiling Connection block to toggle between blocks for serial and CAN. See below for instructions if using CAN.

5 To build and run the model, select the model window, and then press **Ctrl+B**.

Watch the messages in the command window as code is generated and loaded into the TASKING EDE, then the CrossView Pro Debugger starts, connects to the target, and downloads the code.

6 In the CrossView window, click **Run** in the toolbar to start the application running on the target.

7 At the command line, type

```
profile_c166 ('serial')
```

You will see messages in the command window as `profile_c166` runs.

When the data has been obtained the Help browser and a figure window appear, displaying the HTML report and the task execution profile.

8 Scroll to view the HTML report on task timings and use the controls to zoom in on the MATLAB graphic to examine the details of the task overruns.

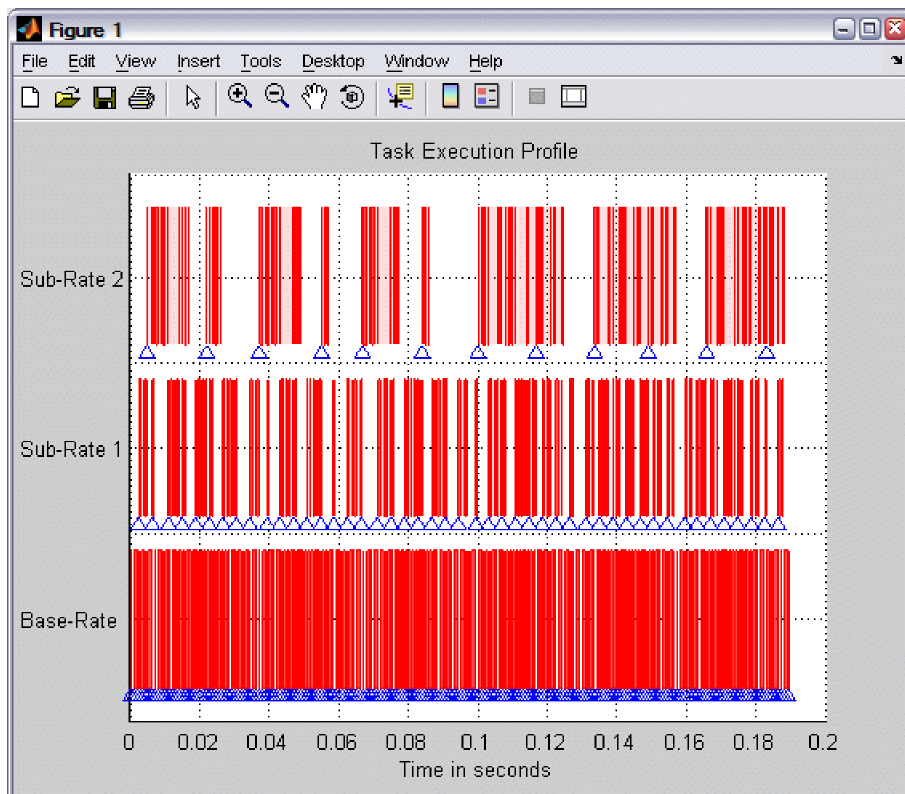
If using CAN, be sure to use CAN channel 0 (not 1) on the PC. You can double-click the Switch Execution Profiling Connection block in the model to switch to CAN, and follow the same instructions as for a serial connection, except step 7 when the application is running. At the command line, type

```
profile_c166 ('CAN')
```

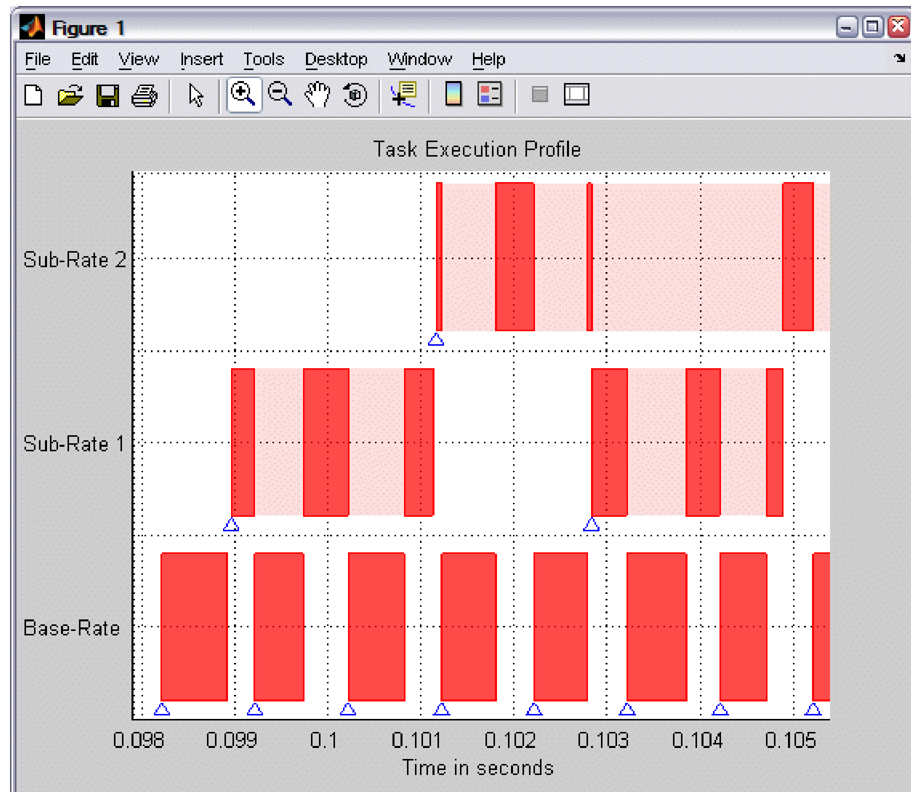
You will see command line messages as the function tests the CAN channel, and requests and collects profiling data. When using CAN, it can be useful to run a monitor program such as `btest32` to verify that the model is running — for example you will see messages appearing on the CAN bus and you can see that you have connected the correct CAN channel.

Interpreting the MATLAB Graphic

Dark shaded areas show the region where a task is executing. Light shaded areas show the region where a task is preempted by a higher priority task or ISR. Triangles indicate the beginning of a task. An example is shown following.



Zoom in to see the details of times that tasks are executing and being preempted, as shown in the following example.



.

The Generated HTML Report

See “Definitions” on page 51-59 for the terms task turnaround time, task execution time, and concurrent task overruns.

All times are in seconds. The timer resolution is 4e-007 seconds and the measurement range is 0.026214 seconds.

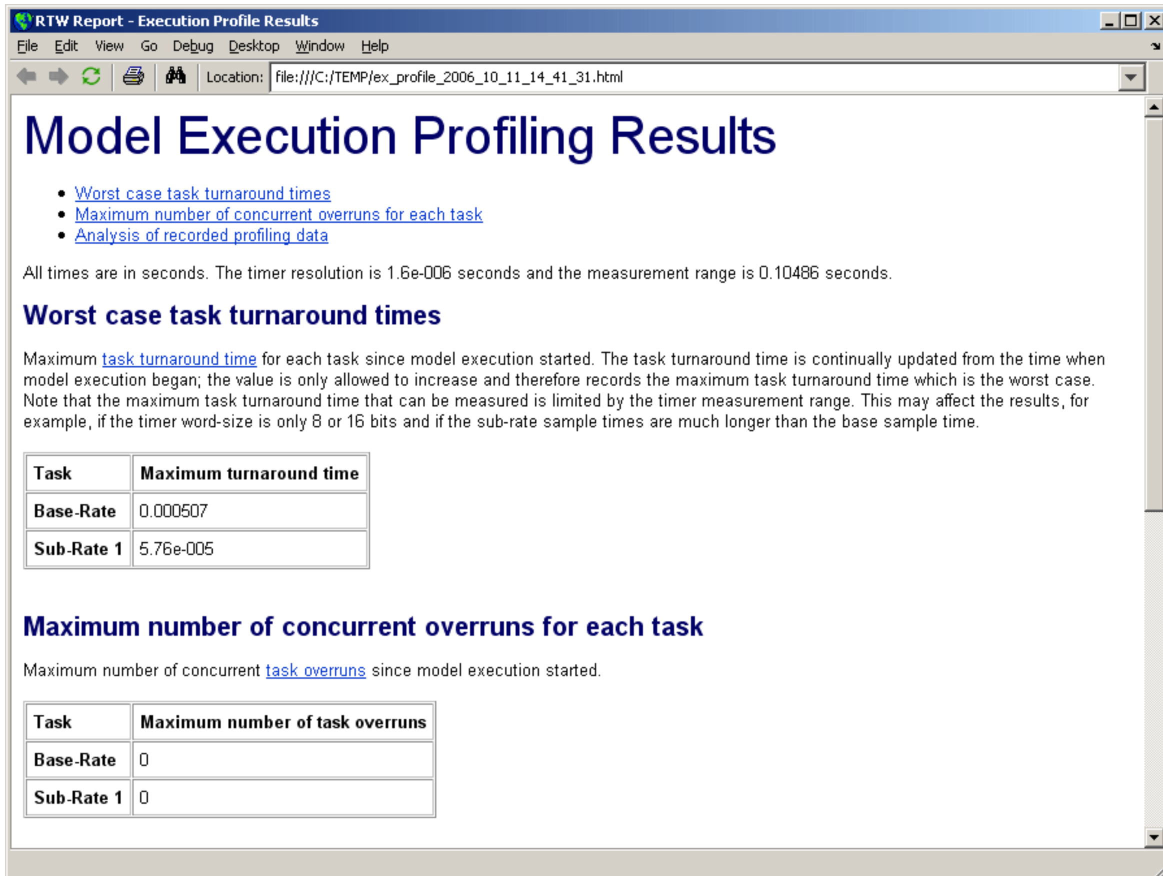
The report contains the following information:

- Worst-case task turnaround times

- Maximum task turnaround time for each task since model execution started. Note that the maximum task turnaround time that can be measured is limited by the timer measurement range.

- Maximum number of overruns for each task
  - Maximum number of concurrent task overruns since model execution started
- Analysis of recorded profiling data
  - Analysis of task turnaround times and task execution times based on recorded data over a period of 0.18139 second

Examples are shown following.


RTW Report - Execution Profile Results

File Edit View Go Debug Desktop Window Help

Location: file:///C:/TEMP/ex_profile_2006_10_11_14_41_31.html

Model Execution Profiling Results

- Worst case task turnaround times
- Maximum number of concurrent overruns for each task
- Analysis of recorded profiling data

All times are in seconds. The timer resolution is 1.6e-006 seconds and the measurement range is 0.10486 seconds.

Worst case task turnaround times

Maximum task turnaround time for each task since model execution started. The task turnaround time is continually updated from the time when model execution began; the value is only allowed to increase and therefore records the maximum task turnaround time which is the worst case. Note that the maximum task turnaround time that can be measured is limited by the timer measurement range. This may affect the results, for example, if the timer word-size is only 8 or 16 bits and if the sub-rate sample times are much longer than the base sample time.

Task	Maximum turnaround time
Base-Rate	0.000507
Sub-Rate 1	5.76e-005

Maximum number of concurrent overruns for each task

Maximum number of concurrent task overruns since model execution started.

Task	Maximum number of task overruns
Base-Rate	0
Sub-Rate 1	0

RTW Report - Execution Profile Results

File Edit View Go Debug Desktop Window Help

Location: file:///C:/TEMP/ex_profile_2006_10_11_14_41_31.html

Analysis of profiling data recorded over 1.6695 seconds.

Profiling data was recorded over 1.6695 seconds. The recorded data for task turnaround times and task execution times is presented in the table below.

Task	Maximum turnaround time	Average turnaround time	Maximum execution time	Average execution time	Average sample time
Base-Rate	0.000462 at 1.03	0.000462	0.000462 at 1.03	0.000462	0.01
Sub-Rate 1	2.24e-005 at 1.14	2.24e-005	2.24e-005 at 1.14	2.24e-005	0.02

Task turnaround time is the elapsed time between start and finish of the task. If the task is not pre-empted then the task turnaround time is equal to the task execution time.

Task execution time is that part of the time between task start and finish when the task is actually running and not pre-empted by another task. Note that the task execution time cannot be measured directly, but is inferred from the task start and finish time and the intervening periods during which it was preempted by another task. Note that, in performing these calculations, no account is taken of processor time consumed by the scheduler while switching tasks: this means that, in cases where preemption has occurred, the reported task execution times will overestimate the true values.

Task overruns occur when a timer task does not complete before that same task is next scheduled to run. Depending on how the real-time scheduler is configured, a task overrun may be handled as a real-time failure. Alternatively, a small number of task overruns may be allowed in order to accommodate cases where a task occasionally takes longer than normal to complete. If a task overrun has occurred and the same task is again scheduled to run *before* the first overrun has been cleared then two *concurrent* task overruns are said to have occurred.

Configuration Parameters

Code Generation Pane: C166 Options

- “C166 Options Tab Overview” on page 51-72
- “Include input/output driver function hooks” on page 51-73
- “Maximum number of concurrent base-rate overruns:” on page 51-74
- “Maximum number of concurrent sub-rate overruns:” on page 51-75
- “Execution profiling” on page 51-77
- “Number of data points:” on page 51-78

C166 Options Tab Overview

Parameters for integrating your own device driver code and controlling execution profiling with the coder product.

Configuration. This pane appears only if you specify the `C166.tlc` or `C166_grt.tlc` system target file.

See Also.

- Overview of C166 Configuration Parameters
- Getting Started

Include input/output driver function hooks

Specify whether to integrate your own device driver code.

Settings. **Default:** Off

On

Include input/output driver function hooks. When you generate code for this model, it includes some extra calls to user-supplied input/output device driver functions, to read and write model inputs and outputs. See Calling the Device Driver Functions from c166_main.c for function names and instructions.

Off

Do not include input/output driver function hooks.

Command-Line Information.

Parameter: `InputOutputDriverHooks`
Type: logical
Value: `0 | 1`
Default: `0`

See Also. Integrating Your Own Device Drivers

Maximum number of concurrent base-rate overruns:

Configure allowable base-rate overruns.

Settings. Default: 5

Minimum: 0

Maximum: No maximum value — it depends on available memory.

Tips.

- Use this option to configure the behavior of the scheduler when timer based tasks do not complete within their allowed sample time.
- It is useful to allow task overruns in the case where a task may occasionally take longer than usual to complete (e.g. if extra processing is required when a special event occurs); if the task overrun is only occasional then it is possible for the scheduler to 'catch up' after the extra processing has been completed.
- If the maximum number of concurrent overruns for any task is exceeded, this is deemed to be a failure and the real-time application is stopped. This in turn will result in a watchdog timer timeout and the processor will be reset.
- The occurrence of base-rate overruns does not affect the numerical behavior of the algorithm (although reading/writing external devices will of course be delayed).

Command-Line Information.

Parameter: `BaseRateMaxOverrunsValue`
Type: int
Value: `0 | 1 | 2...`
Default: `5`

See Also.

- Task Scheduler Overrun Options
- Execution Profiling

Maximum number of concurrent sub-rate overruns:

Configure allowable sub-rate overruns.

Settings. **Default:** 0

Minimum: 0

Maximum: No maximum value — it depends on available memory.

Tips.

Note Allowing sub-rate overruns may cause non-determinism and loss of integrity for data transferred between different rates in the model. Set this value to zero if you require sub-rate overruns to be handled as a failure (recommended).

- If this option is set to a value greater than zero, then the behavior of any Rate-Transition blocks may be affected. Specifically, if the model contains a Rate Transition block where the option "Ensure deterministic data transfer (maximum delay)" is selected, then this setting may not be honored.
- If sub-rate overruns are allowed then the transfer of data between different rates (via rate-transition blocks) in the model may be affected; this causes the numerical behavior in real-time to differ from the behavior in simulation. To see an illustration of this effect try running the demo model `c166_multitasking`. To disallow sub-rate overruns and ensure that this effect does not occur, you should set **Maximum number of concurrent sub-rate overruns** to zero.

Command-Line Information.

Parameter: `SubRateMaxOverrunsValue`
Type: int
Value: `0 | 1 | 2...`
Default: `0`

See Also.

- Task Scheduler Overrun Options
- Execution Profiling

Execution profiling

Specify whether to configure code for execution profiling.

Settings. Default: Off

On

Include function calls in the generated code for the model at the beginning and end of each task or asynchronous Interrupt Service Routine (ISR) to be profiled. When you perform an execution profiling run, these function calls read a timer and log this reading, along with a task identifier, for uploading and analyzing.

Off

Do not add function calls for execution profiling.

Tip. When code for the model is generated, these function calls update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous worst case value is exceeded. Additionally, when a trigger is provided, data can be logged over a period of time to record all task start and task finish times. The trigger signal can be supplied by the execution profiling blocks.

See Also.

- "Options for Execution Profiling" on page 51-60
- Execution Profiling

Number of data points:

Specify number of data points to log for execution profiling runs.

Settings. **Default:** 500

Minimum: This depends on the number of tasks. Three is a sensible minimum to get useful information back.

Maximum: No maximum value - it depends on available memory.

Tip. When a snapshot of task and ISR activity is logged, this data is stored in memory that is statically allocated at build time. Each data point requires 4 bytes on C166 microcontrollers. The larger the number of data points to be stored, the more RAM that must be reserved for this purpose. At the end of a logging run, the data must be uploaded to the host computer for analysis; this is typically achieved by using one of the C166 execution profiling blocks.

Command-Line Information.

Parameter: `ExecutionProfilingNumSamples`
Type: int
Value: `3 | 4 | 5...`
Default: `500`

See Also.

- Number of Data Points
- Execution Profiling

Working with Linux Target

- "Disambiguation" on page 52-2
- "Preparing Models to Run on Linux" on page 52-3
- "Scheduler" on page 52-4
- "Example: Build Generated Code on a BeagleBoard Running Linux" on page 52-7
- "Example: Build Generated Code on a Linux Host, Then Run It Remotely on BeagleBoard" on page 52-9
- "Embedded Linux Topics" on page 52-14

Disambiguation

This documentation uses the term “Linux” generically to refer to:

- Linux running on a host computer
- Linux running on an target processor

If the distinction between host and target is important, the documentation will identify the hardware platform on which Linux is running. For example:

- “Embedded Linux” or “Linux running on a target processor”
- “Linux running on a host computer.”

Preparing Models to Run on Linux

To build an executable that runs on Linux, perform the following steps:

1. Install and configure Eclipse IDE according to the instructions in Chapter 48, “Working with Eclipse IDE”.

2. Locate the Target Preferences block in the Simulink Library Browser, under Embedded Coder > Embedded Targets.

3. Copy the Target Preferences block to your model.

4. In the **Initialize Configuration Parameters** dialog box, set the IDE to Eclipse, and select the processor for which you are generating code.

5. Set **Operating System** to `Linux`. This action creates a **Linux** tab for setting the **Scheduling Mode** and **Base Rate Priority**.

6. Set the **Scheduling Mode** to one of these options:

   - If you select `real-time`, the model uses a timer to trigger the base rate at regular periods.
   - If you select `free-running`, the model does not use a timer. Instead, the model completes each process or thread before running the next one.

7. For Linux, you can set the **Base Rate Priority** relative to other processes and threads. You can enter values from (the number of rates + 1) to 99.

8. In IDE Link, configure the model to build and execute:

   a. In the model, select **Simulation** > **Configuration Parameters**.

   b. Select the **Code Generation** > **IDE Link** pane.

   c. Set **Build action** to `Build and execute`.

9. Build the model. Select **Tools** > **Code Generation** > **Build Model**.

   After the build completes, Embedded Coder software downloads the executable to the remote system and runs it.

Scheduler

In this section...

"Base Rate" on page 52-4

"Running Target Applications on Multicore Processors" on page 57-10

"Avoiding Lock-Up in Free-Running, Multirate, Multitasking Models" on page 52-6

"Limitations" on page 52-6

Base Rate

The base rate in the model maps to a thread and runs as fast as possible. The base rate priority selection in the OS tab allows you to set a static priority for the base rate task. By default, this rate is 40.

The process running single-tasking models has Default scheduling policy when model is single-tasking or there is a single rate in the model. Static priority of the process is 0 in this case.

Running Target Applications on Multicore Processors

If you are generating code for a processor running Linux or VxWorks, you can elect to partition the code such that each rate is placed in its own thread. With code generated from a multi-rate model, the multi-threaded application will be enabled for concurrent multicore execution, as scheduled by the target operating system.

1. Create a multi-rate Simulink model.

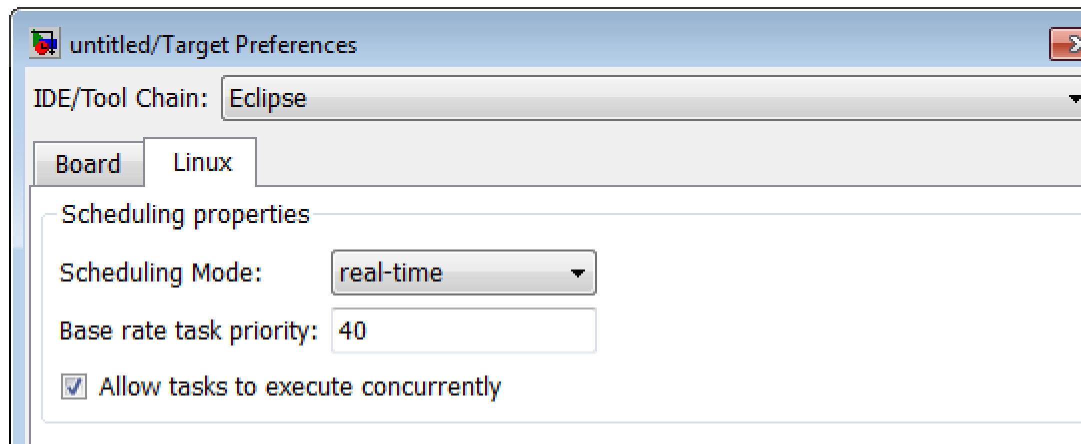
2. Add a Target Preferences block to your model as described in the "Target Preferences" on page 43-4 section.

3. Verify that your model uses a Rate Transition block to transition between rates.

4. Clear the **Ensure deterministic data transfer** checkbox of the Rate Transition block. This action forces the Rate Transition block to use the most recent data available.

5 In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

6 In the Linux or VxWorks tab of the Target Preferences block, select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables the generated multi-threaded code to run concurrently on multicore processors.



7 In the Configuration Parameters dialog box, on the Solver pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

8 Open the Rate Transition block and clear the **Ensure deterministic data transfer** checkbox. This action forces the Rate Transition block to use the most recent data.

9 In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

10 Select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables generated multi-threading code to run concurrently on multicore processors.

11 For the best performance, in the Configuration Parameters dialog box, on the **Solver** pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

12 In your model, click the build button or enter **Ctrl+B**. The software performs the actions you selected for **Build action** in the model Configuration Parameters, under Code Generation > IDE Link.

Avoiding Lock-Up in Free-Running, Multirate, Multitasking Models

Use caution if you select free-running mode for a multirate, multitasking models. Because of the rate monotonic scheduling requirement in Linux, the scheduler runs threads with a SCHED_FIFO scheduling policy. A process scheduled with SCHED_FIFO prevents other process from running while it is ready to run. Therefore, if no blocking peripherals appear in the model, the entire Linux system can become unresponsive while you are running the generated code. Such lock-up can even preempt the shell window from running. To avoid this lock-up, apply one of the following solutions:

- Set **Scheduling Mode** to `real_time`.
- Include a blocking device driver, such as a UDP block, in your model that suspends running thread while data is not available.
- Raise the shell window priority above the base-rate priority so you can kill the process running with SCHED_FIFO class.

Limitations

Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux

Stack Profiling and Execution Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux.

Example: Build Generated Code on a BeagleBoard Running Linux

In this section...

"Overview" on page 52-7

"Configure the Windows Host" on page 52-7

"Configure the BeagleBoard" on page 52-7

"Configure MATLAB" on page 52-8

Overview

This example shows you how to generate code on a Windows host, and then build it remotely on a BeagleBoard running Linux.

Configure the Windows Host

Download and install the following PuTTY utilities from http://www.chiark.greenend.org.uk/~sgtatham/putty/download:

- Plink (a command-line interface to the PuTTY back ends)
- PSCP (an SCP client, i.e. command-line secure file copy)

Warning **PuTTY software may be illegal in countries where encryption is prohibited.**

Configure the BeagleBoard

Install the GNU-compiler toolchain on the BeagleBoard. Open a terminal session with the Linux command line on the BeagleBoard.

For example, on a board running the Angstrom Linux distribution, enter the following commands:

```
root@beagleboard:~# opkg install binutils
root@beagleboard:~# opkg install gcc
root@beagleboard:~# opkg install gcc-symlinks
root@beagleboard:~# opkg install cpp-symlinks
```

```
root@beagleboard:~# opkg install libstdc++-dev
root@beagleboard:~# opkg install make-dev
```

Configure MATLAB

Configure MATLAB to generate code.

1. Enter `xmakefilesetup` at the MATLAB command line. In the XMakefile Configuration dialog, set **template** to `gmake` and **configuration** to `gcc_target`.

2. For the Target Preferences block in your model, set **IDE/Toolchain** to `Eclipse`, and set the **Board** to `BeagleBoard ARM`.

3. In your model, open **Simulation** > **Configuration Settings**. Under **Code Generation**, select the **IDE Link** pane. Set **Build format** to `Makefile`, and set **Build Action** to `Create_makefile`.

4. Generate the code by pressing **Ctrl B**.

5. Use the load method to load the buildInfo.mat file from the project directory to the BeagleBoard. For example, at the MATLAB command prompt enter:

   ```
   >> s =load(`sumdiff_codegen_eclipseide\buildInfo.mat')
   ```

6. Use `remoteBuild` function to build the code on the BeagleBoard. For example, enter:

   ```
   >> remoteBuild(s.buildInfo, '/home/root', '144.212.110.193',
   'root', 'password', 'C:\utils\putty')
   ```

The GNU compiler toolchain automatically builds and runs the software on the BeagleBoard.

For more information, see `remoteBuild`.

Example: Build Generated Code on a Linux Host, Then Run It Remotely on BeagleBoard

In this section...

"Overview" on page 52-9

"Prerequisites" on page 52-9

"Set up your environment for Linux-ARM Code Generation" on page 52-9

"Generate Code for Linux-ARM" on page 52-12

"External Mode Simulation" on page 52-12

Overview

This example shows you how to build a target application locally on a Linux host using Eclipse IDE, and then run the program remotely on the ARM processor of a BeagleBoard running Angstrom Linux.

Prerequisites

To generate code for BeagleBoard, first obtain and install the following third-party software:

- Eclipse™ IDE
- OpenEmbedded build system for BeagleBoard Angstrom

You can find instructions for installing Eclipse IDE here: Chapter 48, "Working with Eclipse IDE"

You can find instructions for building BeagleBoard OpenEmbedded here: The Angstrom Distribution

Set up your environment for Linux-ARM Code Generation

Before attempting to generate code for BeagleBoard, it is important to configure your environment correctly. Use the following configuration steps to generating code for the Linux-ARM target on the BeagleBoard.

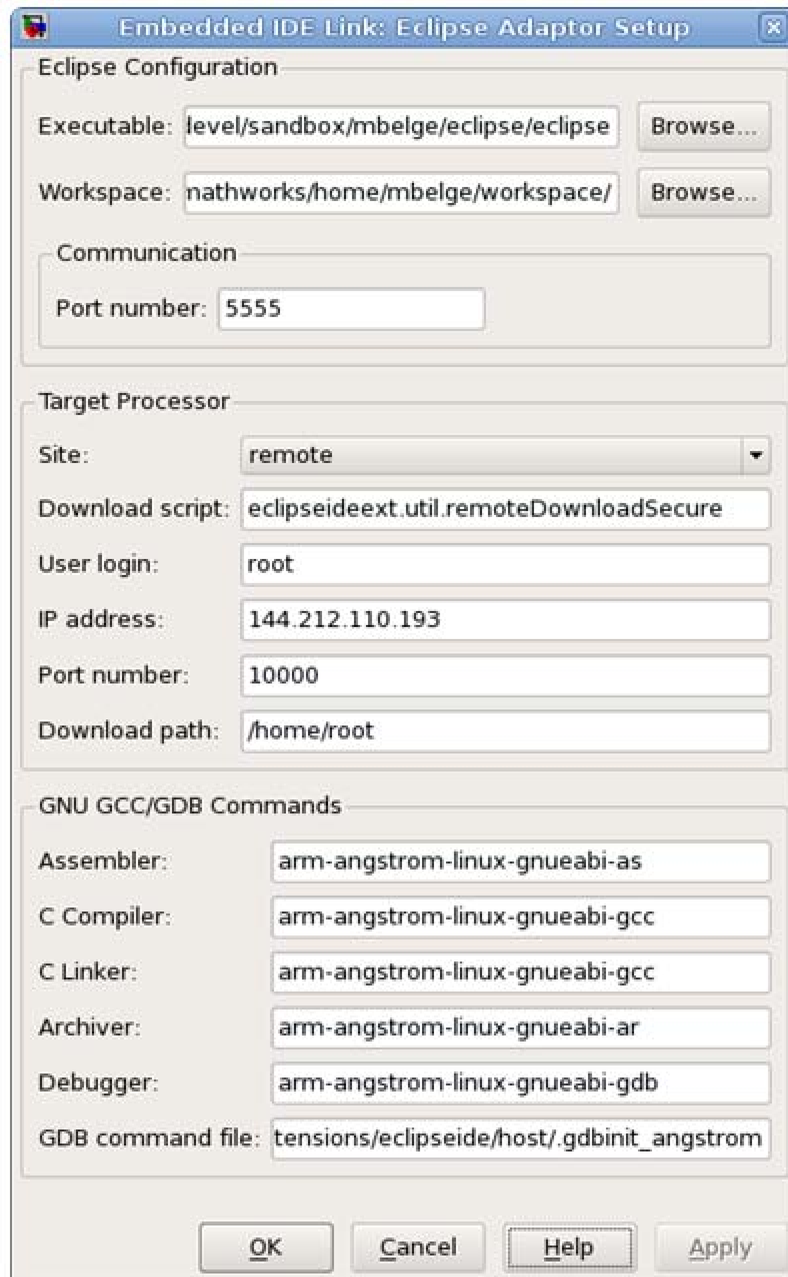
These instructions assume you have already built the OpenEmbedded Linux-Angstrom distribution. Make sure that you build the bitbake recipe for gdb-cross and install gdbserver on the target board. For example, use the `bitbake gdb-cross` command to build the GDB debugger that runs on your host computer. Then use opkg to install gdbserver. For example, enter: `opkg install gdbserver`. The Eclipse project generated by the coder product uses local GDB debugger and a GDB server running on the target board to enable debugging support. You also need an Ethernet connection to your board to debug the generated code.

1 Set up your MATLAB environment to see OpenEmbedded GNU compiler toolchain and the OpenEmbedded root file system by executing the following from MATLAB command line.

```
setenv('PATH', ['<pathtoOEtree>/oe/angstrom-dev/cross/armv7a/bin' ':' getenv('PATH')])
setenv('OETREE', '<pathtoOEtree>/oe')
```

The first setenv command adds the path to the GNU compiler toolchain for MATLAB to locate compiler, linker, and so on, for the BeagleBoard. Internally, MATLAB uses OETREE environment variable to locate the root file system of the BeagleBoard and shared libraries used for linking. Folder information extracted from OETREE environment variable is used to set up the correct GDB initialization script.

2 On the MATLAB command line execute 'eclipseidesetup' and set the entries in the Eclipse Adaptor Setup as shown here.


Embedded IDE Link: Eclipse Adaptor Setup
Eclipse Configuration
Executable:
level/sandbox/mbelge/eclipse/eclipse
Browse...
Workspace:
nathworks/home/mbelge/workspace/
Browse...
Communication
Port number:
5555
Target Processor
Site:
remote
Download script:
eclipseideext.util.remoteDownloadSecure
User login:
root
IP address:
144.212.110.193
Port number:
10000
Download path:
/home/root
GNU GCC/GDB Commands
Assembler:
arm-angstrom-linux-gnueabi-as
C Compiler:
arm-angstrom-linux-gnueabi-gcc
C Linker:
arm-angstrom-linux-gnueabi-gcc
Archiver:
arm-angstrom-linux-gnueabi-ar
Debugger:
arm-angstrom-linux-gnueabi-gdb
GDB command file:
tensions/eclipseide/host/.gdbinit_angstrom
OK
Cancel
Help
Apply

In the Target Processor pane, enter the IP address of your board. Set the Download script to 'eclipseideext.util.remoteDownloadSecure' and the GDB command file to '$(MATLAB_ROOT)/toolbox/idelink/extensions/eclipseide/host/.gdbinit_angstrom'.

Generate Code for Linux-ARM

1. Enter `sumdiff_codegen` at the MATLAB command line. This opens the sumdiff_codegen model.

2. Open the Target Preferences block in the model, set the **IDE/Tool Chain** to `Eclipse`, and set **Board** to `BeagleBoard ARM`.

3. Open Simulation -> Configuration Parameters -> Code Generation -> IDE Link dialog. In the Compiler options string edit box enter -mfloat-abi=softfp. This option is an Angstrom distribution-specific compiler option required to generate correct floating point code for BeagleBoard.

4. Generate and build the ARM code. If build process is successful, you code starts running on the board automatically.

5. You can, alternatively, run the generate code by downloading to the target file system. On a Linux command shell, you can execute the following:

```
> scp <pathtoarmexecutable>/sumdiff_codegen root@10.10.10.1:/home/root/.
> ssh root@10.10.10.1 /home/root/sumdiff_codegen
```

External Mode Simulation

1. To run the simulation in external mode, connect a scope to Out1 port on the sumdiff_codegen model. Open Simulation -> Configuration Parameters -> Code Generation -> Interface dialog. Enter the IP address of the BeagleBoard in the MEX-file Arguments edit box under Host/Target interface. The IP address of the board is needed for External Mode TCP/IP connection. If the IP address of your board is 10.10.10.1, for example, enter '10.10.10.1' in the MEX-file Arguments edit box. Click OK and close the dialog.

2. Click Simulation -> Connect To Target (Ctrl-T) to establish External Mode connection.

3. Double click the Simulink Scope and examine the data retrieved from Linux-ARM target. Observe the Scope displaying zeros indicating correct operation.

Embedded Linux Topics

Troubleshooting "sched_setaffinity: Bad address" Error

When you build an executable, if the build environment and target use different libc versions, the build process terminates immediately with the following error:

```
**starting the model**
Call to sched_setaffinity returned error status (-1).
sched_setaffinity: Bad addres
```

To work around this problem, you can add `-static` to the linker options. However, linking the libraries statically increases the size of the executable. To configure the linker options, complete the following steps:

1. Press **Ctrl+E** to open the model configuration parameters.
2. Select **Code Generation** > **IDE Link**.
3. Add `-static` to the **Linker options string**.

To solve this problem, update the development and target software so they match. For example, in the case of the TMS320DM355 DVEVM, see the "Installing the Software" topic in Texas Instruments *TMS320DM355 DVEVM Getting Started Guide*, literature number SPRUF73.

Working with Microsoft Windows Target

- "Preparing Models to Run on Windows" on page 53-2
- "Scheduler" on page 53-3

Preparing Models to Run on Windows

To build an executable that runs on Windows , perform the following steps:

1. Install and configure Eclipse IDE according to the instructions in Chapter 48, "Working with Eclipse IDE".

2. Enter `idelinklib_common` at the MATLAB prompt. This action opens the idelinklib_common library.

3. Copy the Target Preferences block to your model.

4. In the **Initialize Configuration Parameters** dialog box, set the IDE to Eclipse, and select the processor for which you are generating code.

5. Set **Operating System** to `None`, `Windows`.

   Selecting `Windows` creates a **Windows** tab, which you can use to set **Scheduling Mode**.

6. Set the **Scheduling Mode** to one of these options:

   - If you select `real-time`, the model uses a timer to trigger the base rate at regular periods.
   - If you select `free-running`, the model does not use a timer. Instead, the model completes each process or thread before running the next one.

7. In IDE Link, configure the model to build and execute:

   a. In the model, select **Simulation** > **Configuration Parameters**.

   b. Select the **Code Generation** > **IDE Link** pane.

   c. Set **Build action** to `Build and execute`.

8. Build the model. Select **Tools** > **Code Generation** > **Build Model**.

   After the build completes, Embedded Coder software downloads the executable to the remote system and runs it.

Scheduler

In this section...

"Selecting the Operating System and Scheduling Mode" on page 53-3

"Base Rate" on page 53-4

"Running Target Applications on Multicore Processors" on page 57-10

"Limitations" on page 53-6

Selecting the Operating System and Scheduling Mode

The following table refers to the **Operating System** and **Scheduling Mode** options in the Target Preferences block.

Operating System	Scheduling Mode	Behavior
Windows	free_running	The model generates multithreaded, free-running code. Each rate in the model maps to a separate thread in the generated code. Multithreaded code can potentially run faster than single-threaded code.
Windows	real_time	The model generates multithreaded, real-time code: Each rate in the Simulink model runs at the rate specified in the model. For example, a 1-s rate runs at exactly 1-s intervals.
None	Not applicable	The model generates free-running code that runs in an infinite while loop with no timing.

For more information, see "Scheduling Considerations" in the *Simulink Coder User's Guide*.

Base Rate

The base rate in the model maps to a thread and runs as fast as possible. In Windows target, the timer resolution is 1 ms. The base rate priority selection in the OS tab allows you to set a static priority for the base rate task.

The Windows OS does not have a selection. The default base rate priority is THREAD_PRIORITY_HIGHEST (10) and the process running the generated code has NORMAL_PRIORITY_CLASS.

The process running single-tasking models has Default scheduling policy when model is single-tasking or there is a single rate in the model. Static priority of the process is 0 in this case.

Running Target Applications on Multicore Processors

If you are generating code for a processor running Linux or VxWorks, you can elect to partition the code such that each rate is placed in its own thread. With code generated from a multi-rate model, the multi-threaded application will be enabled for concurrent multicore execution, as scheduled by the target operating system.

1. Create a multi-rate Simulink model.

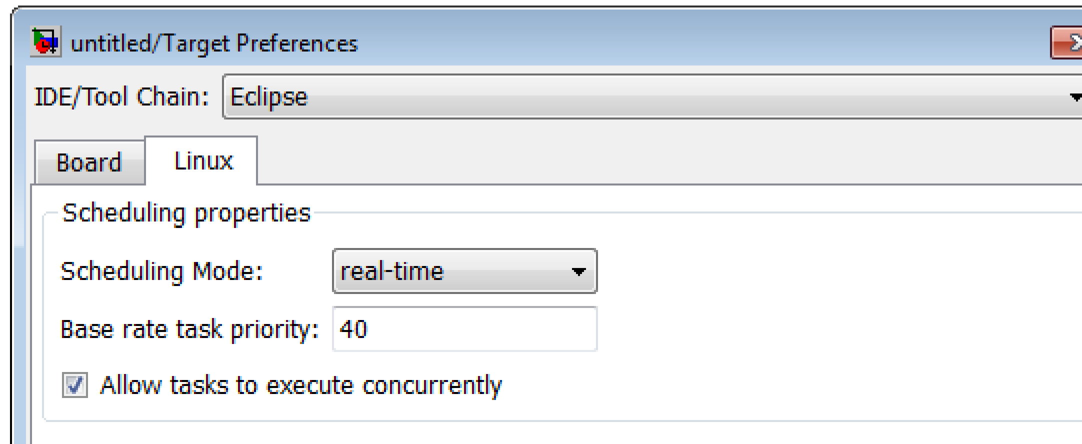
2. Add a Target Preferences block to your model as described in the "Target Preferences" on page 43-4 section.

3. Verify that your model uses a Rate Transition block to transition between rates.

4. Clear the **Ensure deterministic data transfer** checkbox of the Rate Transition block. This action forces the Rate Transition block to use the most recent data available.

5. In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

6. In the Linux or VxWorks tab of the Target Preferences block, select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables the generated multi-threaded code to run concurrently on multicore processors.



7. In the Configuration Parameters dialog box, on the Solver pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

8. Open the Rate Transition block and clear the **Ensure deterministic data transfer** checkbox. This action forces the Rate Transition block to use the most recent data.

9. In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

10. Select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables generated multi-threading code to run concurrently on multicore processors.

11. For the best performance, in the Configuration Parameters dialog box, on the **Solver** pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

12. In your model, click the build button or enter **Ctrl+B**. The software performs the actions you selected for **Build action** in the model Configuration Parameters, under Code Generation > IDE Link.

Limitations

Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux

If you use Embedded Coder with Eclipse to build and run applications on processors running Windows or Linux: The stack profiling and real-time execution profiling is only available for ARM® processors running Linux. Profiling is not available for Intel x86/Pentium and AMD K5/K6/Athlon processors running Windows or Linux.

Working with Texas Instruments Code Composer Studio IDE

- "Code Composer Studio" on page 54-2
- "Getting Started" on page 54-4
- "Automation Interface" on page 54-10
- "Project Generator" on page 54-58
- "Exporting Filter Coefficients from FDATool" on page 54-69
- "Tutorial: Using XMakefile with Code Composer Studio 4.x" on page 54-85
- "Reported Limitations and Tips" on page 54-91

Code Composer Studio

In this section...

"Using Code Composer Studio with Embedded Coder Software" on page 54-2
"Default Project Configuration" on page 54-2

Using Code Composer Studio with Embedded Coder Software

Texas Instruments (TI) facilitates development of software for TI DSPs by offering Code Composer Studio (CCS) Integrated Development Environment (IDE). Used in combination with Embedded Coder software and Simulink Coder software, CCS provides an integrated environment that, once installed, requires no coding.

Executing code generated from Simulink Coder software on a particular target requires that you tailor the code to the specific hardware target. Target-specific code includes I/O device drivers and interrupt service routines (ISRs). The software must use CCS to compile and link the generated source code in order to load and execute on a TI DSP. To help you to build an executable, Embedded Coder software uses Embedded Coder software to start the code building process within CCS. After you download your executable to your target and run it, the code runs wholly on the target. You can access the running process only from the CCS debugging tools or across a link using Embedded Coder software. A wide range of Texas Instruments DSPs are supported:

- TI's C2000
- TI's C5000™
- TI's C6000™

Default Project Configuration

CCS offers two standard project configurations, `Release` and `Debug`. Project configurations define sets of project build options. When you specify the build options at the project level, the options apply to all files in your project. For more information about the build options, refer to your TI documentation. The

models you build with Embedded Coder software use a custom configuration that provides a third combination of build and optimization settings — `CustomMW`.

Default Build Options in the CustomMW Configuration

The default settings for `CustomMW` are the same as the `Release` project configuration in CCS, except for the compiler options.

Your CCS documentation provides complete details on the compiler build options. You can change the individual settings or the build configuration within CCS.

Getting Started

In this section...

"Overview" on page 54-4

"Configuration Information" on page 54-7

Overview

- "Automation Interface" on page 54-5
- "Project Generator" on page 54-6
- "Verification" on page 54-7

Embedded Coder software enables you to use MATLAB functions to communicate with Code Composer Studio software and with information stored in memory and registers on a processor. With the `ticcs` objects, you can transfer information to and from Code Composer Studio software and with the embedded objects you get information about data and functions stored in your signal processor memory and registers, as well as information about functions in your project.

Embedded Coder lets you build, test, and verify automatically generated code using MATLAB, Simulink, Simulink Coder, and the Code Composer Studio integrated development environment. You can use Embedded Coder to verify code executing within the Code Composer Studio software environment using a model in Simulink software. This processor-in-the-loop testing environment uses code automatically generated from Simulink models by Embedded Coder software. A range of Texas Instruments targets are supported:

- TI's C2000
- TI's C5000
- TI's C6000

With Embedded Coder , you can use MATLAB software and Simulink software to interactively analyze, profile and debug processor-specific code execution behavior within CCS. In this way, Embedded Coder automates

deployment of the complete embedded software application and makes it easy for you to assess possible differences between the model simulation and processor code execution results.

Embedded Coder consists of these components:

- Project Generator—add embedded framework code to the C code generated from Simulink models, and package as a complete IDE project
- Automation Interface—use functions in the MATLAB command window to access and manipulate data and files in the IDE and on the processor
- Verification—verify how your programs run on your processor

With Embedded Coder, you create objects that connect MATLAB software to Code Composer Studio software.

Note Embedded Coder uses objects. You work with them the way you use all MATLAB objects. You can set and get their properties, and use their methods to change them or manipulate them and the IDE to which they refer.

The next sections describe briefly the components of Embedded Coder software.

Automation Interface

The automation interface component is a collection of methods that use the Code Composer Studio API to communicate between MATLAB software and Code Composer Studio. With the interface, you can do the following:

- Automate complex tasks in the development environment by writing MATLAB software scripts to communicate with the IDE, or debug and analyze interactively in a live MATLAB software session.
- Automate debugging by executing commands from the powerful Code Composer Studio software command language.
- Exchange data between MATLAB software and the processor running in Code Composer Studio software.

- Set breakpoints, step through code, set parameters and retrieve profiling reports.
- Automate project creation, including adding source files, include paths, and preprocessor defines.
- Configure batch building of projects.
- Debug projects and code.
- Execute API Library commands.

The automation interface provides an application program interface (API) between MATLAB software and Code Composer Studio. Using the API, you can create new projects, open projects, transfer data to and from memory on the processor, add files to projects, and debug your code.

Project Generator

The Project Generator component is a collection of methods that use the Code Composer Studio API to create projects in Code Composer Studio and generate code with Embedded Coder. With the interface, you can do the following:

- Automated project-based build process

  Automatically create and build projects for code generated by Embedded Coder.
- Customize code generation

  Use Embedded Coder with any Embedded Coder system target file (STF) to generate processor-specific and optimized code.
- Customize the build process
- Automate code download and debugging

  Rapidly and effortlessly debug generated code in the Code Composer Studio software debugger, using either the instruction set simulator or real hardware.
- Create and build CCS projects from Simulink software models. Project Generator uses Simulink Coder software or Embedded Coder software to build projects that work with C2000 software, C5000 software, and C6000 software processors.

- Highly customized code generation with the system target file `idelink_ert.tlc` and `idelink_grt.tlc` that enable you to use the Configuration Parameters in your model to customize your generated code.
- Automate the process of building and downloading your code to the processor, and running the process on your hardware.

Verification

Verifying your processes and algorithms is an essential part of developing applications. The components of Embedded Coder combine to provide the following verification tools for you to apply as you develop your code:

Processor in the Loop Simulation. Use simulation techniques to verify generated code running in an instruction set simulator or real processor environment.

Execution Profiling. Gather execution profiling timing measurements with Code Composer Studio to establish the timing requirements of your algorithm. See .

Configuration Information

To determine whether Embedded Coder is installed on your system, type this command at the MATLAB software prompt.

```
ver
```

When you enter this command, MATLAB software displays a list of the installed products. Look for a line similar to the following:

```
Embedded Coder          Version 4.x    (Release Specifier)
```

To get a bit more information about the software, such as the functions provided and where to find demos and help, enter the following command at the prompt:

```
help ticcs
```

If you do not see the listing, or MATLAB software does not recognize the command, you need to install Embedded Coder. Without the software, you cannot use MATLAB software with the objects to communicate with CCS.

Verifying Your Code Composer Studio Installation

To verify that CCS is installed on your machine and has at least one board configured, enter

```
ccsboardinfo
```

at the MATLAB software command line. With CCS installed and configured, MATLAB software returns information about the boards that CCS recognizes on your machine, in a form similar to the following listing.

```
Board Board                       Proc Processor  Processor
 Num  Name                        Num  Name        Type
 ---  ------------------------------  ---  -------------
1  C6xxx Simulator (Texas Instrum .0   6701     TMS320C6701
0  C6x13 DSK (Texas Instruments)   0   CPU      TMS320C6x1x
```

If MATLAB software does not return information about any boards, open your CCS installation and use the Setup Utility in CCS to configure at least one board.

As a final test, start CCS to ensure that it starts up successfully. For Embedded Coder to operate with CCS, the CCS IDE must be able to run on its own.

Embedded Coder uses objects to create:

- Connections to the Code Composer Studio Integrated Development Environment (CCS IDE)
- Connections to the RTDX™ (RTDX) interface. This object is a subset of the object that refers to the CCS IDE.

Concepts to know about the objects in this toolbox are covered in these sections:

- “Constructing ticcs Objects” on page 54-48

- "ticcs Properties and Property Values" on page 54-50
- "Overloaded Functions for ticcs Objects" on page 54-50

Refer to MATLAB Classes and Objects in your MATLAB documentation for more details on object-oriented programming in MATLAB software.

Many of the objects use COM server features to create handles for working with the objects. Refer to your MATLAB documentation for more information about COM as used by MATLAB software.

Automation Interface

In this section...

"Getting Started with Automation Interface" on page 54-10

"Getting Started with RTDX" on page 54-27

"Constructing ticcs Objects" on page 54-48

"ticcs Properties and Property Values" on page 54-50

"Overloaded Functions for ticcs Objects" on page 54-50

"ticcs Object Properties" on page 54-51

Getting Started with Automation Interface

- "Introducing the Automation Interface Tutorial" on page 54-10
- "Selecting Your Processor" on page 54-14
- "Creating and Querying Objects for CCS IDE" on page 54-16
- "Loading Files into CCS" on page 54-18
- "Working with Projects and Data" on page 54-20
- "Closing the Links or Cleaning Up CCS IDE" on page 54-26

Introducing the Automation Interface Tutorial

Embedded Coder provides a connection between MATLAB software and a processor in CCS. You can use objects to control and manipulate a signal processing application using the computational power of MATLAB software. This approach can help you debug and develop your application. Another possible use for automation is creating MATLAB scripts that verify and test algorithms that run in their final implementation on your production processor.

Before using the functions available with the objects, you must select a processor to be your processor because any object you create is specific to a designated processor and a designated instance of CCS IDE. Selecting

a processor is necessary for multiprocessor boards or multiple board configurations of CCS.

When you have one board with a single processor, the object defaults to the existing processor. For the objects, the simulator counts as a board; if you have both a board and a simulator that CCS recognizes, you must specify the processor explicitly.

To get you started using objects for CCS IDE software, Embedded Coder includes a tutorial that introduces you to working with data and files. As you work through this tutorial, you perform the following tasks that step you through creating and using objects for CCS IDE:

1. Select your processor.
2. Create and query objects to CCS IDE.
3. Use MATLAB software to load files into CCS IDE.
4. Work with your CCS IDE project from MATLAB software.
5. Close the connections you opened to CCS IDE.

The tutorial provides a working process (a *workflow*) for using Embedded Coder and your signal processing programs to develop programs for a range of Texas Instruments processors.

During this tutorial, you load and run a digital signal processing application on a processor you select. The tutorial demonstrates both writing to memory and reading from memory in the ““Working with Projects and Data” on page 54-20” portion of the tutorial.

You can use the `read` and `write` methods, as described in this tutorial, to read and write data to and from your processor.

The tutorial covers the object methods and functions for Embedded Coder. The functions listed in the first table apply to CCS IDE independent of the objects — you do not need an object to use these functions. The methods listed in the second and third table requires a `ticcs` object that you use in the method syntax:

Functions for Working With Embedded Coder. The following functions do not require a `ticcs` object as an input argument:

Function	Description
ccsboardinfo	Return information about the boards that CCS IDE recognizes as installed on your PC.
`ticcs`	Construct an object to communicate with CCS IDE. When you construct the object you specify the processor board and processor.

Methods for Working with ticcs Objects. The methods in the following table require a `ticcs` object as an input argument:

Method	Description
`address`	Return the address and page for an entry in the symbol table in CCS IDE.
display	Display the properties of an object to CCS IDE and RTDX.
halt	Terminate execution of a process running on the processor.
info	Return information about the processor or information about open RTDX channels.
`isrtdxcapable`	Test whether your processor supports RTDX communications.
isrunning	Test whether the processor is executing a process.
read	Retrieve data from memory on the processor.
restart	Restore the program counter (PC) to the entry point for the current program.

Method	Description
run	Execute the program loaded on the processor.
visible	Set whether CCS IDE window is visible on the desktop while CCS IDE is running.
write	Write data to memory on the processor.

Methods for Embedded Objects. The methods in the following table enable you to manipulate programs and memory with an embedded object:

Method	Description
list	Return various information listings from Code Composer Studio software.
read	Read the information at the location accessed by an object into MATLAB software as numeric values. Demonstrated with a numeric, string, structure, and enumerated objects.
write	Write to the location referenced by an object. Demonstrated with numeric, string, structure, and enumerated objects.

Running Code Composer Studio Software on Your Desktop — Visibility. When you create a `ticcs` object , Embedded Coder starts CCS in the background.

When CCS IDE is running in the background, it does not appear on your desktop, in your task bar, or on the **Applications** page in the Task Manager. It does appear as a process, `cc_app.exe`, on the **Processes** tab in Microsoft Windows Task Manager.

You can make the CCS IDE visible with the function `visible`. The function `isvisible` returns the status of the IDE—whether it is visible on your desktop. To close the IDE when it is not visible and MATLAB software is not running, use the **Processes** tab in Microsoft Windows Task Manager and look for `cc_app.exe`.

If a link to CCS IDE exists when you close CCS, the application does not close. Microsoft Windows software moves it to the background (it becomes invisible). Only after you clear all links to CCS IDE, or close MATLAB software, does closing CCS IDE unload the application. You can see if CCS IDE is running in the background by checking in the Microsoft Windows Task Manager. When CCS IDE is running, the entry `cc_app.exe` appears in the **Image Name** list on the **Processes** tab.

When you close MATLAB software while CCS IDE is not visible, MATLAB software closes CCS if it started the IDE. This happens because the operating system treats CCS as a subprocess in MATLAB software when CCS is not visible. Having MATLAB software close the invisible IDE when you close MATLAB software prevents CCS from remaining open. You do not need to close it using Microsoft Windows Task Manager.

If CCS IDE is not visible when you open MATLAB software, closing MATLAB software leaves CCS IDE running in an invisible state. MATLAB software leaves CCS IDE in the visibility and operating state in which it finds it.

Running the Interactive Tutorial. You have the option of running this tutorial from the MATLAB software command line or entering the functions as described in the following tutorial sections.

To run the tutorial in MATLAB software, click `run ccstutorial`. This command opens the tutorial in an interactive mode where the tutorial program provides prompts and text descriptions to which you respond to move to the next portion of the lesson. The interactive tutorial covers the same information provided by the following tutorial sections. You can view the tutorial file by clicking `ccstutorial.m`.

Selecting Your Processor

Links for CCS IDE provides two tools for selecting a board and processor in multiprocessor configurations. One is a command line tool called ccsboardinfo

which prints a list of the available boards and processors. So that you can use this function in a script, `ccsboardinfo` can return a MATLAB software structure that you use when you want your script to select a board without your help.

Note The board and processor you select is used throughout the tutorial.

---

1. To see a list of the boards and processors installed on your PC, enter the following command at the MATLAB software prompt:

   ```
   ccsboardinfo
   ```

   MATLAB software returns a list that shows you all the boards and processors that CCS IDE recognizes as installed on your system.

2. To use the Selection Utility, `boardprocsel`, to select a board, enter

   ```
   [boardnum,procnum] = boardprocsel
   ```

   When you use `boardprocsel`, you see a dialog box similar to the following. Note that some entries vary depending on your board set.

3. Select a board name and processor name from the lists.

   You are selecting a board and processor number that identifies your particular processor. When you create the object for CCS IDE in the next section of this tutorial, the selected board and processor become the processor of the object.

4. Click **Done** to accept your board and processor selection and close the dialog box.

   `boardnum` and `procnum` now represent the **Board name** and **Processor name** you selected — `boardnum` = 1 and `procnum` = 0

Creating and Querying Objects for CCS IDE

In this tutorial section, you create the connection between MATLAB software and CCS IDE. This connection, or object, is a MATLAB software object that you save as variable `IDE_Obj`.

You use function `ticcs` to create objects. When you create objects, `ticcs` input arguments let you define other object property values, such as the global timeout. Refer to the `ticcs` reference documentation for more information on these input arguments.

Use the generated object `IDE_Obj` to direct actions to your processor. In the following tasks, `IDE_Obj` appears in all function syntax that interact with CCS IDE and the processor:

1. Create an object that refers to your selected board and processor. Enter the following command at the prompt.

   ```
   IDE_Obj=ticcs('boardnum',boardnum,'procnum',procnum)
   ```

   If you were to watch closely, and your machine is not too fast, you see Code Composer Studio software appear briefly when you call ticcs. If CCS IDE was not running before you created the new object, CCS starts and runs in the background.

2. Enter `visible(IDE_Obj,1)` to force CCS IDE to be visible on your desktop.

   Usually, you need to interact with Code Composer Studio software while you develop your application. The first function in this tutorial, visible, controls the state of CCS on your desktop. `visible` accepts Boolean inputs that make CCS either visible on your desktop (input to `visible` = 1) or invisible on your desktop (input to `visible` = 0). For this tutorial, use `visible` to set the CCS IDE visibility to `1`.

3. Next, enter `display(IDE_Obj)` at the prompt to see the status information.

   ```
   TICCS Object:
     API version      : 1.0
     Processor type   : Cxx
     Processor name   : CPU
     Running?         : No
     Board number     : 0
   ```

```
Processor number : 0
Default timeout  : 10.00 secs

RTDX channels    : 0
```

Embedded Coder provides three methods to read the status of a board and processor:

- `info` — Return a structure of testable board conditions.
- `display` — Print information about the processor.
- `isrunning` — Return the state (running or halted) of the processor.
- `isrtdxcapable` — Return whether the hardware supports RTDX.

4 Type `linkinfo = info(IDE_Obj)`.

The `IDE_Obj` link status information provides information about the hardware as follows:

```
linkinfo =

      boardname: 'Cxxxx Device Simulator'
       procname: 'CPU_1'
    isbigendian: 0
         family: 320
      subfamily: 103
      revfamily: 11
     processortype: 'simulator'
     revsilicon: 0
        timeout: 10
```

5 Check whether the processor is running by entering

```
runstatus = isrunning(IDE_Obj)
```

MATLAB software responds, indicating that the processor is stopped, as follows:

```
runstatus =

     0
```

6 At last, verify that the processor supports RTDX communications by entering

```
usesrtdx = isrtdxcapable(IDE_Obj)
usesrtdx =

     1
```

Loading Files into CCS

You have established the connection to a processor and board. Using three methods you learned about the hardware, whether it was running, its type, and whether CCS IDE was visible. Next, the processor needs something to do.

In this part of the tutorial, you load the executable code for the processor CPU in CCS IDE. Embedded Coder includes a CCS project file. Through the next tasks in the tutorial, you locate the tutorial project file and load it into CCS IDE. The `open` method directs CCS to load a project file or workspace file.

Note CCS has workspace and workspace files that are different from the MATLAB workspace files and workspace. Remember to monitor both workspaces.

After you have executable code running on your processor, you can exchange data blocks with it. Exchanging data is the purpose of the objects provided by Embedded Coder software.

1 To load the appropriate project file to your processor, enter the following command at the MATLAB software prompt. `getdemoproject` is a specialized function for loading Embedded Coder demo files. It is not supported as a standard Embedded Coder function.

```
demopjt= getDemoProject(IDE_Obj,'ccstutorial')

demopjt.ProjectFile

ans =
```

```
C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxx\ccstut.pjt

demoPjt.DemoDir

ans =

C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxx
```

Your paths may be different if you use a different processor. Note where the software stored the demo files on your machine. In general, Embedded Coder software stores the demo project files in

```
EmbIDELinkCCDemos_v#.#
```

Embedded Coder creates this folder in a location where you have write permission. There are two locations where Embedded Coder software tries to create the demo folder, in the following order:

a In a temporary folder on your C drive, such as `C:\temp\`.

b If Embedded Coder software cannot use the `temp` folder, you see a dialog box that asks you to select a location to store the demos.

2 Enter the following command at the MATLAB command prompt to build the processor executable file in CCS IDE.

```
build(IDE_Obj,'all',20)
```

You may get an error related to one or more missing `.lib` files. If you installed CCS IDE in a folder other than the default installation folder, browse in your installation folder to find the missing file or files. Refer to the path in the error message as an indicator of where to find the missing files.

3 Change your working folder to the demo folder and enter `load(IDE_Obj,'`*`projectname`*`.out')` to load the processor execution file, where *`projectname`* is the tutorial you chose, such as `ccstut_67x`.

You have a loaded program file and associated symbol table to the IDE and processor.

4 To determine the memory address of the global symbol `ddat`, enter the following command at the prompt:

```
ddata = address(IDE_Obj,'ddat')
ddata =

   1.0e+009 *

      2.1475         0
```

Your values for `ddata` may be different depending on your processor.

Note The symbol table is available after you load the program file into the processor, not after you build a program file.

---

5 To convert `ddata` to a hexadecimal string that contains the memory address and memory page, enter the following command at the prompt:

```
dec2hex(ddata)
```

MATLAB software displays the following response, where the memory page is `0x00000000` and the address is `0x80000010`.

```
ans =

80000010
00000000
```

Working with Projects and Data

After you load the processor code, you can use Embedded Coder functions to examine and modify data values in the processor.

When you look at the source file listing in the CCS IDE Project view window, there should be a file named `ccstut.c`. Embedded Coder ships this file with the tutorial and includes it in the project.

ccstut.c has two global data arrays — ddat and idat — that you declare and initialize in the source code. You use the functions read and write to access these processor memory arrays from MATLAB software.

Embedded Coder provides three functions to control processor execution — run, halt, and restart.

1. To demonstrate these commands, use the following function to add a breakpoint to line 64 of ccstut.c.

   ```
   insert(IDE_Obj,'ccstut.c',64)
   ```

   Line 64 is

   ```
   printf("Embedded Coder: Tutorial - Memory Modified by Matlab!\n");
   ```

   For information about adding breakpoints to a file, refer to insert in the online Help system. Then proceed with the tutorial.

2. To demonstrate the new functions, try the following functions.

   ```
   halt(IDE_Obj)                 % Halt the processor.
   restart(IDE_Obj)              % Reset the PC to start of program.
   run(IDE_Obj,'runtohalt',30);  % Wait for program execution to stop at
                               % breakpoint (timeout = 30 seconds).
   ```

   When you switch to viewing CCS IDE, you see that your program stopped at the breakpoint you inserted on line 64, and the program printed the following messages in the CCS IDE **Stdout** tab. Nothing prints in the MATLAB command window:

   ```
   Embedded Coder: Tutorial - Initialized Memory
   Double Data array = 16.3 -2.13 5.1 11.8
   Integer Data array = -1-508-647-7000 (call me anytime!)
   ```

3. Before you restart your program (currently stopped at line 64), change some values in memory. Perform one of the following procedures based on your processor.

   C5xxx processor family — Enter the following functions to demonstrate the read and write functions.

a Enter
`ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4).`

MATLAB software responds with

```
ddatv =


     16.3000   -2.1300    5.1000   11.8000
```

b Enter `idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int16',4).`

Now MATLAB software responds

```
idatv =

-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(IDE_Obj,address(IDE_Obj,'idat'),'int8',4)

idatv =

1 0 -4 1
```

c You can change the values stored in `ddat` by entering
```
write(IDE_Obj,address(IDE_Obj,'ddat'),double([pi 12.3
exp(-1)...
sin(pi/4)]))
```

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

d To change `idat`, enter

```
write(IDE_Obj,address(IDE_Obj,'idat'),int32([1:4]))
```

Here you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

e Start the program running again by entering the following command:

```
run(IDE_Obj,'runtohalt',30);
```

The `Stdout` tab in CCS IDE reveals that `ddat` and `idat` contain new values. Next, read those new values back into MATLAB software.

f Enter `ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)`.

```
ddatv =

3.1416 12.3000 0.3679 0.7071
```

`ddatv` contains the values you wrote in step c.

g Verify that the change to `idatv` occurred by entering the following command at the prompt:

```
idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int16',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =

1 2 3 4
```

h Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(IDE_Obj);
```

C6xxx processor family — Enter the following commands to demonstrate the `read` and `write` functions.

a Enter `ddatv = read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)`.

MATLAB software responds with

```
ddatv =


     16.3000    -2.1300     5.1000    11.8000
```

b Enter `idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int16',4)`.

MATLAB software responds

```
idatv =

-1 508 647 7000
```

If you used 8-bit integers (`int8`), the returned values would be incorrect.

```
idatv=read(IDE_Obj,address(IDE_Obj,'idat'),'int8',4)

idatv =

1 0 -4 1
```

c Change the values stored in `ddat` by entering
```
write(IDE_Obj,address(IDE_Obj,'ddat'),double([pi 12.3
exp(-1)...
sin(pi/4)]))
```

The `double` argument directs MATLAB software to write the values to the processor as double-precision data.

d To change `idat`, enter the following command:

```
write(IDE_Obj,address(IDE_Obj,'idat'),int32([1:4]))
```

In this command, you write the data to the processor as 32-bit integers (convenient for representing phone numbers, for example).

e Next, start the program running again by entering the following command:

```
run(IDE_Obj,'runtohalt',30);
```

The **Stdout** tab in CCS IDE reveals that `ddat` and `idat` contain new values. Read those new values back into MATLAB software.

f Enter `ddatv =`
`read(IDE_Obj,address(IDE_Obj,'ddat'),'double',4)`.

```
ddatv =

3.1416 12.3000 0.3679 0.7071
```

Verify that `ddatv` contains the values you wrote in step c.

g Verify that the change to `idatv` occurred by entering the following command:

```
idatv = read(IDE_Obj,address(IDE_Obj,'idat'),'int32',4)
```

MATLAB software returns the new values for `idatv`.

```
idatv =

1 2 3 4
```

h Use `restart` to reset the program counter for your program to the beginning. Enter the following command at the prompt:

```
restart(IDE_Obj);
```

4 Embedded Coder offers more functions for reading and writing data to your processor. These functions let you read and write data to the processor registers: `regread` and `regwrite`. They let you change variable values on the processor in real time. The functions behave slightly differently depending on your processor. Select one of the following procedures to demonstrate `regread` and `regwrite` for your processor.

C5xxx processor family — Most registers are memory-mapped and available using `read` and `write`. However, the PC register is not memory mapped. To access this register, use the special functions — `regread` and `regwrite`. The following commands demonstrate how to use these functions to read and write to the PC register.

a To read the value stored in register PC, enter the following command at the prompt to indicate to MATLAB software the data type to read. The input string `binary` indicates that the PC register contains a value stored as an unsigned binary integer.

```
IDE_Obj.regread('PC','binary')
```

MATLAB software displays

```
ans =

33824
```

b To write a new value to the PC register, enter the following command. This time, the `binary` input argument tells MATLAB software to write the value to the processor as an unsigned binary integer. Notice that you used `hex2dec` to convert the hexadecimal string to decimal.

```
IDE_Obj.regwrite('PC',hex2dec('100'),'binary')
```

c Verify that the PC register contains the value you wrote.

```
IDE_Obj.regread('PC','binary')
```

C6xxx processor family — `regread` and `regwrite` let you access the processor registers directly. Enter the following commands to get data into and out of the A0 and B2 registers on your processor.

a To retrieve the value in register A0 and store it in a variable in your MATLAB workspace. Enter the following command:

```
treg = IDE_Obj.regread('A0','2scomp');
```

`treg` contains the two's complement representation of the value in A0.

b To retrieve the value in register B2 as an unsigned binary integer, enter the following command:

```
IDE_Obj.regread('B2','binary');
```

c Next, enter the following command to use `regwrite` to put the value in `treg` into register A2.

```
IDE_Obj.regwrite('A2',treg,'2scomp');
```

CCS IDE reports that A0, B2, and A2 have the values you expect. Select **View > CPU Registers > Core Registers** from the CCS IDE menu bar to list the processor registers.

Closing the Links or Cleaning Up CCS IDE

Objects that you create in Embedded Coder software have COM handles to CCS. Until you delete these handles, the CCS process (`cc_app.exe` in the Microsoft Windows Task Manager) remains in memory. Closing MATLAB software removes these COM handles, but there may be times when you want to delete the handles without closing the application.

Use `clear` to remove objects from your MATLAB workspace and to delete handles they contain. `clear all` deletes everything in your workspace. To retain your MATLAB software data while deleting objects and handles, use `clear objname`. This applies to IDE handle objects you created with `ticcs`. To remove the objects created during the tutorial, the tutorial program executes the following command at the prompt:

```
clear cvar cfield uintcvar
```

This tutorial also closes the project in CCS with the following command:

```
close(IDE_Obj,projfile,'project')
```

To delete your link to CCS, enter `clear IDE_Obj` at the prompt.

Your development tutorial using Code Composer Studio IDE is done.

During the tutorial you

1. Selected your processor.
2. Created and queried links to CCS IDE to get information about the link and the processor.
3. Used MATLAB software to load files into CCS IDE, and used MATLAB software to run that file.
4. Worked with your CCS IDE project from MATLAB software by reading and writing data to your processor, and changing the data from MATLAB software.
5. Created and used the embedded objects to manipulate data in a C-like way.
6. Closed the links you opened to CCS IDE.

Getting Started with RTDX

- "Introducing the Tutorial for Using RTDX" on page 54-29
- "Creating the ticcs Objects" on page 54-33
- "Configuring Communications Channels" on page 54-36

- "Running the Application" on page 54-38
- "Closing the Connections and Channels or Cleaning Up" on page 54-45
- "Listing Functions" on page 54-48

Texas Instruments Real-Time Data Exchange (RTDX) provides "real-time, continuous visibility into the way target applications operate in the real world. RTDX allows system developers to transfer data between target devices and a host without interfering with the target application."

You can use RTDX with Embedded Coder software and Code Composer Studio to accelerate development and deployment to Texas Instruments C2000 processors. RTDX helps you test and analyze your processing algorithms in your MATLAB workspace. RTDX lets you interact with your process in real time while it's running on the processor. For example, you can:

- Send and retrieve data from memory on the processor
- Change the operating characteristics of the program
- Make changes to algorithms as needed without stopping the program or setting breakpoints in the code

Enabling real-time interaction lets you more easily see your process or algorithm in action, the results as they develop, and the way the process runs.

This tutorial assumes you have Texas Instruments' Code Composer Studio software and at least one target development board. You can use the hardware simulator in CCS IDE to run this tutorial.

After you complete the tutorial, either in the demonstration form or by entering the functions along with this text, you are ready to begin using RTDX with your applications and hardware.

Note To use RTDX with the XDS100 USB JTAG Emulator and the C28027 chip, add the following line to the linker command file:

```
_RTDX_interrupt_mask = ~0x000000008;
```

Introducing the Tutorial for Using RTDX

Digital signal processing development efforts begin with an idea for processing data; an application area, such as audio or wireless communications or multimedia computing; and a platform or hardware to host the signal processing. Usually these processing efforts involve applying strategies like signal filtering, compression, and transformation to change data content; or isolate features in data; or transfer data from one form to another or one place to another.

Developers create algorithms they need to accomplish the desired result. After they have the algorithms, they use models and target processor development tools to test their algorithms, to determine whether the processing achieves the goal, and whether the processing works on the proposed platform.

Embedded Coder and the links for RTDX and CCS IDE ease the job of taking algorithms from the model realm to the real world of the processor on which the algorithm runs.

RTDX and links for CCS IDE provide a communications pathway to manipulate data and processing programs on your processor. RTDX offers real-time data exchange in two directions between MATLAB software and your processor process. Data you send to the processor has little effect on the running process and plotting the data you retrieve from the processor lets you see how your algorithms are performing in real time.

To introduce the techniques and tools available in Embedded Coder for using RTDX, the following procedures use many of the methods in the link software to configure the processor, open and enable channels, send data to the processor, and clean up after you finish your testing. Among the functions covered are:

Functions From Objects for CCS IDE.

Function	Description
`ticcs`	Create connections to CCS IDE and RTDX.
`cd`	Change the CCS IDE working folder from MATLAB software.

Function	Description
`open`	Load program files in CCS IDE.
`run`	Run processes on the processor.

Functions From the RTDX Class.

Function	Description
`close`	Close the RTDX links between MATLAB software and your processor.
`configure`	Determine how many channel buffers to use and set the size of each buffer.
`disable`	Disable the RTDX links before you close them.
`display`	Return the properties of an object in formatted layout. When you omit the closing semicolon on a function, `disp` (a built-in function) provides the default display for the results of the operation.
`enable`	Enable open channels so you can use them to send and retrieve data from your processor.
`isenabled`	Determine whether channels are enabled for RTDX communications.
`isreadable`	Determine whether MATLAB software can read the specified memory location.
`iswritable`	Determine whether MATLAB software can write to the processor.
`msgcount`	Determine how many messages are waiting in a channel queue.

Function	Description
open	Open channels in RTDX.
readmat	Read data matrices from the processor into MATLAB software as an array.
readmsg	Read one or more messages from a channel.
writemsg	Write messages to the processor over a channel.

This tutorial provides the following workflow to show you how to use many of the functions in the links. By performing the steps provided, you work through many of the operations yourself. The tutorial follows the general task flow for developing digital signal processing programs through testing with the links for RTDX.

Within this set of tasks, numbers 1, 2, and 4 are fundamental to all development projects. Whenever you work with MATLAB software and objects for RTDX, you perform the functions and tasks outlined and presented in this tutorial. The differences lie in Task 3. Task 3 is the most important for using Embedded Coder to develop your processing system.

1. Create an RTDX link to your desired processor and load the program to the processor.

   All projects begin this way. Without the links you cannot load your executable to the processor.

2. Configure channels to communicate with the processor.

   Creating the links in Task 1 did not open communications to the processor. With the links in place, you open as many channels as you need to support the data transfer for your development work. This task includes configuring channel buffers to hold data when the data rate from the processor exceeds the rate at which MATLAB software can capture the data.

3. Run your application on the processor. You use MATLAB software to investigate the results of your running process.

4. Close the links to the processor and clean up the links and associated debris left over from your work.

   Closing channels and cleaning up the memory and links you created ensures that CCS IDE, RTDX, and Embedded Coder are ready for the next time you start development on a project.

This tutorial uses an executable program named `rtdxtutorial_6xevm.out` as your application. When you use the RTDX and CCS IDE links to develop your own applications, replace `rtdxtutorial_6xevm.out` in Task 3 with the filename and path to your digital signal processing application.

You can view the tutorial file used here by clicking `rtdxtutorial`. To run this tutorial in MATLAB software, click `run rtdxtutorial`.

Note To be able to open and enable channels over a link to RTDX, the program loaded on your processor must include functions or code that define the channels.

Your C source code might look something like this to create two channels, one to write and one to read.

```
rtdx_CreateInputChannel(ichan); % processor reads from this.
rtdx_CreateOutputChannel(ochan); % processor writes to this.
```

These are the entries we use in `int16.c` (the source code that generates `rtdxtutorial_6xevm.out`) to create the read and write channels.

If you are working with a model in Simulink software and using code generation, use the To Rtdx and From Rtdx blocks in your model to add the RTDX communications channels to your model and to the executable code on your processor.

One more note about this tutorial. Throughout the code we use both the dot notation (direct property referencing) to access functions and link properties and the function form.

For example, use the following command to open and configure `ichan` for write mode.

```
IDE_Obj.rtdx.open('ichan','w');
```

You could use an equivalent syntax, the function form, that does not use direct property referencing.

```
open(IDE_Obj.rtdx,'ichan','w');
```

Or, use

```
open(rx,'ichan','w');
```

if you created an alias `rx` to the RTDX portion of `IDE_Obj`, as shown by the following command:

```
rx = IDE_Obj.rtdx;
```

Creating the ticcs Objects

With your processing model converted to an executable suitable for your desired processor, you are ready to use the objects to test and run your model on your processor. Embedded Coder and the objects do not distinguish the source of the executable — whether you used Embedded Coder, CCS IDE, or some other development tool to program and compile your model to an executable does not affect the object connections. So long as your `..out` file is acceptable to the processor you select, Embedded Coder provides the connection to the processor.

Before continuing with this tutorial, you must load a valid GEL file to configure the EMIF registers of your processor and perform any required processor initialization steps. Default GEL files provided by CCS are stored in `..\IDE_Obj\gel` in the folder where you installed CCS software. Select **File > Load_GEL** in CCS IDE to load the default GEL file that matches your processor family, such as `init6x0x.gel` for the Cxxxx processor family, and your configuration.

Note If you are performing the steps in this tutorial, create demoPjt as described in "Loading Files into CCS" on page 54-18 before continuing.

Begin the process of getting your model onto the processor by creating a an object that refers to CCS IDE. Start by clearing all existing handles and setting echo on so you see functions execute as the program runs:

1. `clear all; echo on;`

   `clear all` has the side effect of removing debugging breakpoints and resetting persistent variables because function breakpoints and persistent variables are cleared whenever the MATLAB file changes or is cleared. Breakpoints within your executable remain after `clear`. Clearing the MATLAB workspace does not affect your executable.

2. Now construct the link to your board and processor by entering

   ```
   IDE_Obj=ticcs('boardnum',0);
   ```

   `boardnum` defines which board the new link accesses. In this example, `boardnum` is 0. Embedded Coder connects the link to the first, and in this case only, processor on the board. To find the `boardnum` and `procnum` values for the boards and simulators on your system, use ccsboardinfo. When you enter the following command at the prompt

   ```
   ccsboardinfo
   ```

3. To open and load the processor file, change the path for MATLAB software to be able to find the file.

   ```
   projname = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp\rtdxtut_sim.pjt


   outFile = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp\rtdxtut_sim.out

   processor_dir = demoPjt.DemoDir

   processor_dir = C:\Temp\EmbIDELinkCCDemos_v4.1\rtdxtutorial\cxx\cxxxp
   ```

```
cd(IDE_Obj,processor_dir); % Go to processor directory
cd(IDE_Obj,tgt_dir); % Or IDE_Obj.cd(tgt_dir)
dir(IDE_Obj); % Or IDE_Obj.dir
```

To load the appropriate project file to your processor, enter the following commands at the MATLAB software prompt. `getDemoProject` is a specialized function for loading Embedded Coder demo files. It is not supported as a standard Embedded Coder function.

```
demoPjt = getDemoProject(IDE_Obj,'ccstutorial');

demoPjt.ProjectFile

ans = C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxxp\ccstut.pjt


demoPjt.DemoDir

ans = C:\Temp\EmbIDELinkCCDemos_v4.1\ccstutorial\cxx\cxxxp
```

Notice where the demo files are stored on your machine. In general, Embedded Coder software stores the demo project files in

```
EmbIDELinkCCDemos_v#.#
```

For example, if you are using version 4.1 of Embedded Coder software, the project demos are stored in `EmbIDELinkCCDemos_v4.1\`. Embedded Coder software creates this folder in a location on your machine where you have write permission. Usually, there are two locations where Embedded Coder software tries to create the demo folder, in the order shown.

a In a temporary folder on the C drive, such as `C:\temp\`.

b If Embedded Coder software cannot use the `temp` folder, you see a dialog box that asks you to select a location to store the demos.

4 You have reset the folder path to find the tutorial file. Now open the `.out` file that matches your processor type.

```
IDE_Obj.open('rtdxtutorial_xxx.out')
```

Because open is overloaded for the CCS IDE and RTDX links, this may seem a bit strange. In this syntax, open loads your executable file onto the processor identified by IDE_Obj. Later in this tutorial, you use open with a different syntax to open channels in RTDX.

In the next section, you use the new link to open and enable communications between MATLAB software and your processor.

Configuring Communications Channels

Communications channels to the processor do not exist until you open and enable them through Embedded Coder and CCS IDE. Opening channels consists of opening and configuring each channel for reading or writing, and enabling the channels.

In the open function, you provide the channel names as strings for the channel name property. The channel name you use is not random. The channel name string must match a channel defined in the executable file. If you specify a string that does not identify an existing channel in the executable, the open operation fails.

In this tutorial, two channels exist on the processor — ichan and ochan. Although the channels are named ichan for input channel and ochan for output channel, neither channel is configured for input or output until you configure them from MATLAB software or CCS IDE. You could configure ichan as the output channel and ochan as the input channel. The links would work just the same. For simplicity, the tutorial configures ichan for input and ochan for output. One more note—reading and writing are defined as seen by the processor. When you write data from MATLAB software, you write to the channel that the processor reads, ichan in this case. Conversely, when you read from the processor, you read from ochan, the channel that the processor writes to:

1 Configure buffers in RTDX to store the data until MATLAB software can read it into your workspace. Often, MATLAB software cannot read data as quickly as the processor can write it to the channel.

```
IDE_Obj.rtdx.configure(1024,4); % define 4 channels of 1024 bytes each
```

Channel buffers are optional. Adding them provides a measure of insurance that data gets from your processor to MATLAB software without getting lost.

2 Define one of the channels as a write channel. Use 'ichan' for the channel name and 'w' for the mode. Either 'w' or 'r' fits here, for write or read.

```
IDE_Obj.rtdx.open('ichan','w');
```

3 Now enable the channel you opened.

```
IDE_Obj.rtdx.enable('ichan');
```

4 Repeat steps 2 and 3 to prepare a read channel.

```
IDE_Obj.rtdx.open('ochan','r');
IDE_Obj.rtdx.enable('ochan');
```

5 To use the new channels, enable RTDX by entering

```
IDE_Obj.rtdx.enable;
```

You could do this step before you configure the channels — the order does not matter.

6 Reset the global time-out to 20s to provide a little room for error. `ticcs` applies a default timeout value of 10s. In some cases this may not be enough.

```
IDE_Obj.rtdx.get('timeout')
ans =
     10
IDE_Obj.rtdx.set('timeout', 20); % Reset timeout = 20 seconds
```

7 Check that the `timeout` property value is now 20s and that your object has the correct configuration for the rest of the tutorial.

```
IDE_Obj.rtdx

RTDX Object:
  API version:       1.0
```

```
Default timeout:   20.00 secs
Open channels:     2
```

Running the Application

To this point you have been doing housekeeping functions that are common to any application you run on the processor. You load the processor, configure the communications, and set up other properties you need.

In this tutorial task, you use a specific application to demonstrate a few of the functions available in Embedded Coder that let you experiment with your application while you develop your prototype. To demonstrate the RTDX readmat, `readmsg`, and `writemsg` functions, you write data to your processor for processing, then read data from the processor after processing:

1. Restart the program you loaded on the processor. `restart` ensures the program counter (PC) is at the beginning of the executable code on the processor.

   ```
   IDE_Obj.restart
   ```

   Restarting the processor does not start the program executing. You use `run` to start program execution.

2. Type `IDE_Obj.run('run');`

   Using '`run`' for the `run` mode tells the processor to continue to execute the loaded program continuously until it receives a halt directive. In this mode, control returns to MATLAB software so you can work in MATLAB software while the program runs. Other options for the mode are

   - '`runtohalt`' — start to execute the program and wait to return control to MATLAB software until the process reaches a breakpoint or execution terminates.
   - '`tohalt`' — change the state of a running processor to '`runtohalt`' and wait to return until the program halts. Use `tohalt` mode to stop the running processor cleanly.

3. Type the following functions to enable the write channel and verify that the enable takes effect.

```
IDE_Obj.rtdx.enable('ichan');
IDE_Obj.rtdx.isenabled('ichan')
```

If MATLAB software responds `ans = 0` your channel is not enabled and you cannot proceed with the tutorial. Try to enable the channel again and verify the status.

4 Write some data to the processor. Check that you can write to the processor, then use `writemsg` to send the data. You do not need to enter the if-test code shown.

```
if IDE_Obj.rtdx.iswritable('ichan'),  % Used in a script application.
    disp('writing to processor...') % Optional to display progress.
    indata=1:10
    IDE_Obj.rtdx.writemsg('ichan', int16(indata))
end  % Used in scripts for channel testing.
```

The `if` statement simulates writing the data from within a MATLAB software script. The script uses `iswritable` to check that the input channel is functioning. If `iswritable` returns `0` the script would skip the write and exit the program, or respond in some way. When you are writing or reading data to your processor in a script or MATLAB file, checking the status of the channels can help you avoid errors during execution.

As your application runs you may find it helpful to display progress messages. In this case, the program directed MATLAB software to print a message as it reads the data from the processor by adding the function

```
disp('writing to processor...')
```

Function `IDE_Obj.rtdx.writemsg('ichan', int16(indata))` results in 20 messages stored on the processor. Here's how.

When you write `indata` to the processor, the following code running on the processor takes your input data from `ichan`, adds one to the values and copies the data to memory:

```
while ( !RTDX_isInputEnabled(&ichan) )

{/* wait for channel enable from MATLAB */}
RTDX_read( &ichan, recvd, sizeof(recvd) );
```

```
puts("\n\n Read Completed ");

for (j=1; j<=20; j++) {
  for (i=0; i<MAX; i++) {
    recvd[i] +=1;
  }
  while ( !RTDX_isOutputEnabled(&ochan) )
    { /* wait for channel enable from MATLAB */ }
  RTDX_write( &ochan, recvd, sizeof(recvd) );
   while ( RTDX_writing != NULL )
   { /* wait for data xfer INTERRUPT DRIVEN for Cxxxx */ }
}
```

Program `int16_rtdx.c` contains this source code. You can find the file in a folder in the `..\tidemos\rtdxtutorial` folder.

5. Type the following to check the number of available messages to read from the processor.

   ```
   num_of_msgs = IDE_Obj.rtdx.msgcount('ochan');
   ```

   `num_of_msgs` should be zero. Using this process to check the amount of data can make your reads more reliable by letting you or your program know how much data to expect.

6. Type the following to verify that your read channel `ochan` is enabled for communications.

   ```
   IDE_Obj.rtdx.isenabled('ochan')
   ```

   You should get back `ans = 0` — you have not enabled the channel yet.

7. Now enable and verify '`ochan`'.

   ```
   IDE_Obj.rtdx.enable('ochan');
   IDE_Obj.rtdx.isenabled('ochan')
   ```

   To show that `ochan` is ready, MATLAB software responds `ans = 1`. If not, try enabling `ochan` again.

8. Type

```
pause(5);
```

The `pause` function gives the processor extra time to process the data in `indata` and transfer the data to the buffer you configured for `ochan`.

9 Repeat the check for the number of messages in the queue. There should be 20 messages available in the buffer.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')
```

With `num_of_msgs = 20`, you could use a looping structure to read the messages from the queue in to MATLAB software. In the next few steps of this tutorial you read data from the `ochan` queue to different data formats within MATLAB software.

10 Read one message from the queue into variable `outdata`.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16')

outdata =
     2     3     4     5     6     7     8     9     10     11
```

Notice the 'int16' `represent` option. When you read data from your processor you need to tell MATLAB software the data type you are reading. You wrote the data in step 4 as 16-bit integers so you use the same data type here.

While performing reads and writes, your process continues to run. You did not need to stop the processor to get the data or send the data, unlike using most debuggers and breakpoints in your code. You placed your data in memory across an RTDX channel, the processor used the data, and you read the data from memory across an RTDX channel, without stopping the processor.

11 You can read data into cell arrays, rather than into simple double-precision variables. Use the following function to read three messages to cell array `outdata`, an array of three, 1-by-10 vectors. Each message is a 1-by-10 vector stored on the processor.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',3)
```

```
outdata =
[1x10  int16]  [1x10  int16]  [1x10  int16]
```

12 Cell array `outdata` contains three messages. Look at the second message, or matrix, in `outdata` by using dereferencing with the array.

```
outdata{1,2}

outdata =
     4     5     6     7     8     9    10    11    12    13
```

13 Read two messages from the processor into two 2-by-5 matrices in your MATLAB workspace.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',[2 5],2)

outdata =
    [2x5 int16]  [2x5 int16]
```

To specify the number of messages to read and the data format in your workspace, you used the `siz` and `nummsgs` options set to `[2 5]` and `2`.

14 You can look at both matrices in `outdata` by dereferencing the cell array again.

```
outdata{1,:}

ans =
     6     8    10    12    14
     7     9    11    13    15
ans =
     7     9    11    13    15
     8    10    12    14    16
```

15 For a change, read a message from the queue into a column vector.

```
outdata = IDE_Obj.rtdx.readmsg('ochan','int16',[10 1])

outdata =
      8
      9
```

```
10
11
12
13
14
15
16
17
```

16 Embedded Coder provides a function for reading messages into matrices–readmat. Use `readmat` to read a message into a 5-by-2 matrix in MATLAB software.

```
outdata = readmat(IDE_Obj.rtdx,'ochan','int16',[5 2])

outdata =
     9   14
    10   15
    11   16
    12   17
    13   18
```

Because a 5-by-2 matrix requires ten elements, MATLAB software reads one message into `outdata` to fill the matrix.

17 To check your progress, see how many messages remain in the queue. You have read eight messages from the queue so 12 should remain.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')

num_of_msgs =
     12
```

18 To demonstrate the connection between messages and a matrix in MATLAB software, read data from `'ochan'` to fill a 4-by-5 matrix in your workspace.

```
outdata = IDE_Obj.rtdx.readmat('ochan','int16',[4 5])

outdata =
    10   14   18   13   17
    11   15   19   14   18
```

```
    12    16    11    15    19
    13    17    12    16    20
```

Filling the matrix required two messages worth of data.

19 To verify that the last step used two messages, recheck the message count. You should find 10 messages waiting in the queue.

```
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')
```

20 Continuing with matrix reads, fill a 10-by-5 matrix (50 matrix elements or five messages).

```
outdata = IDE_Obj.rtdx.readmat('ochan','int16',[10 5])

outdata =
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    14    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
    19    20    21    22    23
    20    21    22    23    24
    21    22    23    24    25
```

21 Recheck the number of messages in the queue to see that five remain.

22 flush lets you remove messages from the queue without reading them. Data in the message you remove is lost. Use flush to remove the next message in the read queue. Then check the waiting message count.

```
IDE_Obj.rtdx.flush('ochan',1)
num_of_msgs = IDE_Obj.rtdx.msgcount('ochan')

num_of_msgs =

     4
```

23 Empty the remaining messages from the queue and verify that the queue is empty.

```
IDE_Obj.rtdx.flush('ochan','all')
```

With the `all` option, `flush` discards all messages in the `ochan` queue.

Closing the Connections and Channels or Cleaning Up

One of the most important programmatic processes you should do in every RTDX session is to clean up at the end. Cleaning up includes stopping your processor, disabling the RTDX channels you enabled, disabling RTDX and closing your open channels. Performing this series of tasks ensures that future processes avoid trouble caused by unexpected interactions with remaining handles, channels, and links from earlier development work.

Best practices suggest that you include the following tasks (or an appropriate subset that meets your development needs) in your development scripts and programs.

We use several functions in this section; each has a purpose — the operational details in the following list explain how and why we use each one. They are

- `close` — close the specified RTDX channel. To use the channel again, you must open and enable the channel. Compare `close` to `disable`. `close('rtdx')` closes the communications provided by RTDX. After you close RTDX, you cannot communicate with your processor.
- `disable` — remove RTDX communications from the specified channel, but does not remove the channel, or link. Disabling channels may be useful when you do not want to see the data that is being fed to the channel, but you may want to read the channel later. By enabling the channel later, you have access to the data entering the channel buffer. Note that data that entered the channel while it was disabled is lost.
- `halt` — stop a running processor. You may still have one or more messages in the host buffer.

Use the following procedure to shut down communications between MATLAB software and the processor, and end your session:

1 Begin the process of shutting down the processor and RTDX by stopping the processor. Type the following functions at the prompt.

```
if (isrunning(IDE_Obj))  % Use this test in scripts.
   IDE_Obj.halt;         % Halt the processor.
end                  % Done.
```

Your processor may already be stopped at this point. In a script, you might put the function in an `if`-statement as we have done here. Consider this test to be a safety check. No harm comes to the processor if it is already stopped when you tell it to stop. When you direct a stopped processor to halt, the function returns immediately.

2 You have stopped the processor. Now disable the RTDX channels you opened to communicate with the processor.

```
IDE_Obj.rtdx.disable('all');
```

If necessary, using `disable` with channel name and processor identifier input arguments lets you disable only the channel you choose. When you have more than one board or processor, you may find disabling selected channels meets your needs.

When you finish your RTDX communications session, disable RTDX to ensure that Embedded Coder releases your open channels before you close them.

```
IDE_Obj.rtdx.disable;
```

3 Use one or all of the following function syntaxes to close your open channels. Either close selected channels by using the channel name in the function, or use the `all` option to close all open channels.

- `IDE_Obj.rtdx.close('ichan')` to close your input channel in this tutorial.
- `IDE_Obj.rtdx.close('ochan')` to close your output channel in the tutorial.
- `IDE_Obj.rtdx.close('all')` to close all of your open RTDX channels, regardless of whether they are part of this tutorial.

Consider using the `all` option with the `close` function when you finish your RTDX work. Closing channels reduces unforeseen problems caused by channel objects that exist but do not get closed correctly when you end your session.

4. When you created your RTDX object (`IDE_Obj = ticcs('boardnum',1)`) at the beginning of this tutorial, the `ticcs` function opened CCS IDE and set the visibility to 0. To avoid problems that occur when you close the interface to RTDX with CCS visibility set to 0, make CCS IDE visible on your desktop. The following `if` statement checks the CCS IDE visibility and changes it if needed.

   ```
   if IDE_Obj.isvisible,
       IDE_Obj.visible(1);
   end
   ```

   Visibility can cause problems. When CCS IDE is running invisibly on your desktop, do not use `clear all` to remove your links for CCS IDE and RTDX. Without a `ticcs` object that references the CCS IDE you cannot access CCS IDE to change the visibility setting, or close the application. To close CCS IDE when you do not have an existing object, either create a new object to access the CCS IDE, or use Microsoft Windows Task Manager to end the process `cc_app.exe`, or close the MATLAB software.

5. You have finished the work in this tutorial. Enter the following commands to close your remaining references to CCS IDE and release the associated resources.

   ```
   clear ('all'); % Calls the link destructors to remove all links.
   echo off
   ```

   `clear all` without the parentheses removes all variables from your MATLAB workspace.

You have completed the tutorial using RTDX. During the tutorial you

1. Opened connections to CCS IDE and RTDX and used those connections to load an executable program to your processor.

2. Configured a pair of channels so you could transfer data to and from your processor.

3 Ran the executable on the processor, sending data to the processor for processing and retrieving the results.

4 Stopped the executing program and closed the links to CCS IDE and RTDX.

This tutorial provides a working process for using Embedded Coder and your signal processing programs to develop programs for a range of Texas Instruments processors. While the processor may change, the essentials of the process remain the same.

Listing Functions

To review a complete list of functions and methods that operate with `ticcs` objects, either CCS IDE or RTDX, enter either of the following commands at the prompt.

```
help ticcs
help rtdx
```

If you already have a `ticcs` object `IDE_Obj`, you can use dot notation to return the methods for CCS IDE or RTDX by entering one of the following commands at the prompt:

- `IDE_Obj.methods`
- `IDE_Obj.rtdx.methods`

In either instance MATLAB software returns a list of the available functions for the specified link type, including both public and private functions. For example, to see the functions (methods) for links to CCS IDE, enter:

```
help ticcs
```

Constructing ticcs Objects

When you create a connection to CCS IDE using the ticcs command, you are creating a “`ticcs` object for accessing the CCS IDE and RTDX interface”. The `ticcs` object implementation relies on MATLAB software object-oriented programming capabilities.

The discussions in this section apply to the `ticcs` objects in Embedded Coder.

Like other MATLAB software structures, objects in Embedded Coder have predefined fields called object properties.

You specify object property values by one of the following methods:

- Setting the property values when you create the `ticcs` object
- Creating an object with default property values, and changing some or all of these property values later

For examples of setting ticcs object properties, refer to `ticcs`.

Example — Constructor for ticcs Objects

The easiest way to create an object is to use the function `ticcs` to create an object with the default properties. Create an object named `IDE_Obj` to refer to CCS IDE by entering

```
IDE_Obj = ticcs
```

MATLAB software responds with a list of the properties of the object `IDE_Obj` you created along with the associated default property values.

```
ticcs object:
  API version      : 1.0
  Processor type   : Cxx
  Processor name   : CPU
  Running?         : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Inspecting the output reveals two objects listed—a CCS IDE object and an RTDX object. CCS IDE and RTDX objects cannot be created separately. By design they maintain a member class relationship; the RTDX object is a class, a member of the CCS object class. In this example, `IDE_Obj` is an instance of the class CCS. If you enter

```
rx = IDE_Obj.rtdx
```

`rx` is a handle to the RTDX portion of the CCS object. As an alias, `rx` replaces `IDE_Obj.rtdx` in functions such as `readmat` or `writemsg` that use the RTDX communications features of the CCS link. Typing `rx` at the command line now produces

```
rx

RTDX channels    : 0
```

The object properties are described in the function reference, and in more detail in ticcs Object Properties. These properties are set to default values when you construct objects.

ticcs Properties and Property Values

Objects in Embedded Coder software have properties associated with them. Each property is assigned a value. You can set the values of most properties, either when you create the link or by changing the property value later. However, some properties have read-only values. And a few property values, such as the board number and the processor to which the link attaches, become read-only after you create the object. You cannot change those after you create your link.

For more information about using objects and properties, refer to “Using Objects” in *MATLAB Programming Fundamentals*.

Overloaded Functions for ticcs Objects

Several functions in this Embedded Coder have the same name as functions in other MathWorks toolboxes or in MATLAB software. These behave similarly to their original counterparts, but you apply these functions directly to an object. This concept of having functions with the same name operate on different types of objects (or on data) is called *overloading* of functions.

For example, the `set` command is overloaded for `ticcs` objects. After you specify your link by assigning values to its properties, you can apply the functions in this toolbox (such as readmat for using RTDX to read an array of data from the processor) directly to the variable name you assign to your object, without specifying your object parameters again.

ticcs Object Properties

- "Quick Reference to ticcs Object Properties" on page 54-51
- "Details About ticcs Object Properties" on page 54-52

Embedded Coder provides an interface to your processor hardware so you can communicate with processors for which you are developing systems and algorithms. Each ticcs object comprises two objects—a CCS IDE object and an RTDX interface object. The objects are not separable; the RTDX object is a subclass of the CCS IDE object. Each of the objects has multiple properties. To configure the interface objects for CCS IDE and RTDX, you set parameters that define details such as the desired board, the processor, the timeout period applied for communications operations, and a number of other values. Because Embedded Coder uses objects to create the interface, the parameters you set are called properties and you treat them as properties when you set them, retrieve them, or modify them.

This section details the properties for the `ticcs` objects for CCS IDE and RTDX. First the section provides tables of the properties, for quick reference. Following the tables, the section offers in-depth descriptions of each property, its name and use, and whether you can set and get the property value associated with the property. Descriptions include a few examples of the property in use.

MATLAB software users may find much of this handling of objects familiar. Objects in Embedded Coder, behave like objects in MATLAB software and the other object-oriented toolboxes. For C++ programmers, discussion of object-oriented programming is likely to be a review.

Quick Reference to ticcs Object Properties

The following table lists the properties for the `ticcs` objects in Embedded Coder. The second column tells you which object the property belongs to. Knowing which property belongs to each object in a `ticcs` object tells you how to access the property.

Property Name	Applies to Which Connection?	User Settable?	Description
apiversion	CCS IDE	No	Reports the version number of your CCS API.
boardnum	CCS IDE	Yes/initially	Specifies the index number of a board that CCS IDE recognizes.
ccsappexe	CCS IDE	No	Specifies the path to the CCS IDE executable. Read-only property.
numchannels	RTDX	No	Contains the number of open RTDX channels for a specific link.
page	CCS IDE	Yes/default	Stores the default memory page for reads and writes.
procnum	CCS IDE	Yes/at start only	Stores the number CCS Setup Utility assigns to the processor.
timeout	CCS IDE	Yes/default	Contains the global timeout setting for the link.
version	RTDX	No	Reports the version of your RTDX software.

Some properties are read only — you cannot set the property value. Other properties you can change at all times. If the entry in the User Settable column is “Yes/initially”, you can set the property value only when you create the link. Thereafter it is read only.

Details About ticcs Object Properties

To use the links for CCS IDE and RTDX interface you set values for:

- `boardnum` — specify the board with which the link communicates.

- `procnum` — specify the processor on the board. If the board has multiple processors, `procnum` identifies the processor to use.
- `timeout` — specify the global timeout value. (Optional. Default is 10s.)

Details of the properties associated with connections to CCS IDE and RTDX interface appear in the following sections, listed in alphabetical order by property name.

Many of these properties are object linking and embedding (OLE) handles. The MATLAB software COM server creates the handles when you create objects for CCS IDE and RTDX. You can manipulate the OLE handles using `get`, `set`, and `invoke` to work directly with the COM interface with which the handles interact.

apiversion. Property `appversion` contains a string that reports the version of the application program interface (API) for CCS IDE that you are using when you create a link. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `apiversion` property value for a link. This example shows the `appversion` value for link `IDE_Obj`.

```
display(IDE_Obj)

TICCS Object:
  API version      : 1.0
  Processor type   : Cxx
  Processor name   : CPU
  Running?         : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0
```

Note that the API version is not the same as the CCS IDE version.

boardnum. Property `boardnum` identifies the board referenced by the IDE handle object for CCS. When you create a link, you use `boardnum` to specify the board you are using. To get the value for `boardnum`, use ccsboardinfo or the CCS Setup utility from Texas Instruments software. The CCS Setup utility assigns the number for each board installed on your system.

ccsappexe. Property `ccsappexe` contains the path to the CCS IDE executable file `cc_app.exe`. When you use `ticcs` to create a link, MATLAB software determines the path to the CCS IDE executable and stores the path in this property. This is a read-only property. You cannot set it.

numchannels. Property `numchannels` reports the number of open RTDX communications channels for an RTDX link. Each time you open a channel for a link, `numchannels` increments by one. For new links `numchannels` is zero until you open a channel for the link.

To see the value for numchannels create a link to CCS IDE. Then open a channel to RTDX. Use `display` to see the RTDX link properties.

```
IDE_Obj=ticcs

TICCS Object:
  API version      : 1.0
  Processor type   : Cxx
  Processor name   : CPU
  Running?         : No
  Board number     : 0
  Processor number : 0
  Default timeout  : 10.00 secs

  RTDX channels    : 0

rx=IDE_Obj.rtdx

  RTDX channels    : 0

open(rx,'ichan','r','ochan','w');

get(IDE_Obj.rtdx)
```

```
ans =

    numChannels: 2
           Rtdx: [1x1 COM ]
    RtdxChannel: {''  ''  ''}
       procType: 103
        timeout: 10
```

page. Property `page` contains the default value CCS IDE uses when the user does not specify the page input argument in the syntax for a function that access memory.

procnum. Property `procnum` identifies the processor referenced by the IDE handle object for CCS. When you create an object, you use `procnum` to specify the processor you are using . The CCS Setup Utility assigns a number to each processor installed on each board. To determine the value of `procnum` for a processor, use ccsboardinfo or the CCS Setup utility from Texas Instruments software.

To identify a processor, you need both the `boardnum` and `procnum` values. For boards with one processor, `procnum` equals zero. CCS IDE numbers the processors on multiprocessor boards sequentially from 0 to the number of processors. For example, on a board with four processors, the processors are numbered 0, 1, 2, and 3.

rtdx. Property `rtdx` is a subclass of the `ticcs` link and represents the RTDX portion of the IDE handle object for CCS. As shown in the example, `rtdx` has properties of its own that you can set, such as `timeout`, and that report various states of the link.

```
get(IDE_Obj.rtdx)

ans =

        version: 1
    numChannels: 0
           Rtdx: [1x1 COM ]
    RtdxChannel: {''  []  ''}
       procType: 103
        timeout: 10
```

In addition, you can create an alias to the rtdx portion of a link, as shown in this code example.

```
rx=IDE_Obj.rtdx

  RTDX channels    : 0
```

Now you can use rx with the functions in Embedded Coder, such as get or set. If you have two open channels, the display looks like the following example:

```
get(rx)

ans =

    numChannels: 2
           Rtdx: [1x1 COM ]
    RtdxChannel: {2x3 cell}
       procType: 98
        timeout: 10
```

rtdxchannel. Property rtdxchannel, along with numchannels and proctype, is a read-only property for the RTDX portion of the IDE handle object for CCS. To see the value of this property, use get with the link. Neither set nor invoke work with rtdxchannel.

rtdxchannel is a cell array that contains the channel name, handle, and mode for each open channel for the link. For each open channel, rtdxchannel contains three fields, as follows:

.rtdxchannel{i,1}	Channel name of the ith-channel, i from 1 to the number of open channels
.rtdxchannel{i,2}	Handle for the ith-channel
.rtdxchannel{i,3}	Mode of the ith-channel, either 'r' for read or 'w' for write

With four open channels, rtdxchannel contains four channel elements and three fields for each channel element.

timeout. Property `timeout` specifies how long CCS IDE waits for any process to finish. Two `timeout` periods can exist — one global, one local. You set the global `timeout` when you create the IDE handle object for CCS. The default global `timeout` value 10 s. However, when you use functions to read or write data to CCS IDE or your processor, you can set a local `timeout` that overrides the global value. If you do not set a specific `timeout` value in a read or write process syntax, the global timeout value applies to the operation. Refer to the help for the read and write functions for the syntax to set the local `timeout` value for an operation.

version. Property `version` reports the version number of your RTDX software. When you create a `ticcs` object, `version` contains a string that reports the version of the RTDX application that you are using. You cannot change this string. When you upgrade the API, or CCS IDE, the string changes to match. Use `display` to see the `version` property value for a link. This example shows the `apiversion` value for object `rx`.

```
get(rx) % rx is an alias for IDE_Obj.rtdx.

ans =

        version: 1
    numChannels: 0
           Rtdx: [1x1 COM ]
    RtdxChannel: {''  []  ''}
       procType: 103
        timeout: 10
```

Project Generator

In this section...

"Introducing Project Generator" on page 54-58
"Project Generation and Board Selection" on page 54-58
"Project Generator Tutorial" on page 54-60
"Model Reference" on page 54-65

Introducing Project Generator

Project generator provides the following features for developing project and generating code:

- Support automated project building for Texas Instruments' Code Composer Studio software that lets you create projects from code generated by Embedded Coder products. The project automatically populates CCS projects in the CCS development environment.
- Configure code generation using model configuration parameters and processor preferences block options
- Select from two system target files to generate code specific to your processor
- Configure project build process
- Automatically download and run your generated projects on your processor

---

Note You cannot generate code for C6000 processors in big-endian mode. Code generation supports only little-endian processor data byte order.

Project Generation and Board Selection

Project Generator uses `ticcs` objects to connect to the IDE. Each time you build a model to generate a project, the build process starts by issuing the ticcs method, as shown here:

```
IDE_Obj=ticcs('boardnum',boardnum,'procnum',procnum)
```

The software attempts to connect to the board (`boardnum`) and processor (`procnum`) associated with the **Board name** and **Processor number** parameters in the Target Preferences block in the model.

The result of the `ticcs` method changes, depending on the boards you configured in CCS. The following table describes how the software selects the board to connect to in your board configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio or Embedded Coder software not installed.	Returns an error message asking you to verify that you installed both Code Composer Studio and Embedded Coder properly.
Code Composer Studio software does not have any configured boards.	Returns an error message that the software could not find any boards in your configuration. Use Setup Code Composer Studio™ to configure at least one board.
Code Composer Studio software has one configured board.	Attaches to the board regardless of the name of the board supplied in the Target Preferences block. You see a warning message telling you which board the software selected.
Code Composer Studio software has one board configured that does not match the board name in the Target Preferences block.(*)	Returns a warning message that the software could not find the board specified in the block and connected to the board listed in the warning message. The software connects to the first board in your CCS configuration.

CCS Board Configuration State	Response by Software
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is one of the configured boards.	Connects to the specified board.
Code Composer Studio has more than one board configured. The board name specified in the Target Preferences block is not one of the configured boards.(*)	Returns a message asking you to select a board from the list of configured boards. You have two choices: • Select a board to use for project generation, and click **OK**. Your selection does not change the board specified in the Target Preferences block. The software connects to the selected board. • Click **Abort** to stop the project build and code generation process. The software does not connect to the IDE or board.

(*)You may encounter the situation where you do not have the correct board configured in CCS because of one of the following conditions:

- You changed your board configuration after you added the Target Preferences block to a model and saved the model. When you reopen the model, the board specified in **Board name** in the block is no longer in your configuration.
- You are working with a model from a source whose board configuration is not the same as yours. The model includes a Target Preferences block.

Use ccsboardinfo at the MATLAB prompt to verify or review your configured boards.

Project Generator Tutorial

- "Creating the Model" on page 54-61

- “Adding the Target Preferences Block to Your Model” on page 54-62
- “Specify Configuration Parameters for Your Model” on page 54-62

In this tutorial you will use the Embedded Coder software to:

- Build a model.
- Generate a project from the model.
- Build the project and run the binary on a processor.

Note The model demonstrates project generation. You cannot not build and run the model on your processor without additional blocks.

To generate a project from a model, complete the following tasks:

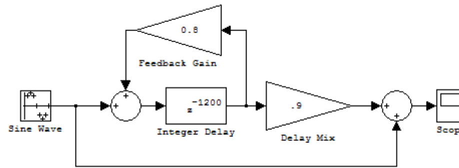
1. Create a model application.
2. Add a Target Preferences block to your model as described in “Target Preferences” on page 43-4.
3. Set the configuration parameters for your model, including
   - Solver parameters such as simulation start and solver options
   - Software options such as processor configuration and processor compiler selection
4. Generate your project.
5. Review your project in CCS.

Creating the Model

To create the model for audio reverberation, follow these steps:

1. Start Simulink software.
2. Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.

3 Use Simulink blocks and DSP System Toolbox blocks to create the following model.



Look for the Integer Delay block in the Discrete library of Simulink blocks and the Gain block in the Commonly Used Blocks library. Do not add the Target Preferences block at this time.

4 Save your model with a suitable name before continuing.

Adding the Target Preferences Block to Your Model

Add and configure a Target Preferences block to your model as described in "Target Preferences" on page 43-4.

You have completed the model. Now configure the model configuration parameters to generate a project in CCS IDE from your model.

Specify Configuration Parameters for Your Model

The following sections describe how to configure the build and run parameters for your model. Generating a project, or building and running a model on the processor, starts with configuring model options in the Configuration Parameters dialog box in Simulink software.

Setting Solver Parameters. After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

1 Open the Configuration Parameters dialog box and set the appropriate options on the `Solver` category for your model and for Embedded Coder.

- Set **Start time** to `0.0` and **Stop time** to `inf` (model runs without stopping). If you set a stop time, your generated code does not honor the setting. Set this to `inf` for completeness.

- Under **Solver options**, select the `fixed-step` and `discrete` settings from the lists
- Set the **Fixed step size** to `Auto` and the **Tasking Mode** to `Single Tasking`

Note Generated code does not honor Simulink software stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, add a Stop Simulation block in your model.

When you use PIL, you can set the **Solver options** to any selection from the **Type** and **Solver** lists.

Ignore the `Data Import/Export`, `Diagnostics`, and `Optimization` categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Code Generation Parameters. To configure your software to use the correct processor files and to compile and run your model executable file, set the options in the `Code Generation` category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your target:

1. Select `Code Generation` on the **Select** tree.
2. In `Target selection`, use the **Browse** button to set **System target file** to `idelink_grt.tlc`.

Setting Embedded Coder Parameters. To configure your software to use the correct code generation options and to compile and run your model executable file, set the options in the `IDE Link` category of the **Select** tree in the Configuration Parameters dialog box. Follow these steps to set the code generation options for your processor:

1. From the **Select** tree, choose `IDE Link` to specify code generation options that apply to your processor.
2. Set the following options in the pane under **Project options**:

- **Project options** should be `Custom`.
- Set **Compiler options string** and **Linker options string** should be blank.

3 Under **Link Automation**, verify that **Export IDE link handle to base workspace** is selected and provide a name for the handle in **IDE handle name** (optional).

4 Set the following **Runtime** options:

- **Build action**: `Build_and_execute`.
- **Interrupt overrun notification method**: `None`.

You have configured the your software options that let you generate a project for you processor. You may have noticed that you did not configure a few categories on the **Select** tree, such as `Comments`, `Symbols`, and `Optimization`.

For your new model, the default values for the options in these categories are correct. For other models you develop, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code. Refer to your product documentation for more information about setting the configuration parameters.

Building Your Project. After you set the configuration parameters and configure the coder software to create the files you need, you direct the build process to create your project:

1 Press **OK** to close the Configuration Parameters dialog box.

2 Click **Ctrl+B** to generate your project into CCS IDE.

   When you click **Build** with `Create_project` selected for **Build action**, the automatic build process starts CCS IDE, populates a new project in the development environment, builds the project, loads the binary on the processor, and runs it.

3 To stop processor execution, use the **Halt** option in CCS or enter `IDE_Obj.halt` at the MATLAB command prompt. (Where “`IDE_Obj`” is the IDE handle name you specified previously in **Configuration Parameters**.)

Model Reference

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your product documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your product documentation in the online Help system.

Model Reference in Simulation. When you simulate the top model, the coder software detects that your model contains referenced models. Simulink software generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink software rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these setting through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation. Embedded Coder software requires executables to generate code from models. If you have not simulated your model at least once, the coder software creates a `.mex` file for simulation.

Next, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, the coder software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference

With few limitations or restrictions, Embedded Coder provides full support for generating code from models that use model reference.

Build Action Setting. The most important requirement for using model reference with the TI's processors is that you must set the **Build action** (go to **Configuration Parameters > IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

1. Open your model.
2. Select **Simulation > Configuration Parameters** from the model menus.

   The Configuration Parameters dialog box opens.
3. From the **Select** tree, choose `IDE Link`.
4. In the right pane, under **Runtime**, select set `Archive_library` from the **Build action** list.

If your top model uses a reference model that does not have the build action set to `Archive_library`, the build process automatically changes the build action to `Archive_library` and issues a warning about the change.

As a result of selecting the `Archive_library` setting, other options are disabled:

- DSP/BIOS is disabled for all referenced models. Only the top model supports DSP/BIOS operation.
- **Interrupt overrun notification method**, **Export IDE link handle to the base workspace**, and **System stack size** are disabled for the referenced models.

Target Preferences Blocks in Reference Models. Each referenced model and the top model must include a Target Preferences block for the correct processor. You must configure all the Target Preferences blocks for the same processor.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require Target Preferences blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations. Model reference with Embedded Coder does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (in `c6000lib`) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (in `c6000lib`) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Embedded Coder block library

Configuring processors to Use Model Reference

processors that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible processor must be derived from the ERT or GRT processors.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation processor.
- The External mode option is not supported in model reference processor builds. Embedded Coder does not support External mode. If you select this option, it is ignored during code generation.
- To support model reference builds, your TMF must support use of the shared utilities folder, as described in Supporting Shared Utility Directories in the Build Process in the Simulink Coder documentation.

To use an existing processor, or a new processor, with Model Reference, you set the `ModelReferenceCompliant` flag for the processor. For information on how to set this option, refer to `ModelReferenceCompliant` in the online Help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference processor, use the following command to set the `ModelReferenceCompliant` flag to `On`:

```
set_param(bdroot,'ModelReferenceCompliant','on')
```

Models that you develop with versions 2.4 and later of Embedded Coder automatically include the model reference capability. You do not need to set the flag.

Exporting Filter Coefficients from FDATool

In this section...

"About FDATool" on page 54-69

"Preparing to Export Filter Coefficients to Code Composer Studio Projects" on page 54-70

"Exporting Filter Coefficients to Your Code Composer Studio Project" on page 54-74

"Preventing Memory Corruption When You Export Coefficients to Processor Memory" on page 54-79

About FDATool

Signal Processing Toolbox™ software provides the Filter Design and Analysis tool (FDATool) that lets you design a filter and then export the filter coefficients to a matching filter implemented in a CCS project.

Using FDATool with CCS IDE enables you to:

- Design your filter in FDATool
- Use CCS to test your filter on a processor
- Redesign and optimize the filter in FDATool
- Test your redesigned filter on the processor

For instructions on using FDATool, refer to the section "Filter Design and Analysis Tool" in the Signal Processing Toolbox documentation.

Procedures in this chapter demonstrate how to use the FDATool export options to export filter coefficients to CCS. Using these procedures, you can perform the following tasks:

- Export filter coefficients from FDATool in a header file—"Exporting Filter Coefficients from FDATool to the CCS IDE Editor" on page 54-75
- Export filter coefficients from FDATool to processor memory—"Replacing Existing Coefficients in Memory with Updated Coefficients" on page 54-81

Caution As a best practice, export coefficients in a header file for the most reliable results. Exporting coefficients directly to processor memory can generate unexpected results or corrupt memory.

Also see the reference pages for the following functions. These primary functions allow you use to access variables and write them to processor memory from the MATLAB Command window.

- `address` — Return the address of a symbol so you can read or write to it.
- `ticcs` — Create a connection between MATLAB software and CCS IDE so you can work with the project in CCS from the MATLAB Command window.
- `write` — Write data to memory on the processor.

Preparing to Export Filter Coefficients to Code Composer Studio Projects

- "Features of a Filter" on page 54-70
- "Selecting the Export Mode" on page 54-71
- "Choosing the Export Data Type" on page 54-72

Features of a Filter

When you create a filter in FDATool, the filter includes defining features identified in the following table.

Defining Feature	Description
Structure	Structure defines how the elements of a digital filter—gains, adders/subtractors, and delays—combine to form the filter. See the Signal Processing Toolbox documentation in the Online Help system for more information about filter structures.
Design Method	Defines the mathematical algorithm used to determine the filter response, length, and coefficients.

Defining Feature	Description
Response Type and Specifications	Defines the filter passband shape, such as lowpass or bandpass, and the specifications for the passband.
Coefficients	Defines how the filter structure responds at each stage of the filter process.
Data Type	Defines how to represent the filter coefficients and the resulting filtered output. Whether your filter uses floating-point or fixed-point coefficients affects the filter response and output data values.

When you export your filter, FDATool exports only the number of and value of the filter coefficients and the data type used to define the coefficients.

Selecting the Export Mode

You can export a filter by generating an ANSI C header file, or by writing the filter coefficients directly to processor memory. The following table summarizes when and how to use the export modes.

To...	Use Export Mode...	When to Use	Suggested Use
Add filter coefficients to a project in CCS	`C header file`	You implemented a filter algorithm in your program, but you did not allocate memory on your processor for the filter coefficients.	• Add the generated ANSI C header file to an appropriate project. Building and loading this project into your processor allocates static memory locations on the processor and writes your filter coefficients to those locations. • Edit the file so the header file allocates extra processor memory and then add the header file to your project. Refer to "Allocating Sufficient or Extra Memory for Filter Coefficients" on page 54-80 in the next section.

To...	Use Export Mode...	When to Use	Suggested Use
			(For a sample generated header file, refer to“Reviewing ANSI C Header File Contents” on page 54-77.)
Modify the filter coefficients in an embedded application loaded on a processor	`Write directly to memory`	You loaded a program on your processor. The program allocated space in your processor memory to store the filter coefficients.	• Optimize your filter design in FDATool. Then, • Write the updated filter coefficients directly to the allocated processor memory. Refer to section “Preventing Memory Corruption When You Export Coefficients to Processor Memory” on page 54-79 for more information.

Choosing the Export Data Type

The export process provides two ways you can specify the data type to use to represent the filter coefficients. Select one of the options shown in the following table when you export your filter.

Specify Data Type for Export	Description
Export suggested	Uses the data type that FDATool suggests to preserve the fidelity of the filter coefficients and the performance of your filter in the project
Export as	Lets you specify the data type to use to export the filter coefficients

FDATool exports filter coefficients that use the following data types directly without modifications:

- Signed integer (8, 16, or 32 bits)
- Unsigned integer (8, 16, or 32 bits)

- Double-precision floating point (64 bits)
- Single-precision floating point (32 bits)

Filters in FDATool in the Signal Processing Toolbox software use double-precision floating point. You cannot change the data type.

If you have installed DSP System Toolbox software, you can use the filter quantization options in FDATool to set the word and fraction lengths that represent your filter coefficients. For information about using the quantization options, refer to Filter Design and Analysis Tool in the Filter Design Toolbox documentation in the Online help system.

If your filter uses one of the supported data types, **Export suggested** specifies that data type.

If your filter does not use one of the supported data types, FDATool converts the unsupported data type to one of the supported types and then suggests that data type. For more information about how FDATool determines the data type to suggest, refer to "How FDATool Determines the Export Suggested Data Type" on page 54-74.

Follow these best-practice guidelines when you implement your filter algorithm in source code and design your filter in FDATool:

- Implement your filter using one of the data types FDATool exports without modifications.
- Design your filter in FDATool using the data type you used to implement your filter.

To Choose the Export Data Type. When you export your filter, follow this procedure to select the export data type to ensure the exported filter coefficients closely match the coefficients of your filter in FDATool.

1. In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.
2. Perform one of the following actions:
   - Select **Export suggested** to export the coefficients in the suggested data type.

- Select **Export as** and choose the data type your filter requires from the list.

Caution If you select **Export as**, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

How FDATool Determines the Export Suggested Data Type. By default, FDATool represents filter coefficients as double-precision floating-point data. When you export your filter coefficients, FDATool suggests the same data type.

If you set custom word and fraction lengths to represent your filter coefficients, the export process suggests a data type to maintain the best fidelity for the filter.

The export process converts your custom word and fraction lengths to a suggested export data type, using the following rules:

- Round the word length up to the nearest larger supported data type. For example, round an 18-bit word length up to 32 bits.
- Set the fraction length to the maintain the same difference between the word and fraction length in the new data type as applies in the custom data type.

  For example, if you specify a fixed-point data type with word length of 14 bits and fraction length of 11 bits, the export process suggests an integer data type with word length of 16 bits and fraction length of 13 bits, retaining the 3 bit difference.

Exporting Filter Coefficients to Your Code Composer Studio Project

- “Exporting Filter Coefficients from FDATool to the CCS IDE Editor” on page 54-75
- “Reviewing ANSI C Header File Contents” on page 54-77

Exporting Filter Coefficients from FDATool to the CCS IDE Editor

In this section, you export filter coefficients to a project by generating an ANSI C header file that contains the coefficients. The header file defines global arrays for the filter coefficients. When you compile and link the project to which you added the header file, the linker allocates the global arrays in static memory locations in processor memory.

Loading the executable file into your processor allocates enough memory to store the exported filter coefficients in processor memory and writes the coefficients to the allocated memory.

Use the following steps to export filter coefficients from FDATool to the CCS IDE text editor.

1. Start FDATool by entering `fdatool` at the MATLAB command prompt.

   ```
   fdatool     % Starts FDATool.
   ```

2. Design a filter with the same structure, length, design method, specifications, and data type you implemented in your source code filter algorithm.

   The following figure shows a Direct-form II IIR filter example that uses second-order sections.

3. Click **Store Filter** to store your filter design. Storing the filter allows you to recall the design to modify it.

4. To export the filter coefficients, select **Targets > Code Composer Studio IDE** from the FDATool menu bar.

   The Export to Code Composer Studio IDE dialog box opens, as shown in the following figure.

5. Set **Export mode** to `C header file`.

6 In **Variable names in C header file**, enter variable names for the **Numerator**, **Denominator**, **Numerator length**, and **Denominator length** parameters where the coefficients will be stored.

   The dialog box shows only the variables you need to export to define your filter.

Note You cannot use reserved ANSI C programming keywords, such as `if` or `int` as variable names, or include invalid characters such as spaces or semicolons (`;`).

---

7 In **Data type to use in export**, select **Export suggested** to accept the recommended export data type. FDATool suggests a data type that retains filter coefficient fidelity.

   You may find it useful to select the **Export as** option and select an export data type other than the one suggested.

Caution If you deviate from the suggested data type, the exported filter coefficients can be very different from the filter coefficients in FDATool. As a result, your filter cutoff frequencies and performance may not match your design in FDATool.

For more information about how FDATool decides which data type to suggest, refer to "How FDATool Determines the Export Suggested Data Type" on page 54-74.

8 If you know the board number and processor number of your target, enter **DSP Board #** and **DSP Processor #** values to identify your board.

   When you have only one board or simulator, Embedded Coder software sets **DSP Board #** and **DSP Processor #** values for your board automatically.

   If you have more than one board defined in CCS Setup:

   - Click **Select target** to open the Selection Utility dialog box.

- From the list of boards and list of processors, select the board name and processor name to use.
- Click **Done** to set the **DSP Board #** and **DSP Processor #** values.

9. Click **Generate** to generate the ANSI header file. FDATool prompts you for a file name and location to save the generated header file.

   The default location to save the file is your MATLAB working folder. The default file name is `fdacoefs.h`.

10. Click **OK** to export the header file to the CCS editor.

    If CCS IDE is not open, this step starts the IDE.

    The export process does not add the file to your active project in the IDE.

11. Drag your generated header file into the project that implements the filter.

12. Add a `#include` statement to your project source code to include the new header file when you build your project.

13. Generate a `.out` file and load the file into your processor. Loading the file allocates locations in static memory on the processor and writes the filter coefficients to those locations.

To see an example header file, refer to “Reviewing ANSI C Header File Contents” on page 54-77.

Reviewing ANSI C Header File Contents

The following program listing shows the exported header (`.h`) file that FDATool generates. This example shows a direct-form II filter that uses five second-order sections. The filter is stable and has linear phase.

Comments in the file describe the filter structure, number of sections, stability, and the phase of the filter. Source code shows the filter coefficients and variables associated with the filter design, such as the numerator length and the data type used to represent the coefficients.

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
```

```
 *
 * Generated by MATLAB(R) 7.8 and the Signal Processing Toolbox 6.11.
 *
 * Generated on: xx-xxx-xxxx 14:24:45
 *
 */

/*
 * Discrete-Time IIR Filter (real)
 * -------------------------------
 * Filter Structure    : Direct-Form II, Second-Order Sections
 * Number of Sections  : 5
 * Stable              : Yes
 * Linear Phase        : No
 */

/* General type conversion for MATLAB generated C-code  */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * $MATLABROOT\extern\include\tmwtypes.h
 */
#define MWSPT_NSEC 11
const int NL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real64_T NUM[MWSPT_NSEC][3] = {
  {
      0.802536131462,                0,                0
  },
  {
     0.2642710234701,   0.5285420469403,   0.2642710234701
  },
  {
                   1,                0,                0
  },
  {
     0.1743690465012,   0.3487380930024,   0.1743690465012
  },
```

```
  {
     0.2436793028081,   0.4873586056161,   0.2436793028081
  },
  {
                   1,                 0,                 0
  },
  {
     0.3768793219093,   0.7537586438185,   0.3768793219093
  },
  {
                   1,                 0,                 0
  }
};
const int DL[MWSPT_NSEC] = { 1,3,1,3,1,3,1,3,1,3,1 };
const real64_T DEN[MWSPT_NSEC][3] = {
  {
                   1,                 0,                 0
  },
  {
                   1,  -0.1842138030775,   0.1775781189277
  },
  {
                   1,                 0,                 0
  },
  {
                   1,  -0.2160098642842,   0.3808329528195
  },
  {
                   1,                 0,                 0
  }
};
```

Preventing Memory Corruption When You Export Coefficients to Processor Memory

- "Allocating Sufficient or Extra Memory for Filter Coefficients" on page 54-80
- "Example: Using the Exported Header File to Allocate Extra Processor Memory" on page 54-80

- "Replacing Existing Coefficients in Memory with Updated Coefficients" on page 54-81
- "Example: Changing Filter Coefficients Stored on Your Processor" on page 54-82

Allocating Sufficient or Extra Memory for Filter Coefficients

You can allocate extra memory by editing the generated ANSI C header file. You can then load the associated program file into your processor as described in "Example: Using the Exported Header File to Allocate Extra Processor Memory" on page 54-80. Extra memory lets you change filter coefficients and overwrite existing coefficients stored in processor memory more easily.

To prevent problems when you update filter coefficients in a project, , such as writing coefficients to unintended memory locations, use the `C header file` export mode option in FDATool to update filter coefficients in your program.

Example: Using the Exported Header File to Allocate Extra Processor Memory

You can edit the generated header file so the linked program file allocates extra processor memory. By allocating extra memory, you avoid the problem of insufficient memory when you export new coefficients directly to allocated memory.

For example, changing the following command in the header file:

```
const real64_T NUM[47] = {...}
```

to

```
real64_T NUM[256] = {...}
```

allocates enough memory for `NUM` to store up to 256 numerator filter coefficients rather than 47.

Exporting the header file to CCS IDE does not add the filter to your project. To incorporate the filter coefficients from the header file, add a `#include` statement:

```
#include "headerfilename.h"
```

Refer to “Exporting Filter Coefficients to Your Code Composer Studio Project” on page 54-74 for information about generating a header file to export filter coefficients.

When you export filter coefficients directly to processor memory, the export process writes coefficients to as many memory locations as they need. The write process does not perform bounds checking. To ensure you write to the correct locations, and have enough memory for your filter coefficients, plan memory allocation carefully.

Replacing Existing Coefficients in Memory with Updated Coefficients

When you redesign a filter and export new coefficients to replace existing coefficients in memory, verify the following conditions for your new design:

- Your redesign did not increase the memory required to store the coefficients beyond the allocated memory.

  Changes that increase the memory required to store the filter coefficients include the following redesigns:

  - Increasing the filter order
  - Changing the number of sections in the filter
  - Changing the numerical precision (changing the export data type)
- Your changes did not change the export data type.

Caution Identify changes that require additional memory to store the coefficients before you begin your export. Otherwise, exporting the new filter coefficients may overwrite data in memory locations you did not allocate for storing coefficients. Also, exporting filter coefficients to memory after you change the filter order, structure, design algorithm, or data type can yield unexpected results and corrupt memory.

Changing the filter design algorithm in FDATool, such as changing from Butterworth to Maximally Flat, often changes the number of filter coefficients (the filter order), the number of sections, or both. Also, the coefficients from

the new design algorithm may not perform properly with your source code filter implementation.

If you change the design algorithm, verify that your filter structure and length are the same after you redesign your filter, and that the coefficients will perform properly with the filter you implemented.

If you change the number of sections or the filter order, your filter will not perform properly unless your filter algorithm implementation accommodates the changes.

Example: Changing Filter Coefficients Stored on Your Processor

This example writes filter coefficients to processor memory to replace the existing coefficients. To perform this process, you need the names of the variables in which your project stores the filter data.

Before you export coefficients directly to memory, verify that your project allocated enough memory for the new filter coefficients. If your project allocated enough memory, you can modify your filter in FDATool and then follow the steps in this example to export the updated filter coefficients to the allocated memory.

If your new filter requires additional memory space, use a C header file to allocate memory on the processor and export the new coefficients as described in "Exporting Filter Coefficients to Your Code Composer Studio Project" on page 54-74.

For important guidelines on writing directly to processor memory, refer to "Preventing Memory Corruption When You Export Coefficients to Processor Memory" on page 54-79.

Follow these steps to export filter coefficients from FDATool directly to memory on your processor.

1. Load the program file that contains your filter into CCS IDE to activate the program symbol table. The symbol must contain the global variables you use to store the filter coefficients and length parameters.

2. Start FDATool.

3. Click **Filter Manager** to open the Filter Manager dialog box, shown in the following figure.

4. Highlight the filter to modify on the list of filters, and select **Edit current filter**. The highlighted filter appears in FDATool for you to change.

   If you did not store your filter from a previous session, design the filter in FDATool and continue.

5. Click **Close** to dismiss the Filter Manager dialog box.

6. Adjust the filter specifications in FDATool to modify its performance.

7. In FDATool, select **Targets > Code Composer Studio IDE** to open the Export to Code Composer Studio IDE dialog box.

   Keep the export dialog box open while you work. When you do so, the contents update as you change the filter in FDATool.

   Tip Click **Generate** to export coefficients to the same processor memory location multiple times without reentering variable names.

   ---

8. In the Export to Code Composer Studio dialog box:

   - Set **Export mode** to `Write directly to memory`
   - Clear **Disable memory transfer warnings** to get a warning if your processor does not support the export data type.

9. In **Variable names in target symbol table**, enter the names of the variables in the processor symbol table that correspond to the memory allocated for the parameters, such as **Numerator** and **Denominator**. Your names must match the names of the filter coefficient variables in your program.

10. Select **Export suggested** to accept the recommended export data type.

For more information about how FDATool determines the data type to suggest, refer to “How FDATool Determines the Export Suggested Data Type” on page 54-74.

11 If you know the board number and processor number of your target, enter **DSP Board #** and **DSP Processor #** values to identify your board.

Note When you have only one board or simulator, Embedded Coder sets **DSP Board #** and **DSP Processor #** to your board automatically.

If you have more than one board defined in CCS Setup:

- Click **Select target** to open the Selection Utility dialog box.
- Select the board name and processor name to use from the list of boards.

12 Click **Generate** to export your filter. If your processor does not support the data type you export, you see a warning similar to the following message.

You can continue to export the filter, or cancel the export process. To prevent this warning dialog box from appearing, select **Disable memory transfer warnings** in the Export to Code Composer Studio IDE dialog box.

13 (Optional) Continue to optimize filter performance by modifying your filter in FDATool and then export the updated filter coefficients directly to processor memory.

14 When you finish testing your filter, return to FDATool, and click **Store filter** to save your changes.

Tutorial: Using XMakefile with Code Composer Studio 4.x

In this section...

"Introduction" on page 54-85

"Set Up XMakefile for CCSv4" on page 54-85

"Prepare Your Model for CCSv4 and Makefiles" on page 54-87

"Create Target Configuration File in CCSv4" on page 54-87

"Load and Run the Embedded Software" on page 54-88

Introduction

This tutorial shows you how to use the XMakefile feature in your MathWorks software to build and run embedded software with Code Composer Studio 4.x (CCSv4). For more information about XMakefile, see "Makefiles" on page 43-24

Note The Embedded Coder Project Generator feature is not available for CCSv4 in the current release. For more information about Project Generator, see "IDE Projects" on page 43-18

To build the target software, complete the process covered in this chapter:

- Set up XMakefile for CCSv4.
- Prepare your model for CCSv4.
- Create a Target Configuration File in CCSv4.
- Load and run the embedded software.

Set Up XMakefile for CCSv4

The XMakefile feature tells your MathWorks software how to create makefiles for a *configuration*, which is a specific combination of tool chain and embedded target. Some configurations require additional information before you can use them.

Select and complete a configuration for Code Composer Studio 4.0 (CCSv4):

1. Enter `xmakefilesetup` at the MATLAB command prompt. This action opens the XMakefile User Configuration dialog box.

2. Deselect **Display operational configurations only**.

3. Set **Configurations** to the choice that matches your target and ends with `ccsv4`, and click **Apply**.

4. If the configuration is incomplete, the software displays a series of **Browse For Folder** dialog boxes that include instructions to provide missing information.

5. Examine the **Tool Directories** tab to see if the paths are correct.

6. When you have supplied the missing information, and the configuration is complete, click **OK** to close the XMakefile User Configuration dialog box.

For example, to generate code for CCSv4 and a C6000 processor:

1. Enter `xmakefilesetup` on the command line.

2. In the XMakefile dialog box, deselect **Display operational configurations only**, set **Configurations** to `ticcs_c6000_ccsv4`, and click **Apply**.

3. A **Browse For Folder** appears, stating "Select the C6000 Code Generation Tools root installation directory...".

   Browse and select a path such as `C:\Program Files\Texas Instruments\C6000 Code Generation Tools`.

4. Another **Browse For Folder** dialog appears, stating "Select the C6000 CSL root installation directory...".

   Browse and select a path such as `C:\Program Files\C6xCSL\`.

5. Examine the **Tool Directories** tab to see if the paths are correct.

6 With the updated information, the `ticcs_c6000_ccsv4` configuration is operational. Click **OK** to save the updated configuration, and close the dialog box.

Prepare Your Model for CCSv4 and Makefiles

Configure your model to generate code for CCSv4 by updating the Target Preferences block.

If your model does not contain a Target Preferences block:

1 Open the Common block library by entering `idelinklib_common` on the command line.

2 Copy the Target Preferences block to your model.

3 Complete the **Target Preferences: Initialize Configuration Parameters** dialog box:

- Set **IDE/Tool Chain** to `Texas Instruments Code Composer Studio v4 (makefile generation only)`.
- Set **Board** and **Processor** to the appropriate choices.

4 Click **Yes**. This action updates the appropriate values in your Target Preferences and model Configuration Parameters.

5 Build your embedded software by pressing **CTRL+B**.

If your model already contains a Target Preferences block, open the block and update the parameters described in step 3 and 4.

Create Target Configuration File in CCSv4

Before loading and running your target software, use the CCSv4 IDE to create a “target configuration file”. The TI Debug Server uses this file while it works with CCSv4 to load and run your target software. The XML-based target configuration file describes the target board and processor. The file name ends with a `*.ccxml` extension.

Create a target configuration file:

1. In the CCSv4, select **File** > **New** > **Target Configuration File** to display a **New Target Configuration** dialog box:
   - For **File name**, update the file name that ends with `.ccxml` to describe your project and hardware.
   - Click **Finish**. This action displays a utility in the CCS editor pane for customizing the target configuration file.
2. Use the utility to select the **Connection** and **Device** type. Typing a partial string next to **Device** filters the list of devices.
3. Click **Save**.

Note For more information about target configuration files, consult the Texas Instruments for CCSv4.

Load and Run the Embedded Software

First set the Windows system variable, Path, so you can call the TI Debug Server Scripting (DSS) API from any folder.

1. In Windows, right-click **My Computer**, and select **Properties**. This action opens the System Properties dialog box.
2. In **System Properties**, select the **Advanced** tab, and click **Environment Variables**. This action opens the Environment Variables dialog box.
3. In **Environment Variables**, under **System variables**, select the `Path` variable, and click **Edit**. This action opens the Edit System Variable dialog box.
4. In Edit System Variable, for **Variable value**, append a semicolon and the full path of the `\ccsv4\scripting\bin` subdirectory. For example, append `;C:\Program Files\Texas Instruments\ccsv4\scripting\bin`.

For more information about using DSS, see http://processors.wiki.ti.com/index.php/Debug_Server_Scripting.

MathWorks provides an example JavaScript file, runProgram.js, for you to use with DSS. This script loads and runs the specified program on the target specified in the target configuration file. You can create a copy of this script and modify it to suit your needs. The location of `runProgram.js` is:

```
[MATLABROOT]\toolbox\idelink\extensions\ticcs\ccsdemos
```

The specific syntax for running dss.bat with runProgram.js is:

```
> dss runProgram.js targetConfigurationFile programFile
```

Replace *targetConfigurationFile* and *programFile* with the appropriate paths and file names. For example, if you are using a working directory called the CCSv4 workspace, and the model name is myProgram, enter:

```
> dss runProgram.js c:\workspace\myC6416dsk.ccxml myProgram.out
```

This command builds and loads your software on the target or simulator. You have completed the process of using XMakefile with CCSv4 to load and run software on an target or simulator.

You have completed the process of loading and running embedded software using XMakefile and CCSv4.

Tip To use advanced DSS features, you can also use the CCSv4 example batch file, loadti.bat, as follows:

Change directories to the `loadti` subdirectory. For example:

```
> cd c:\ccs4_install\ccsv4\scripting\examples\loadti
```

Run loadti.bat using the following syntax:

```
> loadti -a -c=targetConfigurationFile programFile
```

Replace `targetConfigurationFile` with the complete path of the target configuration file.

Replace `programFile` with the name of the .out created using the XMakefile feature. For example:

```
> loadti -a -c=c:\workspace\myC6416dsk.ccxml myProgram.out
```

For more information about loadti and its options, type the following on your system command prompt

```
> loadti -help
```

Reported Limitations and Tips

In this section...

"Demonstration Programs Do Not Run Properly Without Correct GEL Files" on page 54-92

"Changing Values of Local Variables Does Not Take Effect" on page 54-92

"Code Composer Studio Cannot Find a File After You Halt a Program" on page 54-93

"C54x XPC Register Can Be Modified Only Through the PC Register" on page 54-94

"Working with More Than One Installed Version of Code Composer Studio" on page 54-95

"Changing CCS Versions During a MATLAB Session" on page 54-95

"MATLAB Hangs When Code Composer Studio Cannot Find a Board" on page 54-95

"Using Mapped Drives" on page 54-97

"Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting" on page 54-97

"PostCodeGenCommand Commands Do Not Affect Embedded Coder Projects" on page 54-98

Some long-standing issues affect the Embedded Coder product. When you are using `ticcs` objects and the software methods to work with Code Composer Studio and supported hardware or simulators, recall the information in this section.

The latest issues in the list appear at the bottom. HIL refers to "hardware in the loop," also called processor in the loop (PIL) here and in other applications, and sometimes referred to as function calls.

Demonstration Programs Do Not Run Properly Without Correct GEL Files

To run the Embedded Coder demos, you must load the appropriate GEL files before you run the demos. For some boards, the demos run fine with the default CCS GEL file. Some boards need to run device-specific GEL files for the demos to work correctly.

Here are demos and boards which require specific GEL files.

- Board: C5416 DSK

  Demos: `rtdxtutorial`, `rtdxlmsdemo`

  Emulator: XDS-510

  GEL file to load: `c5416_dsk.gel`

In general, if a demo does not run correctly with the default GEL file, try using a device-specific GEL file by defining the file in the CCS Setup Utility.

Changing Values of Local Variables Does Not Take Effect

If you halt the execution of your program on your DSP and modify a local variable's value, the new value may not be acknowledged by the compiler. If you continue to run your program, the compiler uses the original value of the variable.

This problem happens only with local variables. When you write to the local variable via the Code Composer Studio Watch Window or via a MATLAB object, you are writing into the variable's absolute location (register or address in memory).

However, within the processor function, the compiler sometimes saves the local variable's values in an intermediate location, such as in another register or to the stack. That intermediate location cannot be determined or changed/updated with a new value during execution. Thus the compiler uses the old, unchanged variable value from the intermediate location.

Code Composer Studio Cannot Find a File After You Halt a Program

When you halt a running program on your processor, Code Composer Studio may display a dialog box that says it cannot find a source code file or a library file.

When you halt a program, CCS tries to display the source code associated with the current program counter. If the program stops in a system library like the runtime library, DSP/BIOS, or the board support library, it cannot find the source code for debug. You can either find the source code to debug it or select the **Don't show this message again** checkbox to ignore messages like this in the future.

For more information about how CCS responds to the halt, refer the online Help for CCS. In the online help system, use the search engine to search for the keywords "Troubleshooting" and "Support." The following information comes from the online help for CCS, starting with the error message:

File Not Found

The debugger is unable to locate the source file necessary to enable source-level debugging for this program.

To specify the location of the source file

1. Click **Yes**. The Open dialog box appears.

2. In the Open dialog box, specify the location and name of the source file then click **Open**.

The next section provides more details about file paths.

Defining a Search Path for Source Files

The Directories dialog box enables you to specify the search path the debugger uses to find the source files included in a project.

To Specify Search Path Directories

1 Select **Option > Customize**.

2 In the Customize dialog box, select the **Directories** tab. Use the scroll arrows at the top of the dialog box to locate the tab.

   The Directories dialog box offers the following options.

   - `Directories`. The `Directories` list displays the defined search path. The debugger searches the listed folders in order from top to bottom.

     If two files have the same name and are located in different folders, the file located in the folder that appears highest in the `Directories` list takes precedence.

   - `New`. To add a new folder to the `Directories` list, click **New**. Enter the full path or click **browse** [...] to navigate to the appropriate folder. By default, the new folder is added to the bottom of the list.

   - `Delete`. Select a folder in the `Directories` list, then click **Delete** to remove that folder from the list.

   - `Up`. Select a folder in the `Directories` list, then click **Up** to move that folder higher in the list.

   - `Down`. Select a folder in the `Directories` list, then click **Down** to move that folder lower in the list.

3 Click **OK** to close the Customize dialog box and save your changes.

C54x XPC Register Can Be Modified Only Through the PC Register

You cannot modify the XPC register value directly using `regwrite` to write into the register. When you are using extended program addressing in C54x, you can modify the XPC register by using `regwrite` to write a 23-bit data value in the PC register. Along with the 16-bit PC register, this operation also modifies the 7-bit XPC register that is used for extended program addressing. On the C54x, the PC register is 23 bits (7 bits in XPC + 16 bits in PC).

You can then read the XPC register value using `regread`.

Working with More Than One Installed Version of Code Composer Studio

When you have more than one version of Code Composer Studio installed on your machine, you cannot select which CCS version MATLAB Embedded Coder attaches to when you create a `ticcs` object. If, for example, you have both CCS for C5000 and CCS for C6000 versions installed, you cannot choose to connect to the C6000 version rather than the C5000 version.

When you issue the command

```
IDE_obj = ticcs
```

Embedded Coder starts the CCS version you last used. If you last used your C5000 version, the `IDE_obj` object accesses the C5000 version.

Workaround

To make your `ticcs` object access the correct processor:

1. Start and close the appropriate CCS version before you create the `ticcs` object in MATLAB.

2. Create the `ticcs` object using the `boardnum` and `procnum` properties to select your processor, if needed.

   Recall that `ccsboardinfo` returns the `boardnum` and `procnum` values for the processors that CCS recognizes.

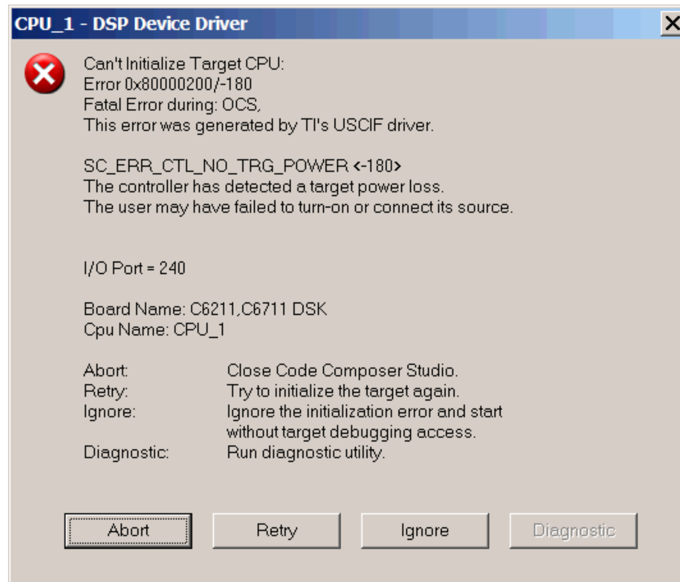
Changing CCS Versions During a MATLAB Session

You can use only one version of CCS in a single MATLAB session. Embedded Coder does not support using multiple versions of CCS in a MATLAB session. To use another CCS version, exit MATLAB software and restart it. Then create your links to the new version of CCS.

MATLAB Hangs When Code Composer Studio Cannot Find a Board

In MATLAB software, when you create a `ticcs` object, the construction process for the object automatically starts CCS. If CCS cannot find a processor

that is connected to your PC, you see a message from CCS like the following DSP Device Driver dialog box that indicates CCS could not initialize the processor.



Four options let you decide how to respond to the failure:

- **Abort** — Closes CCS and suspends control for about 30 seconds. If you used MATLAB software functions to open CCS, such as when you create a `ticcs` object, the system returns control to MATLAB command window after a considerable delay, and issues this warning:

  ```
  ??? Unable to establish connection with Code Composer Studio.
  ```

- **Ignore** — Starts CCS without connecting to any processor. In the CCS IDE you see a status message that says EMULATOR DISCONNECTED in the status area of the IDE. If you used MATLAB to start CCS, you get control immediately and Embedded Coder creates the `ticcs` object. Because CCS is not connected to a processor, you cannot use the object to perform processor operations from MATLAB, such as loading or running programs.

- **Retry** — CCS tries again to initialize the processor. If CCS continues not to find your hardware processor, the same DSP Device Driver dialog box reappears. This process continues until either CCS finds the processor or you choose one of the other options to respond to the warning.

One more option, **Diagnostic**, lets you enter diagnostic mode if it is enabled. Usually, **Diagnostic** is not available for you to use.

Using Mapped Drives

Limitations in Code Composer Studio do not allow you to load programs after you set your CCS working folder to a read-only mapped drive. When you set the CCS working folder to a mapped drive for which you do not have write permissions, you cannot load programs from any location. Load operations fail with an Application Error dialog box.

The following combination of commands does not work:

1. ```
   cd(IDE_obj,'mapped_drive_letter') % Change CCS working
   directory to read-only mapped drive.
   ```

2. ```
   load(IDE_obj,'program_file') % Loading any program fails.
   ```

Uninstalling Code Composer Studio 3.3 Prevents Embedded Coder From Connecting

Description On a machine where CCS V3.3 and CCS V3.1 are installed, uninstalling V3.3 makes V3.1 unusable from MATLAB. This is because the CCS V3.3 uninstaller leaves stale registry entries in the Windows Registry that prevent MATLAB from connecting to CCS V3.1.

Texas Instruments has documented this uninstall problem and the solution on their Web site.

Updated information on this issue may also be available from the Bug Reports section of www.mathworks.com at http://www.mathworks.com/support/bugreports/379676

PostCodeGenCommand Commands Do Not Affect Embedded Coder Projects

PostCodeGenCommand commands, such as the addCompileFlags and addLinkFlags functions in the BuildInfo API do not affect code generated by Embedded Coder.

Use the ’Compiler options string’ and ’Linker options string’ parameters in the **Configuration Parameters > Code Generation > IDE Link** pane instead. You can also automate this process using a model callback to SET_PARAM the ’CompilerOptionsStr’ and ’LinkerOptionsStr’ parameters.

Working with Texas Intruments C2000 Processors

- “Setting Up and Configuring” on page 55-2
- “Data Type Support” on page 55-5
- “Scheduling and Timing” on page 55-6
- “Sharing General Purpose Timers between C281x Peripherals” on page 55-12
- “Overview of Creating Models for C2000 Processors” on page 55-21
- “Using the c2000lib Blockset” on page 55-23
- “Configuring Timing Parameters for CAN Blocks” on page 55-29
- “Configuring Acquisition Window Width for ADC Blocks” on page 55-47
- “Using the IQmath Library” on page 55-53
- “Programming Flash Memory” on page 55-62
- “Configuring LIN Communications” on page 55-68

Setting Up and Configuring

In this section...

"Installing and Configuring Software" on page 55-2
"Verifying the Configuration" on page 55-3

Installing and Configuring Software

Uninstall unsupported or untested versions of the third party products *before* installing supported versions. Doing so prevents errors that occur when the Windows Environment Variables points to the unsupported versions.

Install the software listed in the following order:

1. Install the required and optional MathWorks software. (The software license you purchase determines which products are available.)
2. Install TI Code Composer Studio (CCS).
3. Install TI Service Release for CCS.
4. Install the TI Code Generation Tools for Windows.
5. If you are using a Spectrum Digital board, download and install the matching Spectrum Digital Driver.
6. If you are using RTDX for C28x host/target communications, download and install TI DSP/BIOS.
7. If you are going to program flash memory with stand-alone code, download the TI Flash API for your target processor.

Configure CCS as follows:

1. In CCS, open **Help > About > Component manager > Build tools > TMS320C28XX** and select (check) **C2000 Code Generation Tools**.
2. With the Component manager open, open **Target Content(DSP/BIOS) > TMS320C28XX** and select **Texas Instruments DSP/BIOS**.

3 Save, exit, and restart CCS.

Verifying the Configuration

To determine whether Embedded Coder software is present on your system, enter this command at the MATLAB prompt:

```
c2000lib
```

MATLAB displays the C2000 block libraries.

If you do not see the listed libraries, or MATLAB does not recognize the command, install the Embedded Coder software. Without the software, you cannot use Simulink and Simulink Coder software to develop applications targeted to the TI boards.

To verify that Code Composer Studio (CCS) is present on your machine, enter this command at the MATLAB prompt:

```
ccsboardinfo
```

With CCS installed and configured, MATLAB returns a list of the boards that CCS recognizes on your machine like the following example:

```
Board Board              Proc Processor  Processor
Num   Name               Num  Name       Type
---   ---------------------------------- ---
1     F2812 Simulator     0  CPU         TMS320C28xx
0     F2812 PP Emulator   0  CPU_1       TMS320C28xx
```

If MATLAB does not return information about any boards, revisit the installation and setup instructions in your CCS documentation. If you have not done so already, install the third-party “Board Support Packages” for your boards.

As a final test, launch CCS to ensure that it starts up successfully. For Embedded Coder software to operate with CCS, the CCS IDE must be able to run on its own.

Note For any model to work in the targeting environment, select the discrete-time solver in the **Solver** pane of the Simulink Configuration Parameters dialog box. Targeting does not work with continuous-time solvers.

To select the discrete-time solver, from the main menu in your model window, select **Simulation > Configuration Parameters**. Then in the **Solver** pane, set the **Solver** option to `Discrete (no continuous states)`.

Data Type Support

TI C2000 DSPs support 16 and 32–bit data types, but does not have native 8-bit data types. Simulink models and Embedded Coder software support many data types, including 8-bit data types.

If you select `int8` or `uint8` in your model, your simulation runs with 8-bit data, but in the generated code, that data will be represented as 16-bit. This may cause instances where data overflow and wraparound occurs in the simulation, but not in the generated code.

For example, to make the overflow behavior of the simulation and generated code match for a Simulink Add block in your model, select **Saturate on integer overflow** in that block.

Scheduling and Timing

In this section...

"Overview" on page 55-6
"Timer-Based Interrupt Processing" on page 55-6
"Asynchronous Interrupt Processing" on page 55-7

Overview

Normally the code generated by Embedded Coder software runs in the context of a timer interrupt. Model blocks run in a periodical fashion clocked by the periodical interrupt whose period is tied to the base sample time of the model.

This execution scheduling model, however, is not flexible enough for many systems, especially control and communication systems, which must respond to external events in real time. Such systems require the ability to handle various hardware interrupts in an asynchronous fashion.

Embedded Coder software lets you model and generate code for such systems by creating tasks driven by Hardware Interrupt blocks in addition to the tasks that are left to be handled in the context of the timer interrupt.

Timer-Based Interrupt Processing

For code that runs in the context of the timer interrupt, each iteration of the model solver is run after an interrupt has been posted and serviced by an interrupt service routine (ISR). The code generated for the C280x, C281x, and C28x3x uses `CPU_timer0`.

The timer is configured so that the model's base rate sample time corresponds to the interrupt rate. The timer period and prescaler are calculated and set up to ensure the desired rate as follows:

$$BaseRateSampleTime = \frac{TimerPeriod}{TimerClockSpeed}$$

The minimum achievable base rate sample time depends on the model complexity. The maximum value depends on the maximum timer period value (2^{32}-1 for the F2812, F2808, and F28x35) and the CPU clock speed. The CPU clock speed is 100 MHz for the F2808, and 150 MHz for the F2812 and F28335.

If all the blocks in the model inherit their sample time value, and no sample time is explicitly defined, the default value is 0.2 s.

High-Speed Peripheral Clock

The Event Managers and their general-purpose timers, which drive PWM waveform generation use the high-speed peripheral clock (HISCLK). By default, this clock is always selected in Embedded Coder software. This clock is derived from the system clock (SYSCLKOUT):

HISCLK = [SYSCLKOUT / (high-speed peripheral prescaler)]

The high-speed peripheral prescaler is determined by the HSPCLK bits set in SysCtrl. The default value of HSPCLK is 1, which corresponds to a high-speed peripheral prescaler value of 2.

For example, on the F2812, the HISCLK rate becomes

HISCLK = 150 MHz / 2 = 75 MHz

Asynchronous Interrupt Processing

Simulink and Simulink Coder software facilitate the modeling and generation of code for asynchronous event handling, including servicing of hardware-generated interrupts, by using the following special blocks:

- Hardware Interrupt block

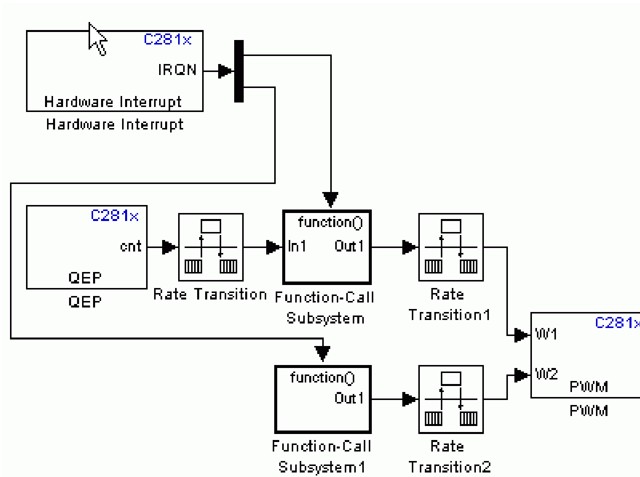
  This block enables selected hardware interrupts, generates the corresponding interrupt service routines (ISRs), and connects them to the corresponding interrupt service vector table entries. When you connect the output of the Hardware Interrupt block to the control input of a triggered subsystem (for example, a function-call subsystem), the generated subsystem code is called from the ISRs.

  Embedded Coder software provides a Hardware Interrupt block for each of the supported processor families.

- Rate Transition blocks

  These blocks support data transfers between blocks running with different sample rates. The built-in Simulink Rate Transition blocks can be used for this purpose.

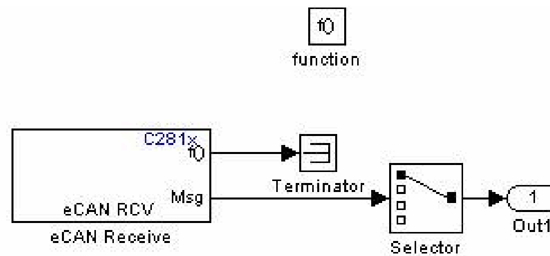
The following diagram illustrates a use case where a Hardware Interrupt block triggers two tasks, connected to other blocks that run periodically in the context of the synchronous scheduler.



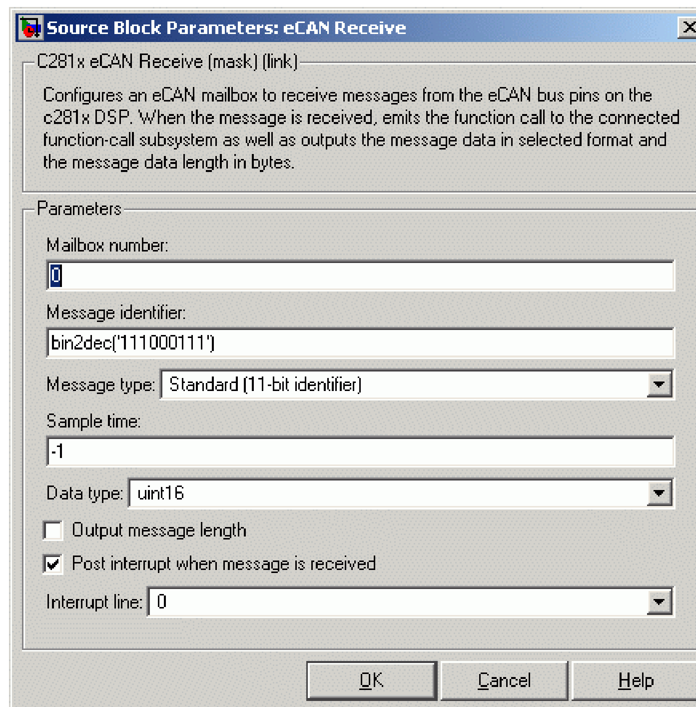
In the preceding figure, the Hardware Interrupt block is set to react on two interrupts. Since only one Hardware Interrupt block is allowed in a model and the output of this block is a vector of length two, you must connect the Hardware Interrupt block to a Demux block to trigger the two function-call subsystems. The function-call subsystems contain the blocks that are executed asynchronously in the context of the hardware interrupt.

The following example shows how to build and configure a model to react on an eCAN message using a hardware interrupt and an asynchronous scheduler:

1. Place the eCAN Receive block in a function-call subsystem.

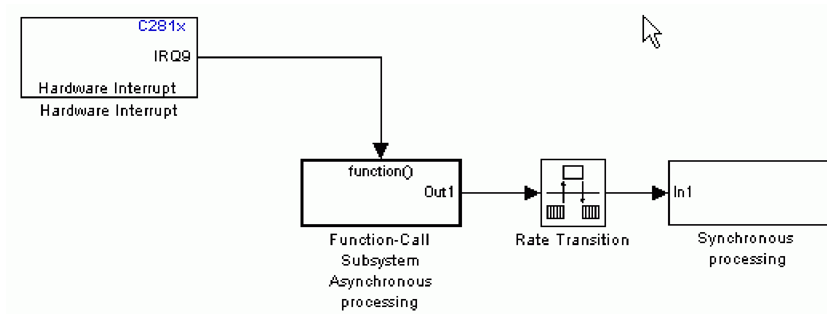


2 On the eCAN Receive block dialog, check the box labeled **Post interrupt when message is received**.

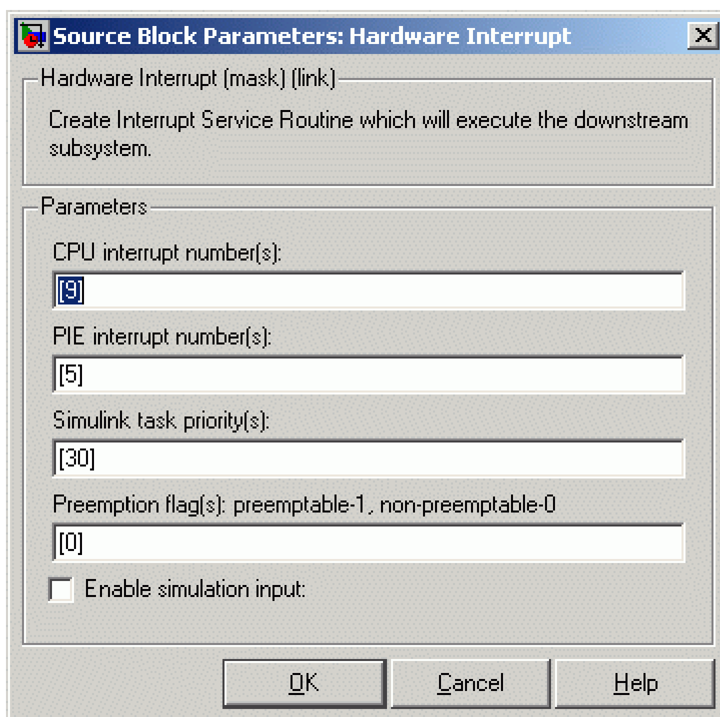


3 Set the **Sample Time** of the eCAN Receive block to -1 since the block will be triggered by the ISR, as shown in the preceding figure.

4 Add the C281x Hardware Interrupt block to your model.



5 The eCAN interrupt on C281x chips is on CPU line 9 and PIE line 5 for module 0. These parameters can be found in the C281x Hardware Interrupt block, C281x Peripheral Interrupt Vector Values figure. Set the hardware interrupt parameters **CPU interrupt number(s):** to 9, and **PIE interrupt number(s):** to 5 as shown in the following figure.



6 Connect the output of the Hardware Interrupt block to the function-call subsystem containing the eCAN block.

At execution time, when a new eCAN message is received, the eCAN interrupt is triggered, and the code you placed in the function-call subsystem is executed. In this example, the eCAN Receive block is placed in the function-call subsystem, which means that the message is read and is passed to the rest of the code.

For more information, see the section on Asynchronous Support in the Simulink Coder documentation.

Sharing General Purpose Timers between C281x Peripherals

In this section...
"Example 1" on page 55-13
"Example 2" on page 55-17

TMS320x281x DSP devices have four General Purpose (GP) timers. Each Event Manager (EV) module includes two GP timers:

- EVA includes GP Timer 1 and GP Timer 2.
- EVB includes GP Timer 3 and GP Timer 4.

You can use the GP Timers independently or to operate peripherals associated with the EV Manager, such as PWM, QEP, and CAP.

The following table describes the timer-peripheral mapping of the c281xlib block library.

GP Timer Use for C281x Peripheral Blocks

	GP Timer 1	GP Timer 2	GP Timer 3	GP Timer 4
PWM1-PWM6	✓			
PWM7-PWM12			✓	
QEP1-QEP2		✓		
QEP3-QEP4				✓
CAP1-CAP3	✓	✓		
CAP4-CAP6			✓	✓

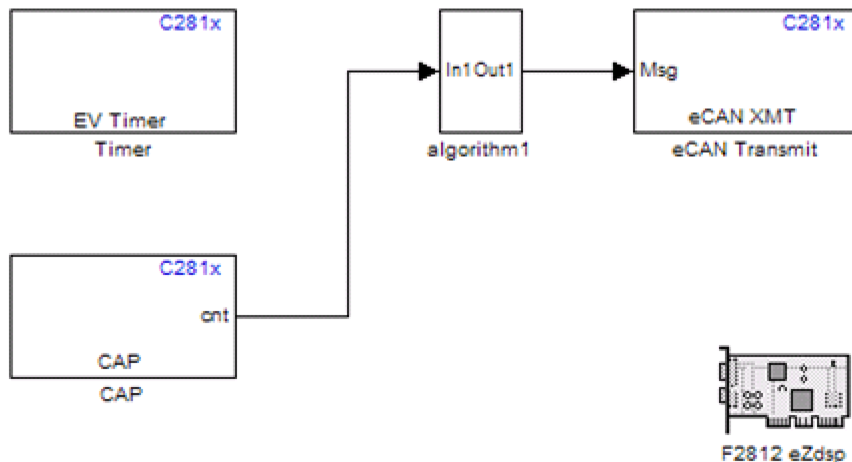
Each PWM and QEP peripheral has access to only one timer, while each CAP peripheral has access to two timers. In the PWM and QEP blocks, you can set the **Module** option to `A` or `B` to determine which unique timer-peripheral combination the block configures. By comparison, in the CAP block, you can use the **Time base** option to select one of two timers for each CAP peripheral.

Each GP timer is available to multiple peripherals. For example:

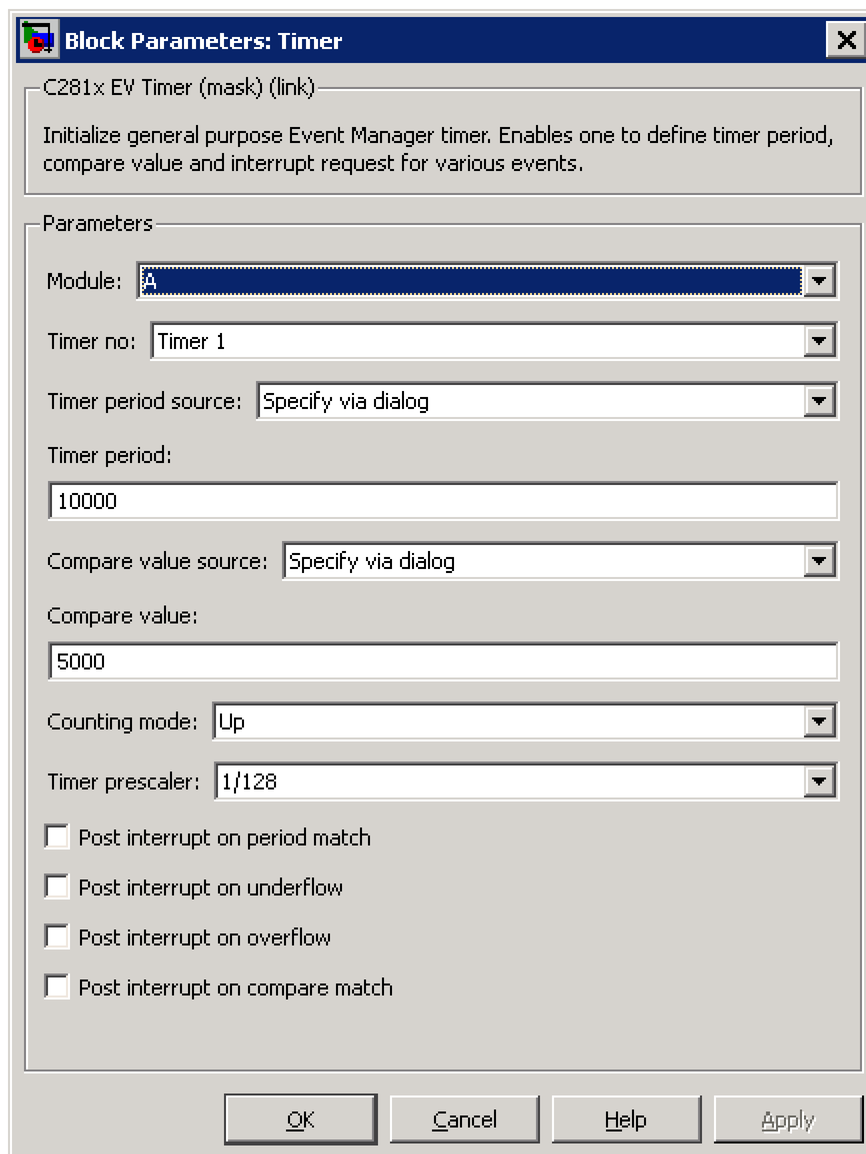
- PWM1-PWM6 and CAP1-CAP3 share GP Timer 1
- PWM7-PWM12 and CAP4-CAP6 share GP Timer 3
- QEP1-QEP2 and CAP1-CAP3 share GP Timer 2
- QEP3-QEP4 and CAP4-CAP6 share GP Timer 4

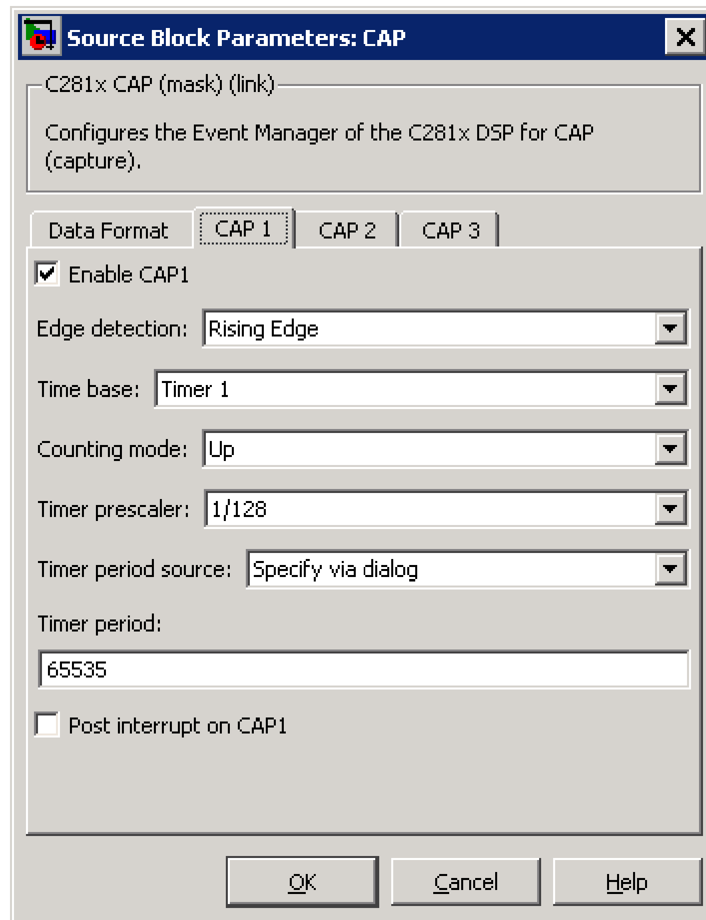
The PWM, QEP, CAP, and Timer blocks each provide independent access to key timer registers. If the blocks in your model share a specific GP timer, ensure that all the timer-related settings are compatible. If the peripheral settings for a shared timer are not compatible, the software generates an error when you update the model or generate code.

Example 1

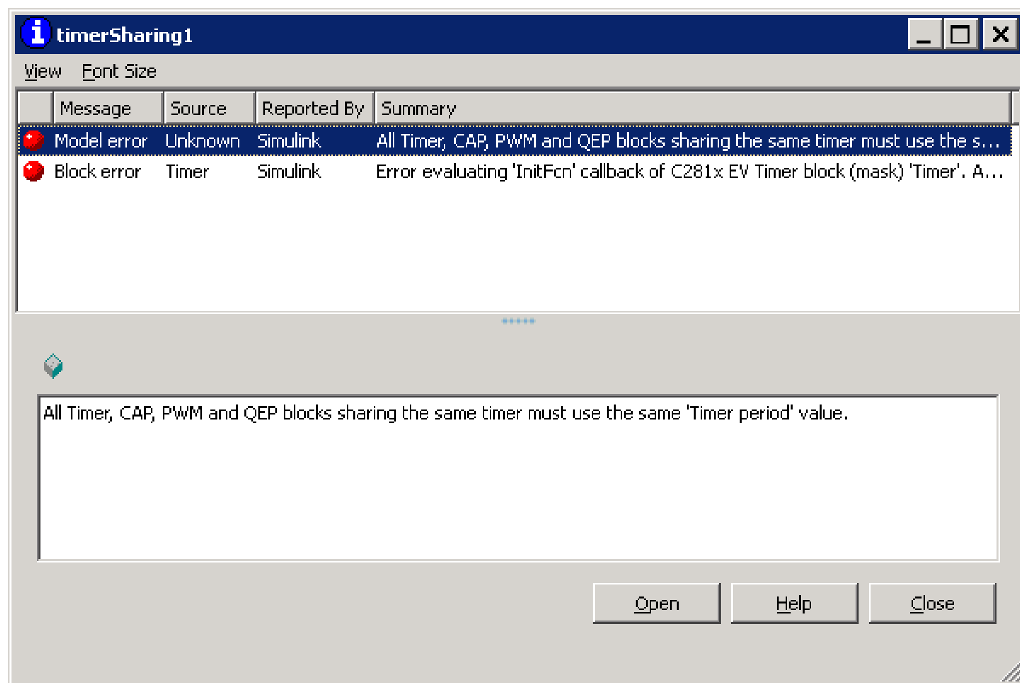


The model contains Timer and CAP blocks that both use Timer 1 (GP Timer 1).


Block Parameters: Timer
C281x EV Timer (mask) (link)
Initialize general purpose Event Manager timer. Enables one to define timer period, compare value and interrupt request for various events.
Parameters
Module:
A
Timer no:
Timer 1
Timer period source:
Specify via dialog
Timer period:
10000
Compare value source:
Specify via dialog
Compare value:
5000
Counting mode:
Up
Timer prescaler:
1/128
Post interrupt on period match
Post interrupt on underflow
Post interrupt on overflow
Post interrupt on compare match
OK
Cancel
Help
Apply

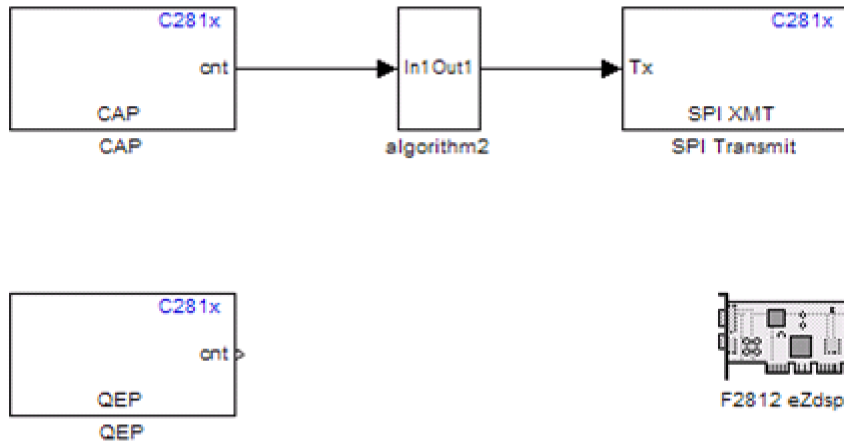


Both blocks have the same values for **Timer prescaler** and **Counting mode**. However, each block has different values for **Timer period**. The value of **Timer period** for Timer 1 is 65535 in the CAP block and 10000 in the Timer block.

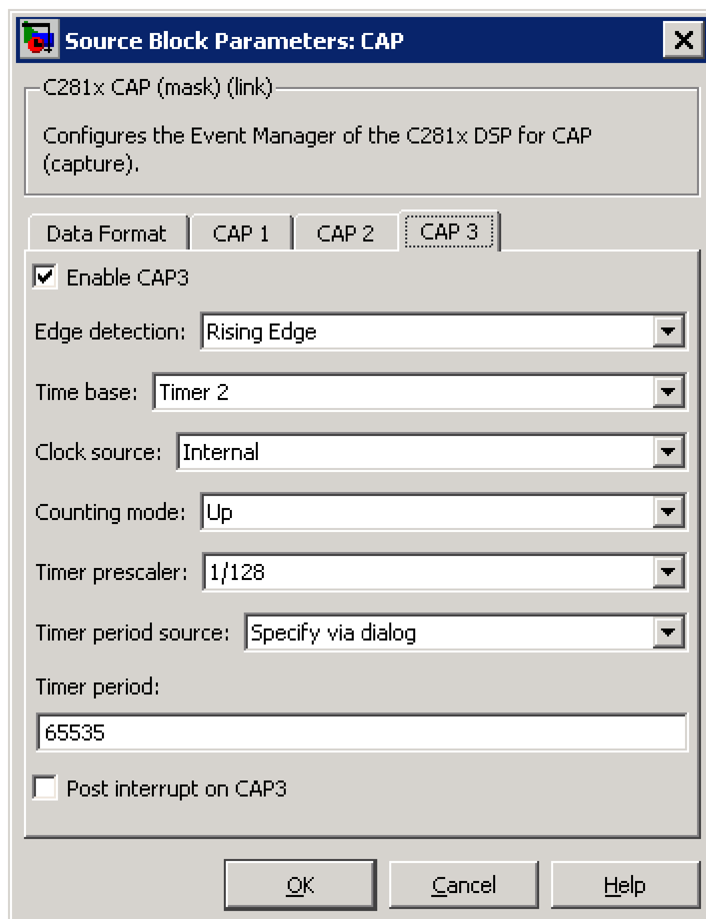


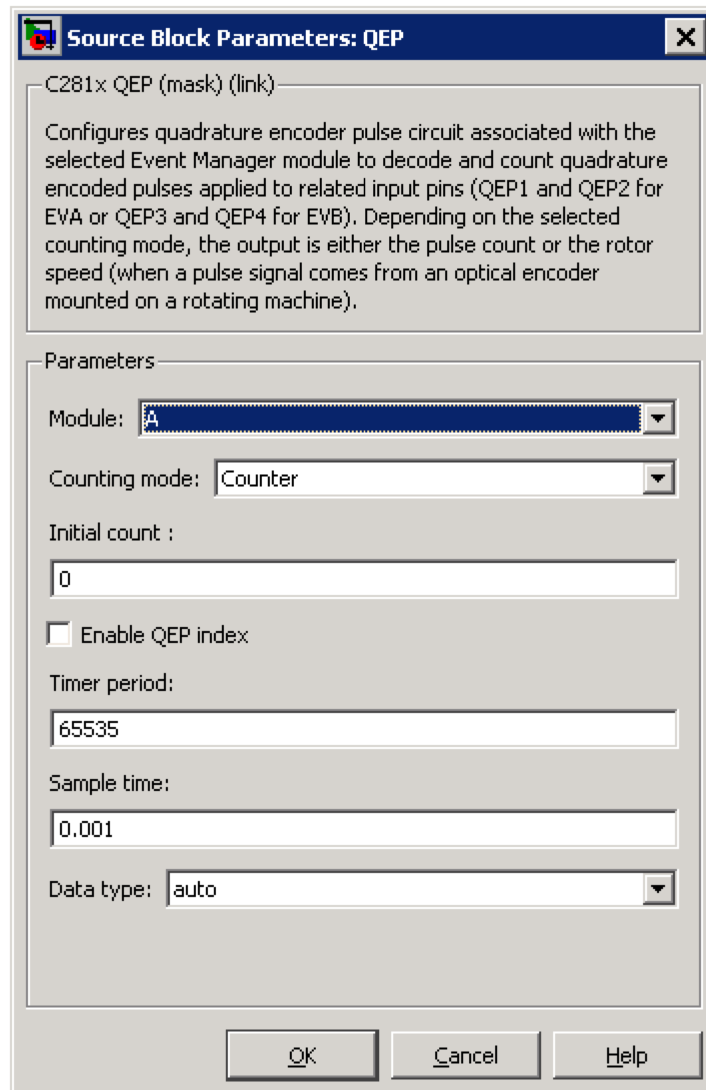
Since both blocks configure the same timer, and settings conflict, the software generates an error when you update the model.

Example 2

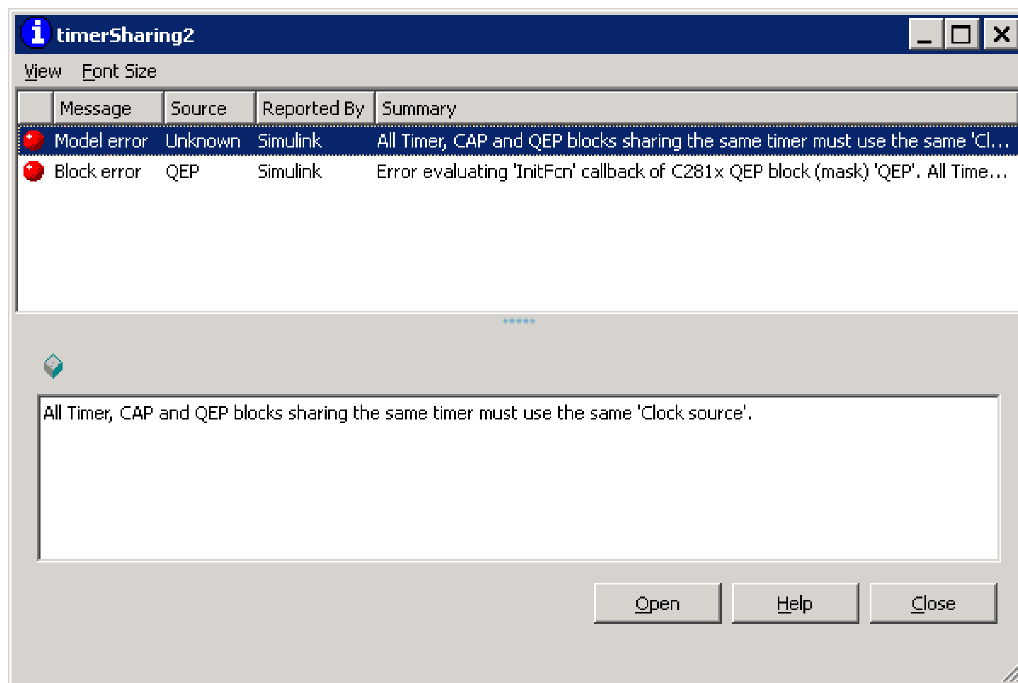


The model contains QEP and CAP blocks that both use Timer 2. In the CAP block, the **Time base** option shows which timer the block uses. In the QEP block, setting **Module** to A configures the block to use QEP1–QEP2. GP Timer Use for C281x Peripheral Blocks on page 55-12 shows that QEP1–QEP2 use Timer 2.


Source Block Parameters: CAP
C281x CAP (mask) (link)
Configures the Event Manager of the C281x DSP for CAP (capture).
Data Format
CAP 1
CAP 2
CAP 3
Enable CAP3
Edge detection:
Rising Edge
Time base:
Timer 2
Clock source:
Internal
Counting mode:
Up
Timer prescaler:
1/128
Timer period source:
Specify via dialog
Timer period:
65535
Post interrupt on CAP3
OK
Cancel
Help



Currently, both blocks define different clock sources for Timer 2. The CAP block uses `Internal` as a **Clock source**. The QEP block, which does not have a configurable **Clock source** setting, uses the QEP circuit as a clock source. If you build the model, the software generates the following error message.



To avoid generating errors when you build the model, change **Clock source** in the CAP block to `QEP device`.

Overview of Creating Models for C2000 Processors

In this section...

"Accessing the Embedded Coder Block Library" on page 55-21

"Building Your Model" on page 55-21

Accessing the Embedded Coder Block Library

After you have installed the supported development board, start MATLAB.

You can open the `c2000lib` blockset in the Simulink library browser, or by typing the following command at the MATLAB command prompt:

```
c2000lib
```

Create your real-time model for your application the same way you create any other Simulink model. Select blocks to build your model from the following sources or products:

- The Target Preferences library block (for setting target and application preferences)
- The appropriate libraries in the `c2000lib` block library (for handling input and output functions for on your target hardware)
- Simulink Coder software
- Discrete time blocks from Simulink
- Any other blockset that meets your needs and operates in the discrete time domain

Building Your Model

With this configuration, you can generate a real-time executable and download it to your TI development board by clicking **Generate Code** on the **Code Generation** pane. Simulink Coder software automatically generates C code and inserts the I/O device drivers as specified by the hardware blocks in your block diagram, if any. These device drivers are inserted in the generated C code.

During the same build operation, block parameter dialog box entries are combined into a project file for CCS for your TI C2000 board. If you selected the `Build and execute` build action in the configuration settings, the TI cross-compiler builds an executable file. After automatically downloading the executable file to the target, the build process runs the file on the board's DSP.

Note After using the run-time `Build` option to generate and build code for your application, you must perform the following reset sequence before you can run that code on your board. If you want to rerun your application manually once it has been generated, you must also use this procedure.

F2812, F2808, and F28335 eZdsp Reset Sequence

1. Reset the board CPU.
2. Load your code onto the target.
3. Run your code on the target.

Using the c2000lib Blockset

In this section...

"Introduction" on page 55-23

"Hardware Setup" on page 55-23

"Starting the c2000lib Library" on page 55-24

"Setting Up the Model" on page 55-24

"Adding Blocks to the Model" on page 55-26

"Generating Code from the Model" on page 55-28

Introduction

This section uses an example to demonstrate how to create a Simulink model that uses Embedded Coder blocks to target your board. The example creates a model that performs PWM duty cycle control via pulse width change. It uses the C2812 ADC block to sample an analog voltage and the C2812 PWM block to generate a pulse waveform. The analog voltage controls the duty cycle of the PWM and you can observe the duty cycle change on the oscilloscope. This model is also provided in the Demos library. The model in the Demos library also includes a model simulation.

Hardware Setup

The following hardware is needed for this example:

- Spectrum Digital eZdsp F2812
- Function generator
- Oscilloscope and probes

To connect the hardware:

1. Connect the function generator output to the ADC input ADCINA0 on the eZdsp F2812.

2. Connect the output of PWM1 on the eZdsp F2812 to the analog input of the oscilloscope.

3 Connect VREFLO to AGND on the eZdsp F2812. See the section on the Analog Interface in Chapter 2 of the *eZdsp™ F2812 Technical Reference*, available from the Spectrum Digital Web site at `http://c2000.spectrumdigital.com/ezf2812/`

Starting the c2000lib Library

At the MATLAB prompt, type the following command:

```
c2000lib
```

This command open the `c2000lib` library blockset, which contains libraries of blocks designed for targeting your board.

Setting Up the Model

Preliminary tasks for setting up a new model include adding a Target Preferences block, setting or verifying Target Preferences, and setting the simulation parameters.

1. In MATLAB, open the Simulink Library Browser.

2. Search for Target Preferences.

3. Right-click the Target Preferences block and select `Add to a new model`. This opens a new model with the Target Preferences block in it.

4. Click **Yes** to allow automatic setup. The following settings are made, referenced in the table below by their locations in the **Simulation > Configuration Parameters** dialog box:

Pane	Field	Setting
Solver	**Stop time**	`10`
Solver	**Type**	`Fixed-step`
Data Import/Export	**Save to workspace - Time**	`tout`
Data Import/Export	**Save to workspace - Output**	`yout`

Pane	Field	Setting
Hardware Implementation	**Device type**	`C2000`
Code Generation	**Target selection - System target file**	`idelink_grt.tlc` or `idelink_ert.tlc`

Note Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

Note One Target Preferences block must be in each target model at the top level. It does not connect to any other blocks, but stands alone to set the Target Preferences for the model.

5 From your model's main menu, select **Simulation > Configuration Parameters** to verify and set the simulation parameters for this model. Parameters you set in this dialog box belong to the model you are building. They are saved with the model and stored in the model file. Refer to your Simulink documentation for information on the Configuration Parameters dialog box.

6 Use the **Code Generation** pane to set options for the real-time model. Refer to your "Simulink Coder" documentation for detailed information on the **Code Generation** pane options.

7 Use the **Browse** button to locate and select a target configuration file, `idelink_grt.tlc` or `idelink_ert.tlc`. When you do this, your coder product chooses the appropriate system target file, and `make` command.

8 Set the configuration parameters by typing **Ctrl-E** and adjust these parameters.

Adding Blocks to the Model

1 Open or double-click the C281x library, `c281xlib`.

2 Drag the ADC block into your model. Double-click the ADC block in the model and set **Sample time** to `64/80000`.

3 Drag the PWM block into your model. Double-click the PWM block in the model and set the following parameters.

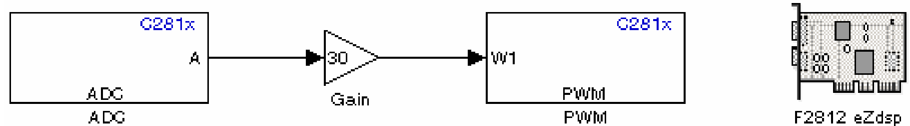
Pane	Field	Parameter
Timer	**Module**	`A`
	Waveform period source	`Specify via dialog`
	Waveform period units	`Clock cycles`
	Waveform period	`64000`
	Waveform type	`Asymmetric`
Outputs	**Enable PWM1/PWM2**	Selected
	Duty cycle source	`Input port`
Logic	**PWM1 control logic**	`Active high`
	PWM2 control logic	`Active low`

Pane	Field	Parameter
Deadband	**Use deadband for PWM1/PWM2**	Selected
	Deadband prescaler	`16`
	Deadband period	`12`
ADC Control	**ADC start event**	`Period interrupt`

4 Enter `Simulink` at the MATLAB command line to open the Simulink Library browser. Drag a Gain block from the Math Operations library into your model. Double-click the Gain block in the model and set the following parameters in the Function Block Parameters dialog box. Click **OK**.

Pane	Field	Parameter
Main	**Gain**	`30`
	Multiplication	`Element-wise(K.*u)`
	Sample time	`-1`
Signal Attributes	**Output data type mode**	`uint(16)`
	Integer rounding mode	`Floor`
Parameter Attributes	**Parameter data type mode**	`Inherit from input`

5 Connect the ADC block to the Gain block and the Gain block to the PWM block.



Generating Code from the Model

This section summarizes how to generate code from your real-time model.

You start the automatic code generation process from the Simulink model window by clicking **Generate code** in the **Code Generation** pane of the Configuration Parameters dialog. Other ways of starting the code generation process are by clicking the **Incremental Build** button on the toolbar of your model, or by pressing the keyboard shortcut, **Ctrl+B**, while your model is open and in focus.

Note In CCS, you see your project with the files in place in the folder structure.

Configuring Timing Parameters for CAN Blocks

In this section...

"The CAN Blocks" on page 55-29

"Setting Timing Parameters" on page 55-29

"Parameter Tuning and Signal Logging" on page 55-34

The CAN Blocks

The bit rate of these four CAN blocks cannot be set directly:

C281x eCAN Receive
C281x eCAN Transmit
C280x/C28x3x eCAN Receive
C280x/C28x3x eCAN Transmit

Setting Timing Parameters

- "Accessing the Timing Parameters" on page 55-29
- "Determining Timing Parameter Values" on page 55-32
- "CAN Bit Timing Example" on page 55-33

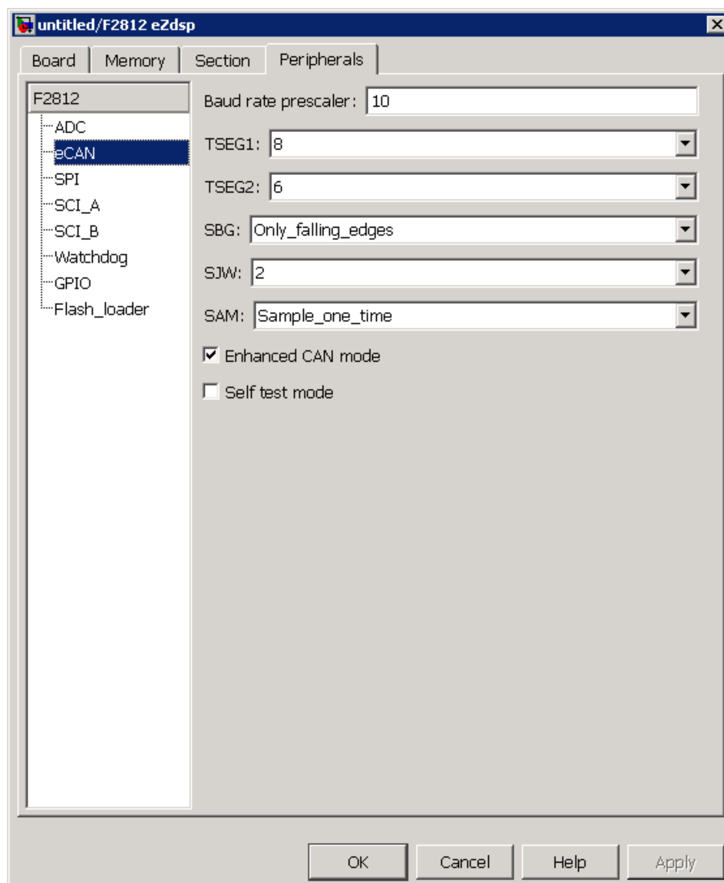
Accessing the Timing Parameters

To set the bit rate for a block whose bit rate cannot be set directly:

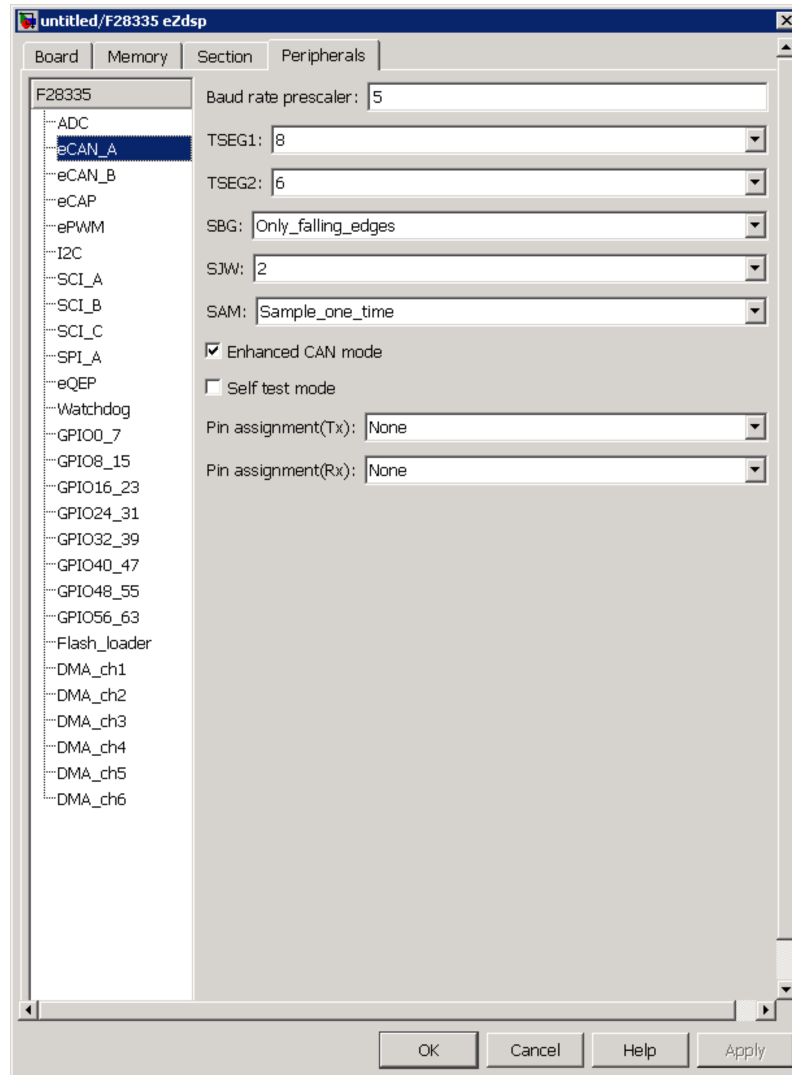
1. Double click the Target Preferences block in your model. This opens the **Target Preferences** dialog box.

2. Under the **Peripherals** tab, use the **TSEG1**, **TSEG2**, and **BaudRatePrescaler (BRP)** parameters to set the bit rate.

For example, the **Target Preferences** block for the F2812 eZdsp, this dialog box is shown in the following figure.



The C280x/C28x3x blocks have two independent eCAN modules, as shown by the Target Preferences Setup dialog box.



The following sections describe the series of steps and rules that govern the process of setting these timing parameters.

Determining Timing Parameter Values

To determine the appropriate values for the timing parameters, complete the following steps:

1. Determine the CAN Bitrate specification based on your application.
2. Determine the frequency of the CAN module clock. For example:
   - 100 MHz for the F2808 (Same as SYSCLKOUT)
   - 150 MHz for the F2812 (Same as SYSCLKOUT)
   - 75 MHz for the F28x3x (150 MHz SYSCLKOUT/2)
3. Estimate the value of the **BaudRatePrescaler (BRP)**.
4. Solve this equation for BitTime:

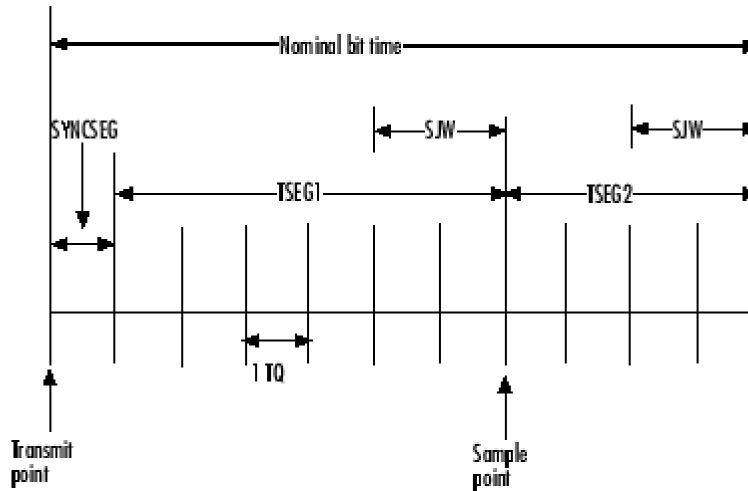
   BitTime = CAN module clock frequency/(BRP * Bitrate)
5. Solve this equation for Bitrate:

   Bitrate = CAN module clock frequency/(BRP * BitTime)
6. Estimate values of **TSEG1** and **TSEG2** that satisfy the following equation:

   $$BitTime = TSEG1 + TSEG2 + 1$$
7. Use the following rules to determine the values of **TSEG1**and **TSEG2**:

   TSEG1 >= **TSEG2**
   IPT (Information Processing Time) = 3/**BRP**
   IPT <= **TSEG1** <= 16 TQ
   IPT <= **TSEG2** <= 8 TQ
   1 TQ <= **SJW** <= min (4 TQ, **TSEG2**)
   where IPT is Information Processing Time, TQ is Time Quanta, and **SJW** is Synchronization Jump Width, also set in the **Target Preferences** dialog box. .
8. Iterate steps 4 through 7 until the values selected for TSEG1, TSEG2, and BRP meet all of the criteria.

The following illustration shows the relationship between the eCAN bit timing parameters.



CAN Bit Timing Example

Assume that SYSCLKOUT = 150 MHz, and a bit rate of 1 Mbits/s is required.

1 Set the BRP to 10. Then substitute the values of bit rate, BRP, and SYSCLKOUT into the following equation, solving for BitTime:

$$BitTime = SYSCLKOUT/(BRP * Bitrate)$$
$$BitTime = 150/(10*1) = 15TQ$$

2 Set the values of **TSEG1** and **TSEG2** to 8TQ and 6TQ respectively. Substitute the values of *BitTime* from the previous equation, and the chosen values for *TSEG1* and **TSEG2** into the following equation:

$$BitTime = TSEG1 + TSEG2 + 1$$
$$15TQ = 8TQ + 6TQ + 1$$

3 Finally, check the selected values against the rules:

IPT = 3/**BRP** = 3/10 = .3

IPT <= **TSEG1** <= 16 TQ True! .3<=8TQ<=16TQ
IPT <= **TSEG2** <= 8TQ True! .3 <= 6TQ <= 8TQ
1TQ <= **SJW** <= min(4TQ, **TSEG2**) which means that **SJW** can be set to either 2, 3, or 4

4 All chosen values satisfy the criteria, so no further iteration is necessary.

The following table provides common timing parameter settings for typical values of Bit Rate and SYSCLKOUT = 150 MHz. This clock frequency is the maximum for the C281x blocks.

Bit Rate	TSEG1	TSEG2	Bit Time	BRP	SJW
.5 Mbit/s	8	6	15	20	2
1 Mbit/s	8	6	15	10	2
2 Mbit/s	8	6	15	5	2

The following table provides common timing parameter settings for typical values of Bit Rate and SYSCLKOUT = 100 MHz. This clock frequency is the maximum for the C280x/C28x3x blocks.

Bit Rate	TSEG1	TSEG2	Bit Time	BRP	SJW
.5	6	3	10	20	2
1	5	4	10	10	2
2	6	3	10	5	2

Parameter Tuning and Signal Logging

- “Overview” on page 55-35
- “Using External Mode” on page 55-35
- “Using a Third Party Calibration Tool” on page 55-45

Overview

Embedded Coder software supports parameter tuning and signal logging either using Simulink external mode or with a third party calibration tool. In both cases the model must include a CAN Calibration Protocol block.

Using External Mode

The Simulink external mode feature enables you to log signals and tune parameters without requiring a calibration tool. This section describes the steps for converting a model to use external mode.

External mode is supported using the CAN Calibration Protocol block and ASAP2 interface. The CAN Calibration Protocol block is used to communicate with the target, download parameter updates, and upload signal information. The ASAP2 interface is used to get information about where in the target memory a parameter or signal lives.

Note You must configure the host-side CAN application channel. See "Configuring the Host Vector CAN Application Channel" on page 55-36.

To prepare your model for external mode, follow these steps:

1. Add a CCP driver block.

2. Add a Switch External Mode Configuration Block (for ease of use; you can also make changes manually).

3. Identify signals you want to tune, and associate them with `Simulink.Parameter` or `canlib.Parameter` objects with `ExportedGlobal` storage class. It is important to set the data type and value of the object. See "Using Supported Objects and Data Types" on page 55-37.

4. Identify signals you want to log, and associate them with `canlib.Signal` objects. It is important to set the data type of the `canlib.Signal`. See "Using Supported Objects and Data Types" on page 55-37.

   For information about visualizing logged signal data, see "Viewing and Storing Signal Data" on page 55-40.

5. Load the `Simulink.Parameter` or `canlib.Parameter` and `canlib.Signal` data objects into the base workspace.

6. Configure the model for building by double-clicking the Switch External Mode Configuration block. In the dialog box, select **Building an executable**, and click **OK**.

7. Build the model, and download the executable to the target

8. After downloading the executable to the target, you can switch the model to external mode by double-clicking the Switch External Mode Configuration Block. In the dialog box that appears, select **External Mode**, and click **OK**.

9. You can now connect to the target using external mode by clicking the **Connect** button.

10. If you have set up tunable parameters, you can now tune them. See "Tuning Parameters" on page 55-39.

If you do not want to use the Switch External Mode Configuration block, you can configure for building and then external mode manually. For instructions, see "Manual Configuration For External Mode" on page 55-43.

See the following topics for more information:

- "Configuring the Host Vector CAN Application Channel" on page 55-36
- "Using Supported Objects and Data Types" on page 55-37
- "Tuning Parameters" on page 55-39
- "Viewing and Storing Signal Data" on page 55-40
- "Manual Configuration For External Mode" on page 55-43
- "Limitations" on page 55-44

Configuring the Host Vector CAN Application Channel. External mode expects that the host-side CAN connection is using the `'MATLAB 1'` application channel. To configure the application channel used by the Vector CAN drivers, enter the following at the MATLAB command line:

```
TargetsComms_VectorApplicationChannel.configureApplicationChannels
```

The Vector CAN Configuration tool appears. Use this tool to configure your host-side CAN channel settings.

If you try to connect using an application channel other than `'MATLAB 1'`, then you see the following warning in the command window:

```
Warning:
It was not possible to connect to the target using CCP.
An error occurred when issuing the CONNECT command.
```

If you have not already installed the Vector CAN drivers, you will get the following error message:

```
??? Error using ==>
TargetsComms_VectorApplicationChannel.TargetsComms_VectorApplicationChannel>
TargetsComms_VectorApplicationChannel.configureApplicationChannels at 40
Unable to launch the application channel configuration utility.
The "vcanconf" utility was not found on the Windows System Path.
To fix this error, make sure the required CAN drivers are installed on this computer;
refer to the product documentation for details.
```

If you want to use CAN to transmit or receive CAN messages between your host PC and your target, you require Vector-Informatik CAN hardware supported by the Vector CAN Driver Library. You must install the correct driver libraries to support profiling, downloading, and external mode.

Note For CANcaseXL, you must install *both* the Vector XL-driver library and Vector CAN Driver Library `vcand32.dll`.

For older CAN hardware, you must install the Vector CAN Driver Library `vcand32.dll`.

Make sure that the library, `vcand32.dll`, is placed in the Windows `system32` folder.

Using Supported Objects and Data Types. Supported objects:

- `Simulink.Parameter` or `canlib.Parameter` for parameter tuning

- `canlib.Signal` for signal logging

Supported data types:

- uint8, int8
- uint16, int16
- uint32, int32
- single

You need to define data objects for the signals and parameters of interest for ASAP 2 file generation. For ease of use, create a MATLAB file to define the data objects, so that you only have to set up the objects once.

To set up tunable parameters and signal logging:

1. Associate the parameters to be tuned with `Simulink.Parameter` or `canlib.Parameter` objects with ExportedGlobal storage class. It is important to set the data type and value of the parameter object. See the following code for an example of how to create such a Simulink.Parameter object for tuning:

   ```
   stepSize = Simulink.Parameter;
   stepSize.DataType = 'uint8';
   stepSize.RTWInfo.StorageClass = 'ExportedGlobal';
   stepSize.Value = 1;
   ```

2. Associate the signals to be logged with canlib.Signal objects. It is important to set the data type of the canlib.Signal. The following code example shows how to declare such a canlib.Signal object for logging:

   ```
   counter = canlib.Signal;
   counter.DataType = 'uint8';
   ```

3. Associate the data objects you defined in the MATLAB file with parameters or signals in the model. For the previous code examples, you could set the **Constant value** in a Source block to `stepSize`, and set a **Signal name** to `counter` in the Signal Properties dialog box. Remember that `stepSize` and `counter` are data objects defined in the code.

Tuning Parameters. To tune a parameter, follow these steps:

1. Set `*dataobject*.value` in the workspace while the model is running in external mode. For example, to tune the parameter `stepSize` (that is, to change its value) from 1 to 2, enter the following at the command line:

   ```
   stepSize.value = 2
   ```

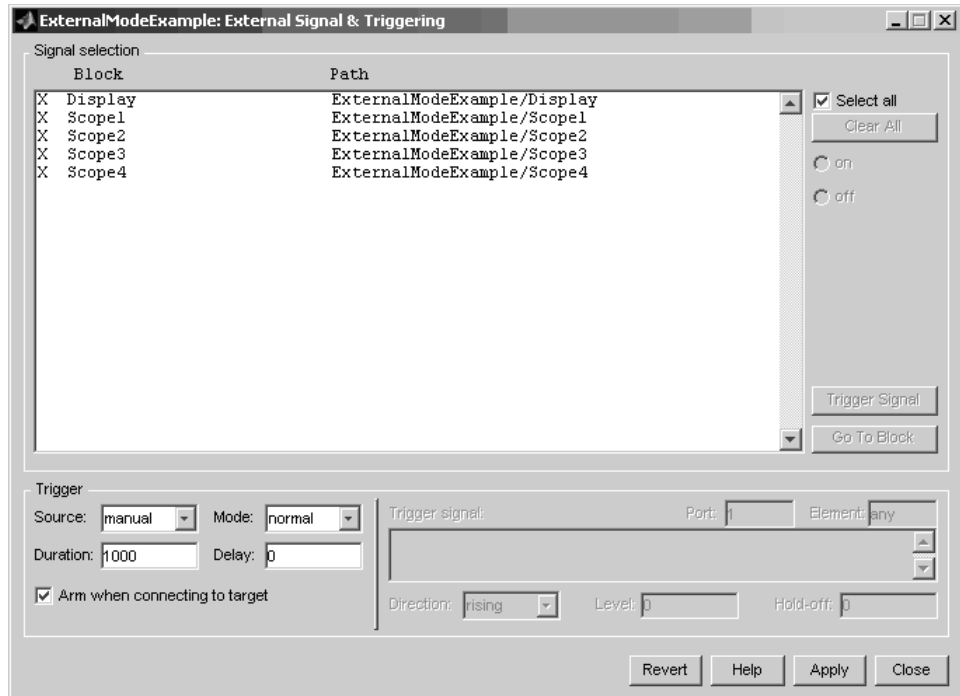
You see output similar to the following:

```
stepSize =

Simulink.Parameter (handle)
        RTWInfo: [1x1 Simulink.ParamRTWInfo]
    Description: ''
       DataType: 'uint8'
            Min: -Inf
            Max: Inf
       DocUnits: ''
          Value: 2
     Complexity: 'real'
     Dimensions: [1 1]
```

2 Return to your model, and update the model (press **Ctrl+D**) to apply the changed parameter.

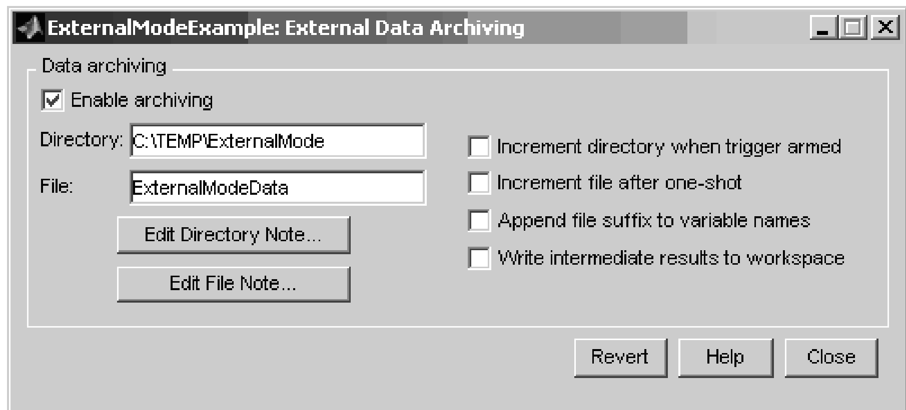
Viewing and Storing Signal Data. To view the logged signals attach a supported scope type to the signal (see “Limitations” on page 55-44 for supported scope types).

Select which signals you want to log by using the External Signal & Triggering dialog box. Access the External Mode Control Panel from the Tools menu, and click the **Signal & Triggering** button. By default, all displays appear as selected to be logged, as shown in the following example. Edit these settings if you do not want to log all displays. Individual displays can be selected manually.



Storing signal data for further analysis. It is possible to store the logged data for further analysis in MATLAB.

1 To use the Data Archiving feature of external mode, click **Data Archiving** in the External Mode Control Panel. The External Data Archiving dialog box appears.

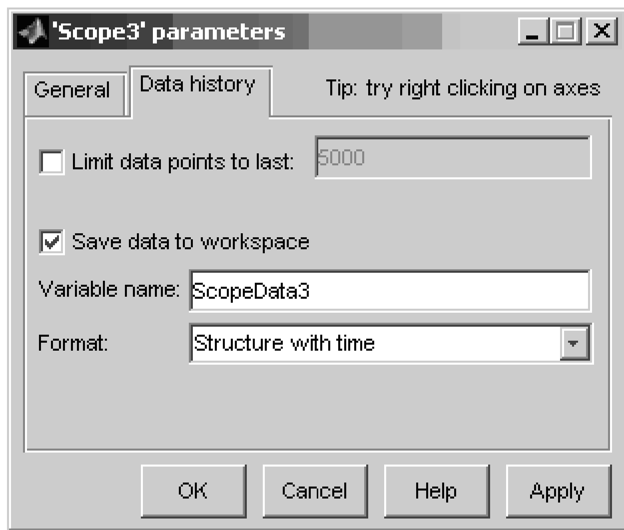


a Select the check box **Enable archiving**

b Edit the **Folder** and **Filename** and any other desired settings.

c Close the dialog box.

2 Open the Scope parameters, and select the check box **Save data to workspace**.



3 You may want to edit the **Variable name** in the edit box. The data that is displayed on the scope at the end of the external mode session is available in the workspace with this variable name.

The data that was previously displayed in the scope is stored in `.mat` files as previously setup using Data Archiving.

For example, at the end of an external mode session, the following variable and files could be available in the workspace and current folder:

- A variable `ScopeData5` with the data currently displayed on the scope:

```
ScopeData5

ScopeData5 =

         time: [56x1 double]
      signals: [1x1 struct]
    blockName: 'mpc555rt_ccp/Scope1'
```

- In the current folder, .mat files for the three previous **Durations** of scope data:

```
ExternalMode_0.mat
ExternalMode_2.mat
ExternalMode_1.mat
```

Manual Configuration For External Mode. As an alternative to using the Switch External Mode Configuration block, you can configure models manually for build and execution with external mode.

To configure a model to be built for external mode:

1 Select **Inline parameters** (under Optimization in the Configuration Parameters dialog box). The **Inline parameters** option is required for ASAP2 generation.

2 Select **Normal** simulation mode (in either the Simulation menu, or the drop-down list in the toolbar).

3 Select `ASAP2` as the **Interface** (under **Code Generation**, **Interface**, in the **Data Exchange** pane, in the Configuration Parameters dialog box).

After you build the model, you can configure it for external mode execution:

1 Make sure **Inline parameters** are selected (under **Optimization** in the Configuration Parameters dialog box). The **Inline parameters** option is required for external mode.

2 Select **External** simulation mode (in either the **Simulation** menu, or the drop-down list in the toolbar).

3 Select `External mode` as the **Interface** (under **Code Generation**, **Interface**, in the **Data Exchange** pane, in the Configuration Parameters dialog box).

Limitations. Multiple signal sinks (e.g. scopes) are not supported.

Only the following kinds of scopes are supported with External Mode Logging:

- Simulink Scope block
- Simulink Display block
- Viewer type: scope — To use this option, right-click a signal in the model, and select **Create & Connect Viewer > Simulink > Scope**. The other scope types listed there are not supported (e.g., floating scope).

  Before connecting to external mode, you also need to right-click the signal, and select **Signal Properties**. In the dialog box, select the **Test point** check box, and click **OK**.

GRT is supported but only for parameter tuning.

It is not possible to log signals with sample rates in excess of 10 kHz.

Subsystem builds are not supported for external mode, only top-level builds are supported.

Logging and tuning of nonscalars is not supported. It is possible to log nonscalar signals by breaking the signal down into its scalar components. For an example of how to do this signal deconstruction, see the CCP demo models, which use a Demux and Signal Conversion block with contiguous copy.

Logging and tuning of complex numbers is not supported. It is possible to work with complex numbers by breaking the complex number down into its real and imaginary components. This breakdown can be performed using the following blocks in the Simulink Math Operations library: Complex to Real-Imag, Real-Imag to Complex, Magnitude-Angle to Complex, Complex to Magnitude-Angle.

Using a Third Party Calibration Tool

Embedded Coder allows an ASAP2 data definition file to be generated during the code generation process. This file can be used by a third party tool to access data from the real-time application while it is executing.

ASAP2 is a data definition standard by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description for data measurement, calibration, and diagnostic systems. Embedded Coder software lets you export an ASAP2 file containing information about your model during the code generation process.

Before you begin generating ASAP2 files with Embedded Coder software, see “Generating an ASAP2 File” in the product help for Simulink Coder. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

Select the ASAP2 option before the build process as follows:

1. Select **Simulation > Configuration Parameters**.

   The Configuration Parameters dialog box appears.

2. Select **Interface** (under **Code Generation)** in the tree.

3. Select the `ASAP2` option from the **Interface** drop-down menu, in the **Data exchange** frame.

4. Click **Apply**.

The build process creates an ASAM-compliant ASAP2 data definition file for the generated C code.

- The standard ASAP2 file generation does not include the memory address attributes in the generated file. Instead, it leaves a placeholder that must be replaced with the actual address by postprocessing the generated file.
- The map file options in the template project need to be set up a certain way for this procedure to work. If you have created your own template projects, and you do not have the correct settings, you see the following instructions:

```
Warning: It was not possible to do ASAP2 processing on your
.map file.This is because your IDE project template is not
configured to generate a .map file in the correct format.
To generate a .map file in the correct format you need to
setup the following options in your IDE project template:
Generate section map should be checked on
Generate register map should be checked off
Generate symbol table should be checked on
Format list file into pages should be checked off
Generate summary should be checked off
Page width should be equal to 132 characters
Symbol colums should be 1
You can change these options via Project -> Project Options
 -> Linker/Locator -> Map File -> Map File Format.
```

Embedded Coder software performs this postprocessing for you. To do this, it first extracts the memory address information from the map file generated during the link process. Secondly, it replaces the placeholders in the ASAP2 file with the actual memory addresses. This postprocessing is performed automatically and requires no additional input from you.

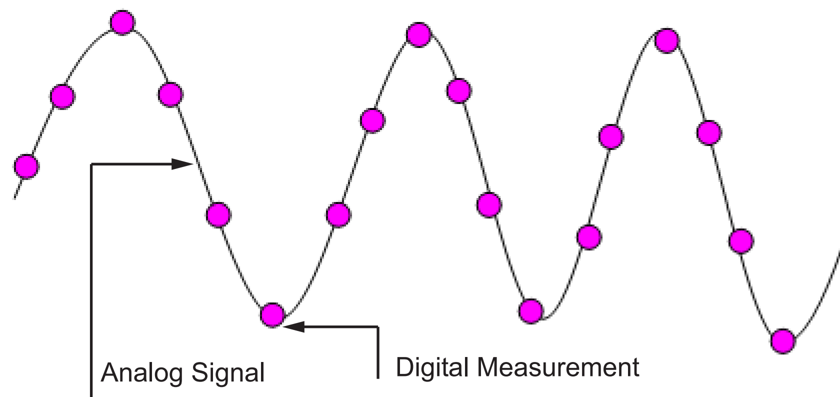
Configuring Acquisition Window Width for ADC Blocks

In this section...

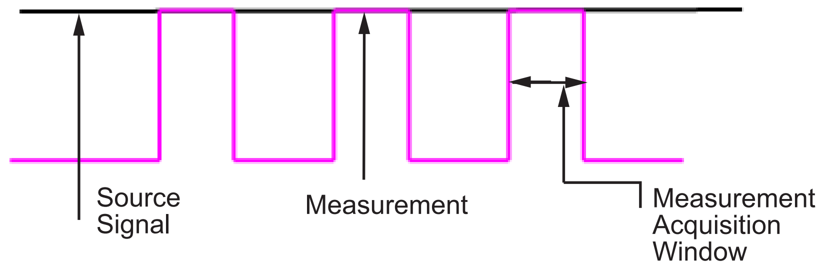
"What Is an Acquisition Window?" on page 55-47
"Configuring ADC Parameters for Acquisition Window Width" on page 55-49

What Is an Acquisition Window?

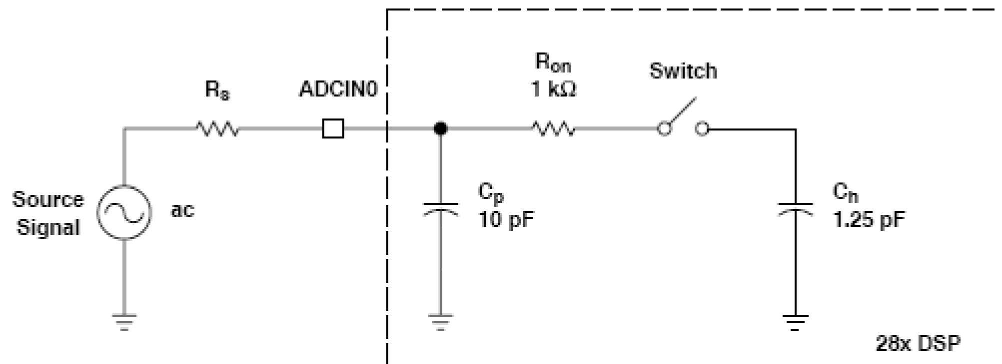
ADC blocks take a signal from an analog source and measure it with a digital device. The digital device does not measure in a continuous process, but in a series of discrete measurements, close enough together to approximate the source signal with the required accuracy.



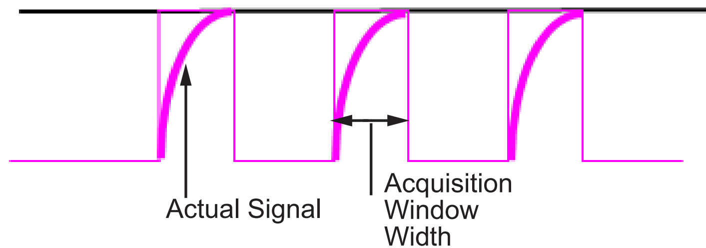
The digital measurement itself is not an instantaneous process, but is a measurement window, where the signal is acquired and measured, as shown in the following figure.



Ideally, as soon as the measurement window is opened, the actual signal coming in would be measured perfectly. In reality the signal does not reach its full magnitude immediately. The measurement process can be modeled by a circuit similar to the one shown in the following figure for the ADC found on the F2812 eZdsp



where the measurement circuit is characterized by a certain capacitance. In the preceding figure, when the switch is closed, the measurement begins. In this circuit, which is characterized by its capacitance, the signal received is not in a form of a step function as shown by the ideal measurement, but a ramp up to the true signal magnitude. The following figure shows what happens to the signal when the sampler switch is closed and the signal is received to be measured.



Because the signal acquisition is not instantaneous, it is very important to set a wide enough acquisition window to allow the signal to ramp up to full strength before the measurement is taken. If the window is too narrow, the measurement is taken before the signal has reached its full magnitude, resulting in erroneous data. If the window is too wide, the source signal itself may change, and the sampling may be too infrequent to reflect the actual value, also resulting in erroneous data. You must calculate the necessary width of the acquisition window based on the circuit characteristics of resistance and capacitance of your specific circuit. Then, using the ADC parameters described in the following section, you can configure the necessary acquisition window width.

Configuring ADC Parameters for Acquisition Window Width

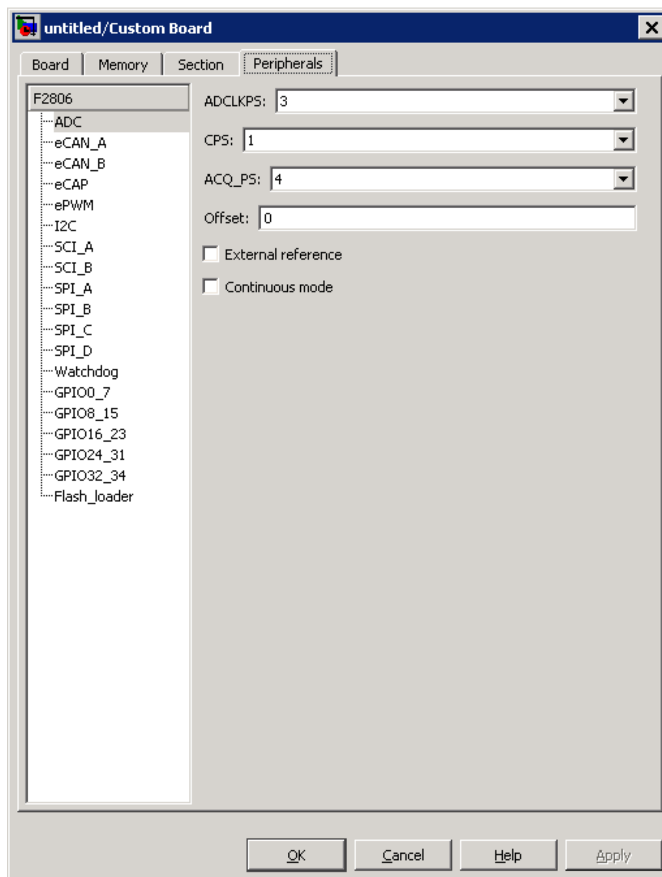
- "Accessing the ADC Parameters" on page 55-49
- "Examples" on page 55-51

Accessing the ADC Parameters

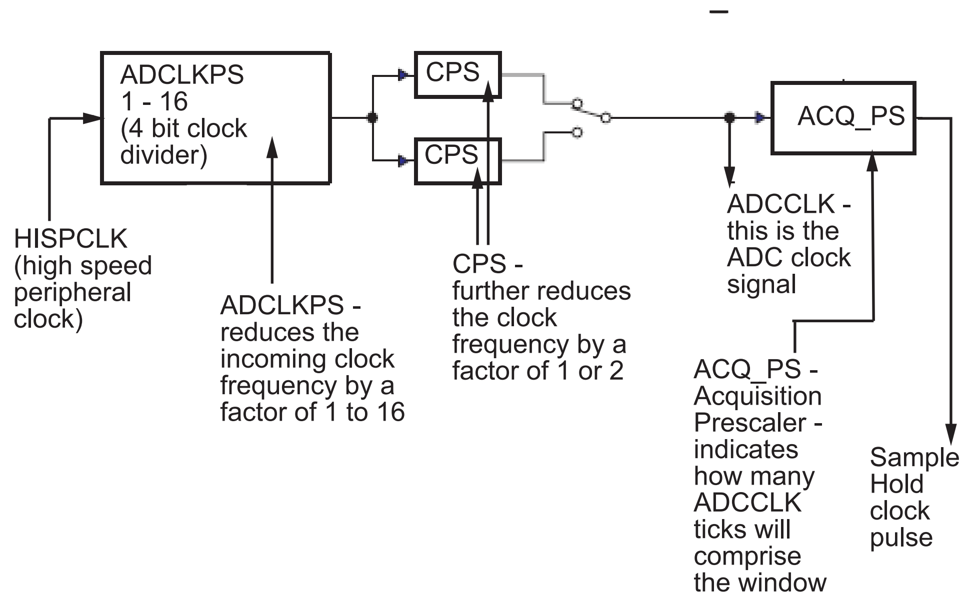
The ADC parameters can be set from the **Peripherals tab** of the Target Preferences block.

- You can set **ACQ_PS** — Acquisition Prescaler — to a value from 0 to 15. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 16.
- You can set **ADCLKPS** — AD Clock Prescaler — to a value from 0 to 15. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 16.

- You can set **CPS** — Clock Prescaler — to a value from 0 to 1. To obtain the actual value, increment the setting by 1. This produces an actual range from 1 to 2.



These three prescalers serve to reduce the speed of the clock and to set the acquisition window width. The following diagram shows how these prescalers are used.



In the preceding diagram, the high speed peripheral clock frequency is received and then divided by the **ADCLKPS**. The reduced clock frequency is then further divided by **CPS**. The resulting frequency is the **ADCCLK** signal. The value of **ACQ_PS** then determines how many **ADCCLK** ticks comprise one S/H (sample and hold) period, or in other words, the length of the acquisition window.

Examples

The following examples show how you can use ADC parameters to configure the acquisition window width:

Example 1:

If the HISPCLK = 30 MHz, and **ADCLKPS**=1 (which is a value of 2), the result is 15 MHz.

If **CPS**= 1 (which is a value of 2), then ADCCLK = 7.5 MHz.

If **ACQ_PS** = 0 (which is a value of 1), then the sample/hold period is 1 ADCCLK tick, or .1333 microseconds.

Example 2:

If the HISPCLK = 30 MHz, and **ADCLKPS**=1 (which is a value of 2), the result is 15 MHz.

If **CPS**= 1 (which is a value of 2), then ADCCLK = 7.5 MHz.

If **ACQ_PS** = 15 (which is a value of 16), then the sample/hold period is 16 ADCCLK ticks, or 2.1333 microseconds.

Note HISPCLK is set automatically for the user, and it is not possible to change the rate. For more information, see "High-Speed Peripheral Clock" on page 55-7

Using the IQmath Library

In this section...

"About the IQmath Library" on page 55-53

"Fixed-Point Numbers" on page 55-54

"Building Models" on page 55-59

About the IQmath Library

- "Introduction" on page 55-53
- "Common Characteristics" on page 55-54
- "References" on page 55-54

Introduction

The C28x IQmath Library blocks perform processor-optimized fixed-point mathematical operations. These blocks correspond to functions in the Texas Instruments C28x IQmath Library, an assembly-code library for the TI C28x family of digital signal processors.

Note The implementation of this library for the TI C28x processor produces the same simulation and code-generation output as the TI version of this library, but it does not use a global Q value, as does the TI version. The Q format is dynamically adjusted based on the Q format of the input data.

The IQmath Library blocks generally input and output fixed-point data types and use numbers in Q format. The C28x IQmath Library block reference pages discuss the data types accepted and produced by each block in the library. For more information, consult the "Fixed-Point Numbers" on page 55-54 and "Q Format Notation" on page 55-56 topics, as well as the Simulink Fixed Point product documentation, which includes more information on fixed-point data types, scaling, and precision issues.

You can use IQmath Library blocks with some core Simulink blocks and Simulink Fixed Point blocks to run simulations in Simulink models before generating code. Once you develop your model, you can generate equivalent code that is optimized to run on a TI C28x DSP. During code generation, a call is made to the IQmath Library for each IQmath Library block in your model to create target-optimized code. To learn more about creating models that include IQmath Library blocks and blocks from other blocksets, consult "Building Models" on page 55-59.

Common Characteristics

The following characteristics are common to all IQmath Library blocks:

- Sample times are inherited from driving blocks.
- Blocks are single rate.
- Parameters are not tunable.
- All blocks support discrete sample times.

To learn more about characteristics particular to each block in the library, see C28x IQmath for links to the individual block reference pages.

References

For detailed information on the IQmath library, see the user's guide for the *C28x IQmath Library - A Virtual Floating Point Engine*, Literature Number SPRC087, available at the Texas Instruments Web site. The user's guide is included in the zip file download that also contains the IQmath library (registration required).

Fixed-Point Numbers

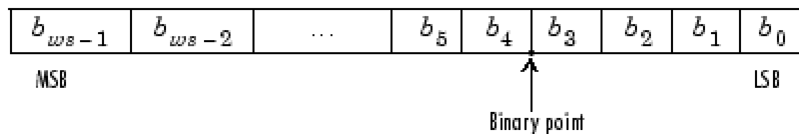
- "Notation" on page 55-55
- "Signed Fixed-Point Numbers" on page 55-56
- "Q Format Notation" on page 55-56

Notation

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1s and 0s). How hardware components or software functions interpret this sequence of 1s and 0s is defined by the data type.

Binary numbers are used to represent either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a fractional fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the *i*th binary digit.
- *ws* is the word size in bits.
- b_{ws-1} is the location of the most significant (highest) bit (MSB).
- b_0 is the location of the least significant (lowest) bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of 4.

Note For Embedded Coder, the results of fixed-point and integer operations in MATLAB/Simulink match the results on the hardware target down to the least significant bit (bit-trueness). The results of floating-point operations in MATLAB/Simulink do not match those on the hardware target, because the libraries used by the third-party compiler may be different from those used by MATLAB/Simulink.

Signed Fixed-Point Numbers

Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is used by TI digital signal processors.

Negation using signed two's complement representation consists of a bit inversion (translation to one's complement representation) followed by the binary addition of a 1. For example, the two's complement of 000101 is 111011, as follows:

000101 ->111010 (bit inversion) ->111011 (binary addition of a 1 to the LSB)

Q Format Notation

The position of the binary point in a fixed-point number determines how you interpret the scaling of the number. When it performs basic arithmetic such as addition or subtraction, hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a binary point. They perform signed or unsigned integer arithmetic — as if the binary point is to the right of b_0. Therefore, you determine the binary point.

In the IQmath Library, the position of the binary point in the signed, fixed-point data types is expressed in and designated by Q format notation. This fixed-point notation takes the form

$Qm.n$

where

- Q designates that the number is in Q format notation — the Texas Instruments representation for signed fixed-point numbers.

- *m* is the number of bits used to designate the two's complement integer portion of the number.
- *n* is the number of bits used to designate the two's complement fractional portion of the number, or the number of bits to the right of the binary point.

In Q format, the most significant bit is always designated as the sign bit. Representing a signed fixed-point data type in Q format always requires m+n+1 bits to account for the sign.

Note The range and resolution varies for different Q formats. For specific details, see Section 3.2 in the *Texas Instruments C28x Foundation Software, IQmath Library Module User's Guide.*

When converting from Q format to floating-point format, the accuracy of the conversion depends on the values and formats of the numbers. For example, for single-precision floating-point numbers that use 24 bits, the resolution of the corresponding 32-bit number cannot be achieved. The 24-bit number approximates its value by truncating the lower end. For example:

32-bit integer 11110000 11001100 10101010 00001111
Single-precision float +1.1110000 11001100 10101010 x 231
Corresponding value 11110000 11001100 10101010 00000000

Example — Q.15. For example, a signed 16-bit number with n = 15 bits to the right of the binary point is expressed as

```
Q0.15
```

in this notation. This is (1 sign bit) + (m = 0 integer bits) + (n = 15 fractional bits) = 16 bits total in the data type. In Q format notation, the m = 0 is often implied, as in

```
Q.15
```

In Simulink Fixed Point software, this data type is expressed as

```
sfrac16
```

or

```
sfix16_En15
```

In DSP System Toolbox software, this data type is expressed as

```
[16 15]
```

Example — Q1.30. Multiplying two Q0.15 numbers yields a product that is a signed 32-bit data type with n = 30 bits to the right of the binary point. One bit is the designated sign bit, thereby forcing m to be 1:

m+n+1 = 1+30+1 = 32 bits total

Therefore, this number is expressed as

```
Q1.30
```

In Simulink Fixed Point software, this data type is expressed as

```
sfix32_En30
```

In DSP System Toolbox software, this data type is expressed as

```
[32 30]
```

Example — Q-2.17. Consider a signed 16-bit number with a scaling of $2^{(-17)}$. This requires n = 17 bits to the right of the binary point, meaning that the most significant bit is a *sign-extended* bit.

Sign extension fills additional bits with the value of the MSB. For example, consider a 4-bit two's complement number 1011. When this number is extended to 7 bits with sign extension, the number becomes 1111101 and the value of the number remains the same.

One bit is the designated sign bit, forcing m to be -2:

m+n+1 = -2+17+1 = 16 bits total

Therefore, this number is expressed as

```
Q-2.17
```

In Simulink Fixed Point software, this data type is expressed as

```
sfix16_En17
```

In DSP System Toolbox software, this data type is expressed as

```
[16 17]
```

Example — Q17.-2. Consider a signed 16-bit number with a scaling of 2^(2) or 4. This means that the binary point is implied to be 2 bits to the right of the 16 bits, or that there are n = -2 bits to the right of the binary point. One bit must be the sign bit, thereby forcing m to be 17:

m+n+1 = 17+(-2)+1 = 16

Therefore, this number is expressed as

```
Q17.-2
```

In Simulink Fixed Point software, this data type is expressed as

```
sfix16_E2
```

In DSP System Toolbox software, this data type is expressed as

```
[16 -2]
```

Building Models

- "Overview" on page 55-59
- "Converting Data Types" on page 55-60
- "Using Sources and Sinks" on page 55-60
- "Choosing Blocks to Optimize Code" on page 55-60
- "Double and Single-Precision Parameter Values" on page 55-60

Overview

You can use IQmath Library blocks in models along with certain core Simulink, Simulink Fixed Point, and other blockset blocks. This section discusses issues you should consider when building a model with blocks from these different libraries.

Converting Data Types

As always, it is vital to make sure that any blocks you connect in a model have compatible input and output data types. In most cases, IQmath Library blocks handle only a limited number of specific data types. You can refer to any block reference page in the alphabetical block reference for a discussion of the data types that the block accepts and produces.

When you connect IQmath Library blocks and Simulink Fixed Point blocks, you often need to set the data type and scaling in the block parameters of the Simulink Fixed Point block to match the data type of the IQmath Library block. Many Simulink Fixed Point blocks allow you to set their data type and scaling through inheritance from the driving block, or through backpropagation from the next block. This can be a good way to set the data type of a Simulink Fixed Point block to match a connected IQmath Library block.

Some DSP System Toolbox blocks and core Simulink blocks also accept fixed-point data types. Make the appropriate settings in these blocks' parameters when you connect them to an IQmath Library block.

Using Sources and Sinks

The IQmath Library does not include source or sink blocks. Use source or sink blocks from the core Simulink library or Simulink Fixed Point in your models with IQmath Library blocks.

Choosing Blocks to Optimize Code

In some cases, blocks that perform similar functions appear in more than one blockset. For example, the IQmath Library and Simulink Fixed Point software have a Multiply block. When you are building a model to run on C2000 DSP, choosing the block from the IQmath Library always yields better optimized code. You can use a similar block from another library if it gives you functionality that the IQmath Library block does not support, but you will generate code that is less optimized.

Double and Single-Precision Parameter Values

When you enter double-precision floating-point values for parameters in the IQ Math blocks, the software converts them to single-precision values that

are compatible with the behavior on c28x processor. For example, with the **Ramp Generator** block, the software converts the value of the **Maximum step angle** parameter to a single-precision value.

Programming Flash Memory

In this section...

"Introduction" on page 55-62

"Installing TI Flash APIs" on page 55-62

"Configuring the DSP Board Bootloader" on page 55-63

"Configuring the Software for Automatic Flash Programming" on page 55-64

"Selectively Erase, Program, or Verify Specific Flash Sectors" on page 55-64

"Placing Additional Code or Data on Unused Flash Sectors" on page 55-65

Introduction

The Embedded Coder software includes a feature for programming Flash memory on the DSP target. You can configure this feature to automatically program Flash memory when you build and execute models for DSP boards. You can also use the Flash programming feature to selectively erase, program, or verify specific sectors of Flash memory.

Note Reprogramming Flash memory thousands of times may deplete its ability to hold data. Consult the manufacturer's documentation for specifications.

Requirements:

- A F2812, F2808, or F28335 eZdsp board
- A working model that includes a Target Preferences block for "**Stand alone code using Flash Memory**"
- The TI Flash API for your specific target

Installing TI Flash APIs

1 Visit the Texas Instruments Web site and download the TI Flash API installation software for your target:

- F281x: http://focus.ti.com/docs/toolsw/folders/print/sprc125.html
- F280x: http://focus.ti.com/docs/toolsw/folders/print/sprc193.html
- F2802x: http://focus.ti.com/docs/toolsw/folders/print/sprc848.html
- F2804x: http://focus.ti.com/docs/toolsw/folders/print/sprc325.html
- F2823x: http://focus.ti.com/docs/toolsw/folders/print/sprc665.html
- F2833x: http://focus.ti.com/docs/toolsw/folders/print/sprc539.html

2. Start the TI Flash API installation software (.exe) contained in the ZIP file.

3. During installation, *use the default folder location* for **Location to Save Files**.

   Otherwise, each time you create a model, you must configure **Specify API Location**, located under the **Peripherals** tab of the Target Preferences block.

4. Complete the installation process.

Configuring the DSP Board Bootloader

Configure the bootloader switch or jumper on the DSP board so that, upon startup, the DSP board executes the program from Flash memory. Consult the manufacturer's hardware documentation to identify the specific switch and settings.

Typically, you can enable the bootloader switch or jumper by moving it from the factory default position (Flash disabled) to the opposite position (enabled). For example:

- On the F2812 eZdsp, change jumper JP7 from the factory default setting.
- On the F2808 eZdsp, change switches 1 and 3 on bank SW1 from the factory default settings.
- On F28335 eZdsp, change switch 3 on bank SW1 from the factory default setting.

Configuring the Software for Automatic Flash Programming

Configure Embedded Coder software to program Flash memory on the target board when you build and execute a model.

1. On your keyboard, press Ctrl+E to open the Configuration Parameters dialog box, select *Code Generation* and *IDE Link*, and confirm **Build Action** is set to **Build_and_execute**.

2. Open the Target Preferences block in your model, select the **Peripherals** tab, and then select **Flash_loader**.

3. Set **Enable flash programmer** to **Erase, Program, Verify**.

4. Click **OK** to save and close the new configuration.

When you build the model, the software automatically erases, programs, and verifies Flash memory. When the DSP board restarts, it loads and executes the program from Flash memory.

Selectively Erase, Program, or Verify Specific Flash Sectors

You can manually erase, program, and verify specific sectors of Flash memory:

1. Open the Target Preferences block in your model, and select the **Peripherals** tab.

2. Select **Flash_loader** from the **Peripherals** list.

3. Set **Enable flash programmer** to erase, program, or verify flash.

4. (Optional) To protect specific Flash sectors:

   a. Disable **Detect Flash sectors to erase from COFF file**.

   b. Deselect the flash sectors you want to protect.

5. Click **Execute**. The software performs the action you specified upon the unprotected flash sectors.

Note Erase Flash sectors before programming them.

Placing Additional Code or Data on Unused Flash Sectors

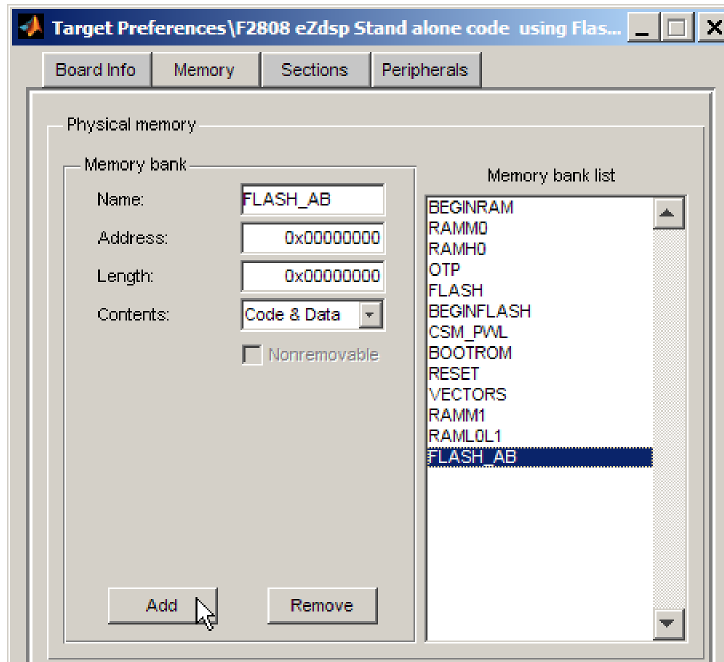
To place additional code or data on unused Flash sectors:

1. Determine the address and length of the individual Flash sectors. You may need to refer to the manufacturer's specifications.

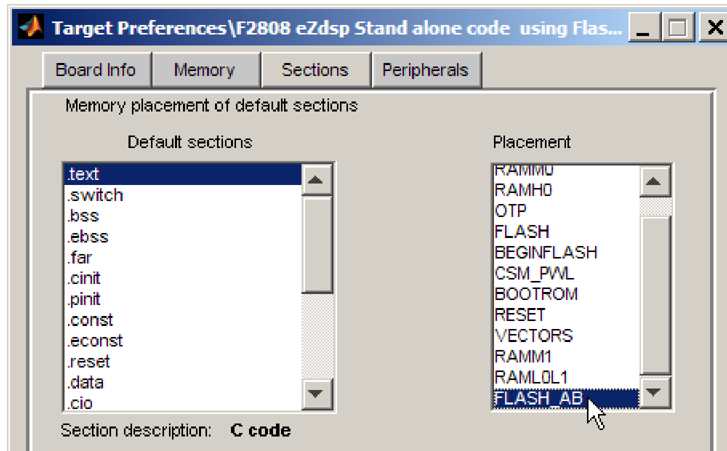
2. Determine the size of the primary C code program and the number of Flash sectors it occupies.

3. Determine the size of the additional code or data and the number of Flash sectors it will occupy.

4. Under the Target Preferences **Memory** tab, click **Add** to create two or more new memory banks; one for the primary C code program (e.g., FLASH_AB) and one or more for the additional code or data (e.g., FLASH_CD). The address and length of each memory bank must align with those of the flash sectors.



5 Under the **Sections** tab, under **Default sections**, select **.text**. Then, under **Placement**, select the new memory bank (e.g., FLASH_AB) you created for the primary C code program. The next time you program the Flash memory, the software places the **.text** C code file in the new memory bank.



6 Similarly, select items from the **Default sections** or **Custom sections list**, and place them in the new memory banks (e.g., FLASH_CD) for the previously unoccupied Flash sectors.

Configuring LIN Communications

In this section...

"Overview" on page 55-68

"Configuring Your Model" on page 55-68

Overview

The LIN communications architecture supports a single master node and up to 16 slave nodes on a LIN network.

LIN nodes use message frames to exchange data. The message has two parts:

- Frame header, generated by the Master node.
- Frame response, which contains data generated by either Slave node or a slave task on a Master node (but not both).

Configuring Your Model

First, study, and understand the LIN addressing system. See the "Message Filtering and Validation" topic in the TMS320F2803x Piccolo Local Interconnect Network (LIN) Module, Literature Number: SPRUGE2A.

Configure the LIN node in your model as a master or slave node:

1. Add a Target Preferences block to your model.
2. In the Target Preferences block, select the **Peripherals** tab, and then select **LIN**.
3. Set **LIN mode** to `Master` or `Slave`.

If the LIN node is a Master node:

- Add a LIN Transmit block to the model. This block enables the Master to generate message headers.
- To send data, set the **ID** input and **Tx ID Mask** input to make Tx ID Match happen on this node.

- To receive data, place LIN Receive block in the model. Set the **Rx ID Mask** input to make Rx ID Match happen on this node.

For example, to configure a model with a master node that receives data from a slave node:

- Add a LIN Transmit block and a LIN Receive block to the model.
- In the Target Preferences block, configure the **ID Slave Task Byte**.
- For the LIN Transmit block, set the **ID** input.
- For the LIN Receive block, set the **Rx ID Mask** input so that: **Rx ID Mask** = **ID** XOR **Slave Task ID Byte**.

If the LIN node is a Slave node:

- To send data, place LIN Transmit block in the model. Set the **ID** input to match the LIN frame header issued by the remote Master. Set **Tx ID Mask** to make a Tx ID Match happen on this node.
- To receive data, place LIN Receive block in the model. Set the **Rx ID Mask** input to make an Rx ID Match happen on this node.

For example, to configure a model with a slave node that transmits data to a master node:

- Add a LIN Transmit block to the model.
- In the Target Preferences block, configure the **ID byte** or **ID Slave Task Byte** (depending on the **ID filtering** option).
- In the LIN Transmit block, set the **ID** input and **Tx ID Mask** input so that: **Tx ID Mask** = **ID** XOR (**ID Byte** or **ID Slave Task Byte**).

Always set the **Data type** and **Data length** values in your LIN Receive blocks to match the type and length of the transmitted data. These values enable the receive block reconstruct the data from the message frames correctly.

Note The LIN Transmit block inherits the data type and length from its input.

Working with Texas Instruments C6000 Processors

- "Getting Started" on page 56-2
- "Targeting C6000 DSP Hardware" on page 56-7
- "Targeting with DSP/BIOS Options" on page 56-71
- "Using the C62x and C64x DSP Libraries" on page 56-99
- "Configuring Timing Parameters for CAN Blocks" on page 56-109
- "Hardware Issues" on page 56-113

Getting Started

In this section...
“Overview” on page 56-2
“Using This Guide” on page 56-2
“Configuration Information” on page 56-3
“Setting Up and Configuring” on page 56-4

Overview

Product Description

Use Embedded Coder to deploy generated code for real-time execution on embedded microprocessors, microcontrollers, and DSPs. Using Embedded Coder, you can integrate peripheral devices with the algorithms created using Embedded MATLAB™, Simulink®, and Stateflow®. You can deploy the resulting executable onto embedded hardware for on-target rapid prototyping, real-time performance analysis, and field production.

Using This Guide

Expected Background

This document introduces you to using Embedded Coder software to develop digital signal processing applications for the Texas Instruments C6000 family of DSP development hardware, such as the TI TMS320C6713 DSP Starter Kit. To get the most out of this manual, you should be familiar with MATLAB software and its associated programs, such as DSP System Toolbox software and Simulink software. We do not discuss details of digital signal processor operations and applications, except to introduce concepts related to using specific targets. For more information about digital signal processing, you may find one or more of the following books helpful:

- McClellan, J. H., R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*, Prentice Hall, 1998.

- Lapsley, P., J. Bier, A. Sholam, and E. A. Lee, *DSP Processor Fundamentals Architectures and Features,* IEEE Press, 1997.
- Oppenheim, A.V., R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.
- Mitra, S. K., *Digital Signal Processing — A Computer-Based Approach*, The McGraw-Hill Companies, Inc, 1998.
- Steiglitz, K, *A Digital Signal Processing Primer*, Addison-Wesley Publishing Company, 1996.

Refer to the documentation for your TI boards for information about setting them up and using them.

If You Are a New User. **New users** should read "Getting Started" on page 56-2, which introduces the Embedded Coder environment—the required software and hardware, installation requirements, and the board configuration settings that you need. You will find descriptions of the blocks associated with the targeting software, and an introduction to the range of digital signal processing applications of which the Embedded Coder is capable.

If You Are an Experienced User. **All users** should read "Targeting C6000 DSP Hardware" on page 56-7 for information and examples about using the new blocks and build software to target your C6713 DSK. Two example models introduce the targeting software and build files, and give you an idea of the range of applications supported by the Embedded Coder.

Configuration Information

To determine whether Embedded Coder software is installed on your system, type this command at the MATLAB prompt.

```
c6000lib
```

Entering that command displays the C6000 block library:

To verify that the CCS IDE is installed on your machine, enter:

```
>> ccsboardinfo
Board Board                                  Proc Processor          Processor
 Num  Name                                   Num  Name               Type
 ---  ---------------------------------      ---  -----------------------------
 0    C6713 Device Cycle Accurate Si ...     0    TMS320C6713        TMS320C6000
```

With the CCS IDE installed and configured, the command line returns information about the boards that CCS IDE recognizes on your machine, in a form similar to the preceding example.

If the command line does not return information about any boards, revisit your CCS IDE installation and setup in your CCS IDE documentation.

As a final test, launch CCS IDE to ensure that it starts up successfully. For the Embedded Coder to operate with this application, the CCS IDE must be able to run on its own.

Setting Up and Configuring

- "System Requirements" on page 56-4
- "Supported Hardware" on page 56-5
- "Installing and Configuring Software" on page 56-5

System Requirements

For detailed information about the software and hardware required to use Embedded Coder software, refer to the Embedded Coder system requirements areas on the MathWorks Web site:

- Requirements for Embedded Coder: www.mathworks.com/products/target-package/requirements.html
- Requirements for use with TI's C6000: www.mathworks.com/products/target-package/ti-adaptor/

Supported Hardware

For a list of supported hardware, visit http://www.mathworks.com/products/target-package/supportedio.html.

Installing and Configuring Software

Consult the "System Requirements" on page 56-4 for Embedded Coder . Only use *supported versions* of the software listed under "Third-Party Embedded Coder Requirements". Uninstall unsupported versions *before* installing supported versions. Doing so prevents errors that occur when the Windows Environment Variables points to the unsupported versions.

The System Requirements describe where you can obtain the additional third-party software, and when available, provide links for downloading that software.

Install the software (only the supported versions!) in the following order:

1 If needed, install the required and optional MathWorks software. (The software license you purchase determines which products are available.)

2 If needed, install TI Code Composer Studio (CCS).

3 Install TI Service Release for CCS.

4 Install the TI Code Generation Tools for you processor.

5 If you are using a Spectrum Digital board, download and install the matching Spectrum Digital Driver.

6 Install additional board-specific packages in the order in which they appear on the System Requirements web page.

Configure CCS as follows:

1 In CCS, open **Help > About > Component Manager > Build tools**

2 Open each target processor you will be using and enable the supported version of **Code Generation Tools**.

3. Open **Help > About > Component Manager > Build Tools > Target Content (DSP/BIOS)** .

4. Open each target processor you will be using and enable the supported version of **Texas Instruments DSP/BIOS**.

5. In Component Manager, select Save the changes. Then exit and restart CCS.

6. If you have a Spectrum Digital DM6437EVM board and or an Avnet S3ADSP DM6437 board, refer to “Installing and Configuring the Avnet Board Support Library” on page 56-116.

7. Verify the installation by repeating the instructions in “Configuration Information” on page 56-3.

Targeting C6000 DSP Hardware

In this section...

"Introduction to Targeting" on page 56-7

"C6000 and Code Composer Studio IDE" on page 56-8

"Targeting Tutorial — Single Rate Application" on page 56-11

"Schedulers and Timing" on page 56-21

"Model Reference and Embedded Coder Software" on page 56-34

"Targeting Supported Boards" on page 56-38

"Simulink Models and Targeting" on page 56-43

"Targeting Tutorial II — A More Complex Application" on page 56-43

"Targeting Your C6713 DSK and Other Hardware" on page 56-50

"Creating Code Composer Studio Projects Without Building" on page 56-54

"Targeting Custom Hardware" on page 56-55

"Using Embedded Coder Software" on page 56-69

Introduction to Targeting

- "Overview" on page 56-7
- "About the Tutorials" on page 56-8

Overview

The Embedded Coder software lets you use Simulink Coder software to generate a C language real-time implementation of your Simulink model. You can compile, link, download, and execute the generated code on the C6713 DSP Starter Kit (DSK). The Embedded Coder is ideal for rapid prototyping and developing embedded systems applications for C6713 digital signal processors. The Embedded Coder focuses on developing real-time digital signal processing (DSP) applications for C6000 hardware. Additional hardware that we support is listed in "Hardware Issues" on page 56-113.

Although the tutorials in this chapter focus on the C6713 DSK, the techniques and processes apply to any supported hardware, with minor adjustments for the processor involved.

This chapter describes how to use the Embedded Coder to create and execute applications on Texas Instruments C6000 development boards. To use the targeting software, you should be familiar with using Simulink software to create models and with the basic concepts of Simulink Coder software automatic code generation. To read more about Simulink Coder software, refer to your Simulink Coder documentation.

About the Tutorials

In most cases, this chapter deals with the C6713 DSK targets. Fortunately, all members of the C6000 family of processors that we support work in a manner similar to the C6713 DSK. While you review the contents of this chapter, and follow the tutorials, recall that the concepts and techniques or development processes apply, with a few adjustments, to all supported C6000 processors and boards.

Later sections discuss the Embedded Coder software and targeting custom hardware.

Tip To make your figure easier to read, use easily distinguishable colors and line styles.

C6000 and Code Composer Studio IDE

- "Using Code Composer Studio with Embedded Coder Software" on page 56-8
- "About Simulators" on page 56-9
- "Typical Hardware Setup for a Development Board" on page 56-10

Using Code Composer Studio with Embedded Coder Software

Texas Instruments (TI) markets a complete set of software tools to use when you develop applications for your C6000 hardware boards. This

section provides a brief example of how Embedded Coder software uses Code Composer Studio (CCS) Integrated Development Environment (IDE) with the Simulink Coder software and the c6000lib blockset.

Executing code generated from Simulink Coder software on a particular target in real time requires that Simulink Coder software generate target code that is tailored to the specific hardware target. Target-specific code includes I/O device drivers and an interrupt service routine (ISR). Since these device drivers and ISRs are specific to particular hardware targets, you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, TI C6000 uses the MATLAB links in Embedded Coder software to invoke the code building process within the CCS IDE. After you download your executable to your target and run it, the code runs wholly on the target; you can access the running process only from the CCS IDE debugging tools. Otherwise the running process is not accessible.

Used in combination with your Embedded Coder and Simulink Coder software, TI products provide an integrated development environment that, once installed, needs no additional coding.

About Simulators

The CCS IDE offers simulators for the C6000 processors in the CCS IDE Setup utility. Much of your model and algorithm development efforts work with the simulators, such as code generation. And, since the Embedded Coder provides a software-based scheduler, your models and generated code run on the simulators just as they do on your hardware. For more information about the simulators in CCS IDE, refer to your CCS online help system.

When you set up a simulator, match the processor on your target exactly to simulate your target hardware. For example, to target a C6713DSK board, your simulator must contain a C6713 processor, not just a C6xxx simulator. Simulators must match the target processor because the codecs on the board are not the same and the simulator needs to identify the correct codec. Correctly matching your simulator to your hardware ensures that the memory maps and registers match those of your intended target signal processor.

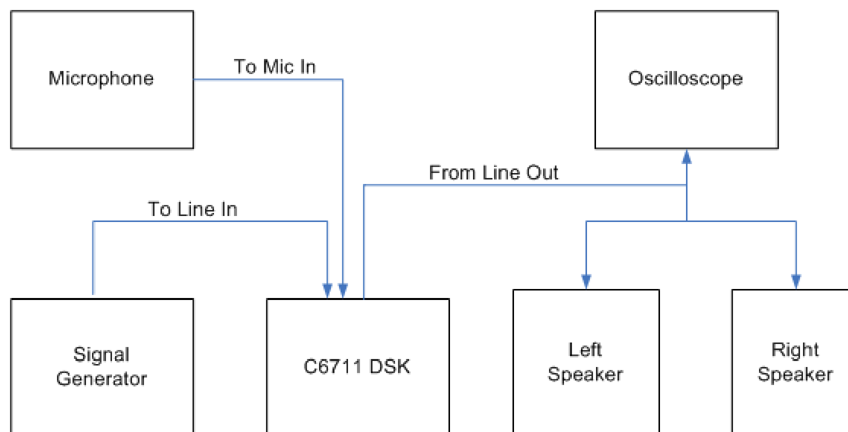
To use a simulator. Open the Target Preferences block. On the **Board** pane, under **IDE Support**, use the **Get from IDE** button to get a list of simulators

installed with your IDE. The use **Board Name** to select one of the installed simulators.

In general, use the device cycle accurate simulators provided by CCS Setup to simulate your processor.

Typical Hardware Setup for a Development Board

The following block diagram represents typical inputs and output for a C6713 DSK development board.



After installing a supported development board, start MATLAB software. At the command prompt, type `c6000lib`. This opens a Simulink blockset named c6000lib that includes libraries that contain blocks predefined for C6000 input and output devices.

The board-based block library for the C6713 DSK contains these blocks:

- ADC block
- DAC block
- DIP Switch block (optional, refer to the reference page for the DIP Switch block for your target)
- LED block
- Reset block

Blocks from these libraries are associated with your boards and hardware. As needed, add the devices to your model. If you choose not to include either an ADC or DAC block in your model (they are available in the target specific libraries), the Embedded Coder provides a timer that produces the interrupts required for timing and running your model, either on your hardware target or on a simulator.

Targeting Tutorial — Single Rate Application

- “Overview” on page 56-11
- “Building the Audio Reverberation Model” on page 56-12
- “Adding C6713 DSK Blocks to Your Model” on page 56-13
- “Configuring Embedded Coder Blocks” on page 56-14
- “Specifying Configuration Parameters for Your Model” on page 56-18

Overview

In this tutorial you create and build a model that simulates audio reverberation applied to an input signal. Reverberation is similar to the echo effect you can hear when you shout across an open valley or canyon, or in a large empty room.

You can choose to create the Simulink model for this tutorial from blocks in DSP System Toolbox software and Simulink block libraries, or you can find the model in Embedded Coder demos. For this example, you see the model as it appears in the demonstration program. The demonstration model name is `c6713dskafxr.mdl` as shown in the next figure. Open this model by entering `c6713dskafxr` at the MATLAB prompt.

To run this model you need a microphone connected to the **Mic In** connector on your C6713 DSK, and speakers and an oscilloscope connected to the **Line Out** connector on your C6713 DSK. To test the model, speak into the microphone and listen to the output from the speakers. You can observe the output on the oscilloscope as well.

To download and run your model on your C6713 DSK, complete the following tasks:

1. Use Simulink blocks, DSP System Toolbox software blocks, and blocks from other blocksets to create your model application.

2. Add Embedded Coder blocks that let your signal sources and output devices communicate with your C6713 DSK—the C6713 DSK ADC and C6713 DSK DAC blocks that you find in Embedded Coder c6000lib blockset.

3. Add the Target Preferences block to your model. Verify and set the block parameters for your hardware. In most cases, the default settings work fine.

   If you are using a C6713 simulator, use the **Get from IDE** button on the **Board** pane of the Target Preferences block. Then set **Board Name** under **IDE Support** to `C6713 Device Cycle Accurate Simulator`.

4. Set the configuration parameters for your model, including

   - Solver parameters such as simulation start and solver options
   - Simulink Coder software options such as target configuration and target compiler selection

5. Build your model to the selected target.

6. Test your model running on the target by changing the input to the target and observing the output from the target.

Your target for this tutorial is your C6713 DSK installed on your PC. Be sure to configure and test your board as directed in “Configuring Your C6713DSK” on page 56-41 in this guide before continuing this tutorial.

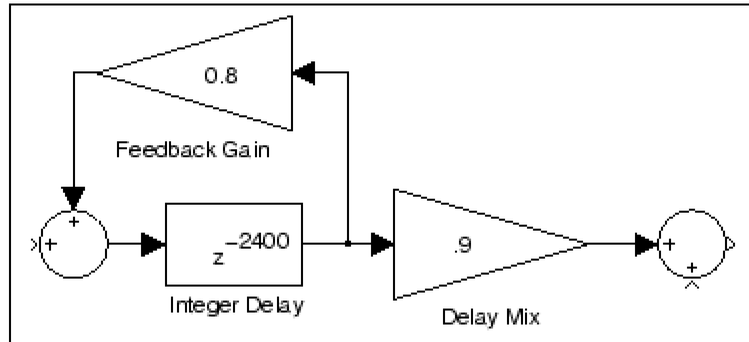
Building the Audio Reverberation Model

To build the model for audio reverberation, follow these steps:

1. Start Simulink.

2. Create a new model by selecting **File > New > Model** from the **Simulink** menu bar.

3. Use Simulink blocks and DSP System Toolbox software blocks to create the following model.



Look for the Integer Delay block in the Signal Operations library of the DSP System Toolbox software. You do not need to add the input and output signal lines at this time. When you add the C6713 DSK blocks in the next section, you add the input and output to the sum blocks.

4 Save your model with a suitable name before continuing.

Adding C6713 DSK Blocks to Your Model

So that you can send signals to your C6713 DSK and get signals back from the board, Embedded Coder software includes a block library containing five blocks designed to work with the codec on your C6713 DSK:

- Input block (C6713 DSK ADC)
- Output block (C6713 DSK DAC)
- Light emitting diode block (C6713 DSK LED)
- Software reset block (Reset C6713 DSK)
- DIP switch block (C6713 DSK DIP Switch)

Entering `c6713dsklib` at the MATLAB prompt opens the block library for the C6713 DSK. This block library is included in Embedded Coder `c6000lib` blockset in the Simulink Library browser.

The C6713 DSK ADC and C6713 DSK DAC blocks generate code that configures the codec on your C6713 DSK to accept input signals from the input connectors on the board, and send the model output to the output connector

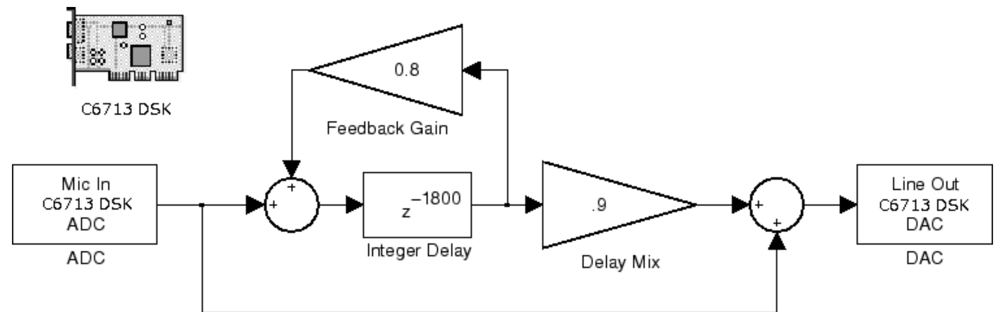
on the board. Essentially, the C6713 DSK ADC and C6713 DSK DAC blocks add driver software that controls the behavior of the codec for your model.

To add C6713 DSK target blocks to your model, follow these steps:

1. Double-click Embedded Coder software in the Simulink Library browser to open the c6000lib blockset.

2. Click the block library for the C6713 DSK to see the blocks available for your C6713 DSK.

3. Drag and drop C6713 DSK ADC and C6713 DSK DAC blocks to your model as shown in the figure.



4. Connect new signal lines as shown in the figure.

5. Finally, add the Target Preferences block to the model. Notice that it is not connected to any other block in the model.

Configuring Embedded Coder Blocks

To configure Embedded Coder blocks in your model, follow these steps:

1. Click the C6713 DSK ADC block to select it.

2. Select **Block Parameters** from the Simulink **Edit** menu.

3. Set the following parameters for the block:

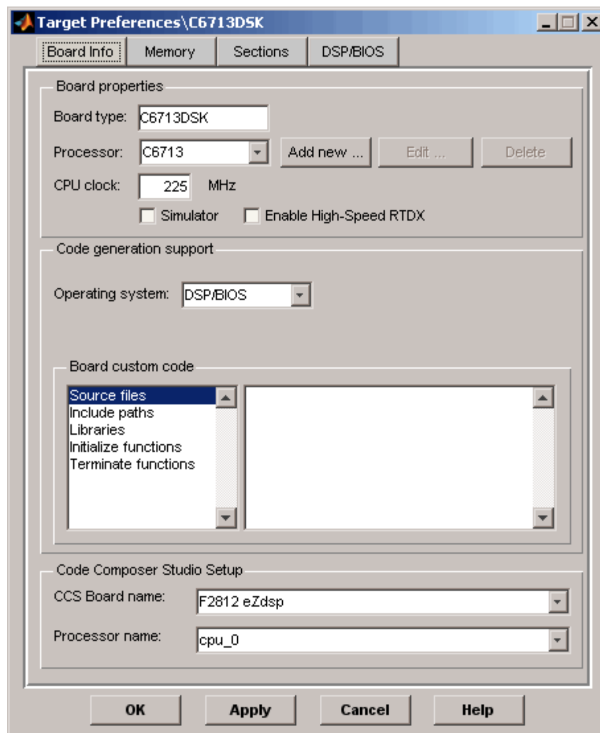
   - Clear the **Stereo** check box.

- Select the **+20 dB mic gain boost** check box.

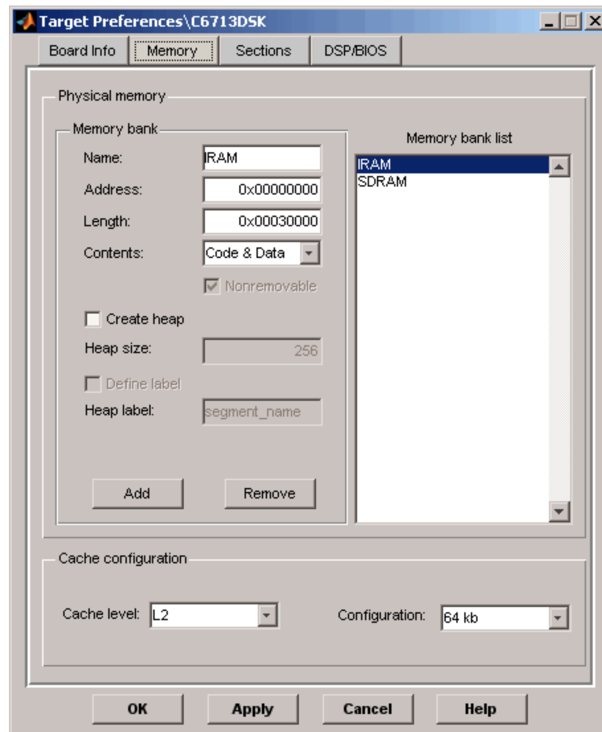
  From the list, set **Sample rate** to `8000`.
- Set **Codec data format** to `16-bit linear`.
- For **Output data type**, select `Double` from the list.
- Set **Scaling** to `Normalize`.
- Set **Source gain** to `0.0`.
- Enter `64` for **Samples per frame**.

Include a signal path directly from the input to the output so you can display both the input signal and the modified output signal on the oscilloscope for comparison.

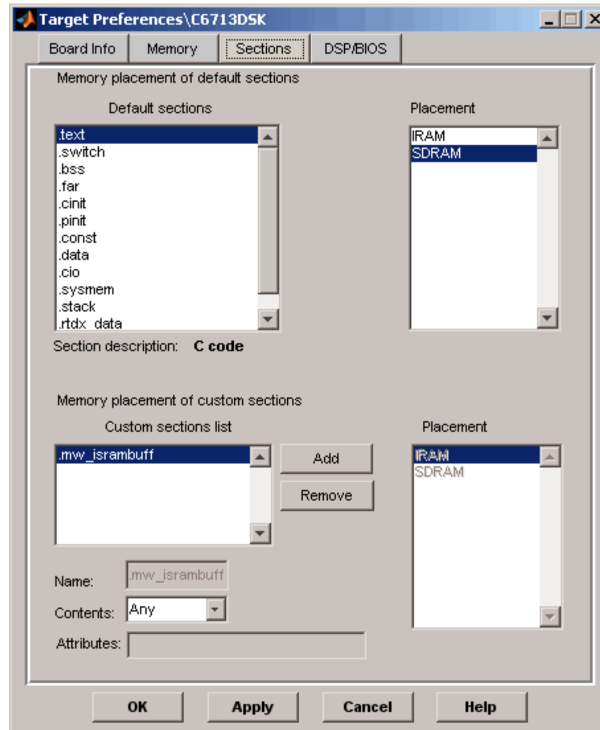
4. For **C6713 DSK ADC source**, select `Mic In`.
5. Click **OK** to close the C6713 DSK ADC dialog box.
6. Now set the options for the C6713 DSK DAC block.
   - Set **Codec data format** to `16-bit linear`.
   - Set **Scaling** to `Normalize`.
   - For **DAC attenuation**, enter `0.0`.
   - Set **Overflow mode** to `Saturate`.
7. Click **OK** to close the dialog box.
8. Click the Target Preferences block.
9. Select **Block Parameters** from the Simulink **Edit** menu.
10. Verify the parameter settings for the C6713 DSK target. The figures below show the proper values.



Board Settings



Memory Settings



Section Settings

You have completed the model. Now configure the Simulink Coder software options to build and download your new model to your C6713 DSK.

Specifying Configuration Parameters for Your Model

The following sections describe how to build and run real-time digital signal processing models on your C6713 DSK. Running a model on the target starts with configuring and building your model from the Configuration Parameters dialog box in Simulink software.

Setting Simulink Configuration Parameters. After you have designed and implemented your digital signal processing model in Simulink software, complete the following steps to set the configuration parameters for the model:

1. Open the Configuration Parameters dialog box and set the appropriate options on the `Solver` category for your model and for Embedded Coder software.

   - Set **Start time** to `0.0` and **Stop time** to `inf` (model runs without stopping). Generated code does not honor this setting if you set a stop time. Set this to `inf` for completeness.
   - Under **Solver options**, select the `Fixed-step` and `Discrete` settings from the lists
   - Set the **Fixed step size** to `Auto` and the **Tasking Mode** to `Single Tasking`

Note Generated code does not honor Simulink stop time from the simulation. Stop time is interpreted as `inf`. To implement a stop in generated code, you must put a Stop Simulation block in your model.

Ignore the `Data Import/Export`, `Diagnostics`, and `Optimization` categories in the Configuration Parameters dialog box. The default settings are correct for your new model.

Setting Simulink Coder Target Build Options. You can configure Simulink Coder software to generate and build code that is appropriate for your hardware target. Follow these steps to set the Simulink Coder options to target your C6713 DSK:

1. Open the Configuration Parameters dialog box by entering **Ctrl+E** or by selecting the **Simulation** menu item and then **Configuration Parameters**.

2. From the **Select** tree, choose **Code Generation**.

3. Verify that the system target file is set to **idelink_grt.tlc**. If needed, click **Browse** and select **idelink_grt.tlc**.

4. From the **Select** tree, choose **IDE Link**.

5. Among the **Runtime Options**, set **Build action** to **Build_and_execute**, and set **Interrupt overrun notification method** to **Print_message**.

6. Among the **Project Options**, keep the default settings.

7. Among the **Code Generation** options, clear **Profile real-time execution**.

8. Among the **Link Automation** options, verify that **Export IDE link handle to base workspace** is selected and that **IDE link handle name** has a name (e.g., **CCS_Obj**).

9. From the **Select** tree, choose **Hardware Implementation**.

10. Verify that **Byte ordering** is **Little endian**.

When you have completed these steps, you have configured the Simulink Coder options for the C6713 DSK target. Some Simulink Coder categories on the **Select** tree, such as **Comments**, **Symbols**, and **Optimization**, do not require configuration. The default values for the options in these categories are already correct for your new model. For other models, you may want to set the options in these categories to provide information during the build and to run TLC debugging when you generate code.

Building and Executing Your Model on Your C6713 DSK. After you set the configuration parameters and configure Simulink Coder software to create the files you need, you direct Simulink Coder software to build, download, and run your model executable on your target:

1. Change the category to `Code Generation` on the Configuration Parameters dialog box.

2. Clear **Generate code only** and click **Build** to generate and build an executable file targeted to your C6713 DSK.

   When you click **Build** with `Build_and_execute` selected for **Build action**, the automatic build process creates an executable file that can be run by the C6713 DSP on your C6713 DSK, and then downloads the executable file to the target and runs the file.

3. To stop model execution, click the **Reset C6713 DSK** block or use the **Halt** option in CCS IDE. You could type halt from the MATLAB command prompt as well.

Testing Your Audio Reverb Model. With your model running on your C6713 DSK, speak into the microphone you connected to the board. The model should generate a reverberation effect out of the speakers, delaying and echoing the words you speak into the mike. If you built the model yourself, rather than using the supplied model `c6713dskafxr`, try running the demonstration model to compare the results.

Schedulers and Timing

- "Timer-Based Versus Asynchronous Interrupt Processing" on page 56-21
- "Synchronous Scheduling" on page 56-22
- "Asynchronous Scheduling" on page 56-23
- "Asynchronous Scheduler Examples" on page 56-24
- "Uses for Asynchronous Scheduling" on page 56-28
- "Scheduling Considerations" on page 56-33

Timer-Based Versus Asynchronous Interrupt Processing

Code generated for periodic tasks, both single- and multitasking, runs out of the context of a timer interrupt. The generated code that represents model blocks for periodic tasks runs periodically, clocked by the periodic interrupt whose period is equal to the base sample time of the model. This description of scheduling and timing applies both to generated code operation that incorporates DSP/BIOS real-time operating system (RTOS) and basic code generation mode where DSP/BIOS RTOS is not included.

Note In timer-based models, the timer counts through one full base-sample-time before it creates an interrupt. When the model is finally executed, it is for time 0.

This execution scheduling scheme is not flexible enough for some systems, such as control and communication systems that must respond to asynchronous events in real time. Such systems may need to handle a variety of hardware interrupts in an asynchronous, or aperiodic, fashion.

When you plan your project or algorithm, select your scheduling technique based on your application needs.

- If your application processes hardware interrupts asynchronously, add the appropriate asynchronous scheduling blocks from the Embedded Coder library to your model, listed here.

Blocks in the DSP/BIOS (dspbioslib) library

  - Hardware Interrupt — Create interrupt service routine on C6000 hardware target.
  - Task — Create task that runs as separate DSP/BIOS thread.
  - Triggered Task — Create asynchronously triggered task.

Blocks in the Scheduling (c6000dspcorelib) library

  - Block Processing — Repeat user-specified operation on submatrices of input matrix, using internal memory of DSP for increased efficiency.
  - CPU timer — Generate interrupt service routine.
  - EDMA — Configure EDMA Controller on C6000 processor.

Blocks in the Embedded Coder library for Texas Instruments Code Composer Studio (idelinklib_ticcs)

  - C6000 Hardware Interrupt — Generate interrupt service routine. Same as the DSP/BIOS interrupt block.

Blocks in the idelinklib_common library

  - Idle Task — Create free-running background task

- If your application does not service asynchronous interrupts, your model should include only the algorithm and device driver blocks that specify the periodic sample times. Generating code from a model like this automatically enables and manages a timer interrupt. The periodic timer interrupt clocks the entire model.

Synchronous Scheduling

For code that runs synchronously in the context of the timer interrupt, each iteration of the model runs after an interrupt has been posted and serviced by an interrupt service routine (ISR). The code generated for Embedded Coder software uses Timer 1 in DSP/BIOS mode and bare-board mode. Timer 1 is configured so that the base rate sample time for the coded process corresponds

to the interrupt rate. The Embedded Coder calculates and configures the timer period to ensure the desired sample rate.

The minimum achievable base rate sample time depends on the algorithm complexity and the CPU clock speed. The maximum value depends on the maximum timer period value and the CPU clock speed.

If all the blocks in the model inherit their sample time value, and no sample time is defined explicitly, Simulink assigns a default sample time of 0.2 second.

Note In timer-based models, the timer counts through one full base-sample-time before it creates an interrupt. When the model is finally executed, it is for time 0.

Asynchronous Scheduling

Embedded Coder software facilitates modeling and automatically generating code for asynchronous systems by using the following scheduling blocks:

- Hardware Interrupt and Idle Task blocks for bare-board code generation mode
- DSP/BIOS Hardware Interrupt, DSP/BIOS Task, and DSP/BIOS Triggered Task blocks for DSP/BIOS code generation mode

C6000 Hardware Interrupt blocks enable selected hardware interrupts for the TI TMS320C6000 DSP, generate corresponding ISRs, and connect them to the corresponding interrupt service vector table entries.

When you connect the output of the C6000 Hardware Interrupt block to the control input of a function-call subsystem, the generated subsystem code is called from the ISRs each time the interrupt is raised.

The C6000 Idle Task block specifies one or more functions to execute as background tasks in the code generated for the model. The functions are created from the function-call subsystems to which the Idle Task block is connected.

The DSP/BIOS Hardware Interrupt block (in DSP/BIOS code generation mode) has the same functionality as the bare-board C6000 Hardware Interrupt block. The configuration and low-level handling of the hardware interrupts is implemented through DSP/BIOS using DSP/BIOS Hardware Interrupt module and DSP/BIOS dispatcher.

DSP/BIOS Task blocks (DSP/BIOS code generation mode) spawn free-running tasks as separate DSP/BIOS threads. The spawned task runs the function-call subsystem connected to its output. Blocks in the subsystem may use various conditions and techniques to control sharing sources with other tasks.

DSP/BIOS Triggered Task blocks (in DSP/BIOS code generation mode) spawn semaphore-controlled tasks as separate DSP/BIOS threads. The semaphore that enables execution of a single instance of the task is posted by an ISR that is created by a DSP/BIOS Hardware Interrupt block. This block is connected to a DSP/BIOS Triggered Task block.

Asynchronous Scheduler Examples

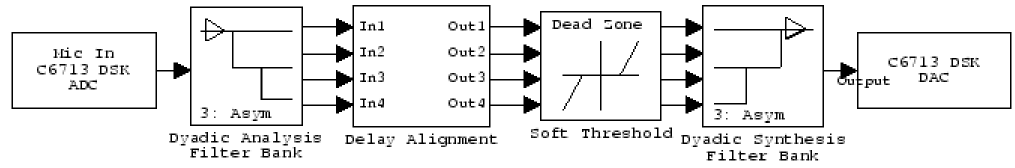
Now you can use an asynchronous (real-time) scheduler for your target application. Earlier versions of Embedded Coder software used a synchronous CPU timer interrupt-driven scheduler. With the asynchronous scheduler you can define interrupts and tasks to occur when you want them to using blocks in the following libraries:

- Core Support library (`idelinklib_common`)
- DSP/BIOS library (`dspbioslib`)

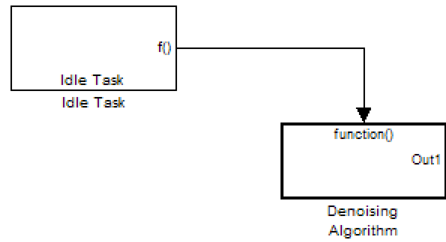
Also, you can schedule multiple tasks for asynchronous execution using those blocks libraries.

The following figures show a model updated to use the asynchronous scheduler rather than the synchronous scheduler.

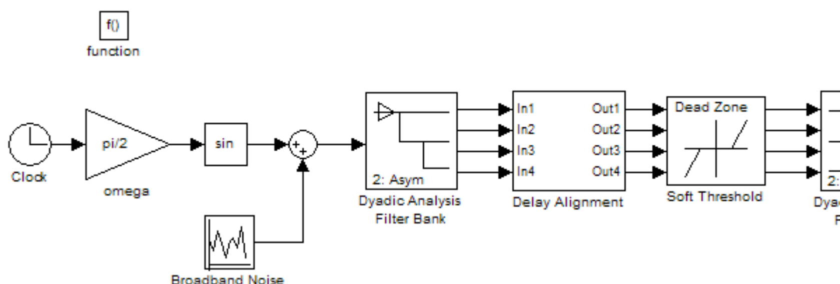
Before.



After.



Model Inside the Function Call Subsystem Block.



Compatibility Considerations

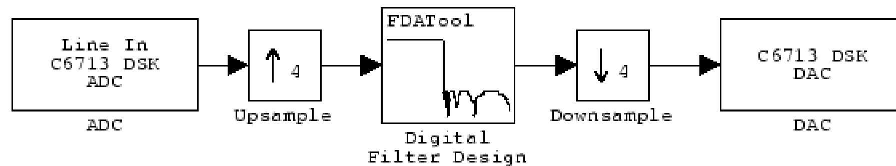
The V3.0 changes in the real-time scheduler can break some existing multirate models that contain codec blocks such as the ADC and DAC. The models affected contain at least one sample rate that is faster than the codec block rate. You do not run into this problem if all rates in the model are lower than the codec rate.

The new scheduler provides improved control for your processing and improved performance. You should recast all of your models to use the new asynchronous scheduler. To update your models, embed the entire processing algorithm or system in a function-call subsystem driven by a DSP/BIOS Task or Idle Task block from the DSP/BIOS library.

An example of such a model contains a combination of an ADC block and a DAC block, with a processing algorithm between them that executes at the higher rate. If you run code generated for such a model in `multitasking` or `auto` solver mode, you might hear occasional audio glitches or your program

may overrun. The exact symptom of the problem depends on the run-time overrun action setting in the `IDE Link` options.

The following model demonstrates one possible model configuration that can demonstrate the audio problems.



This multirate model uses two interrupts to control real-time execution of the generated code:

- A DMA interrupt to drive the execution of the code for ADC and DAC blocks
- A timer interrupt to drive the execution of the code for the FIR filter at an increased sample rate

In earlier product versions, the generated scheduler constantly synchronized the DMA and timer interrupts to ensure they remained in sync with one another, despite the possible clock drift with interrupts that are recorded by independent clock sources.

With the new real-time scheduler, the product does not synchronize the ADC and timer interrupts.

One interrupt may get out of sync with the other, with the time difference between them (drift) fluctuating with changes in the independent interrupt clocks. When the drift reaches a critical threshold, processing may skip an instance of a lower-priority task.

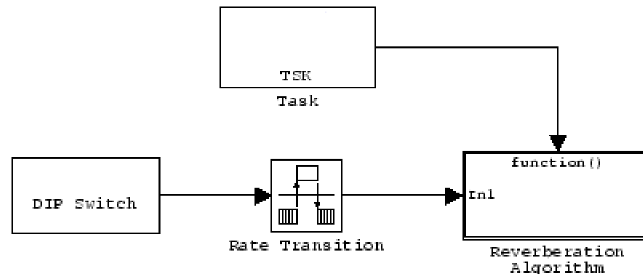
At that point, the interrupts are back in sync and the process continues. Losing synchronization between the interrupts can corrupt the audio signal or lead to an interrupt overrun.

To avoid the audio problems in an existing model that you cannot update to the new scheduler, set the run-time overrun action for the model to either `None` or `Notify_and_continue` to prevent the program from overrunning.

Uses for Asynchronous Scheduling

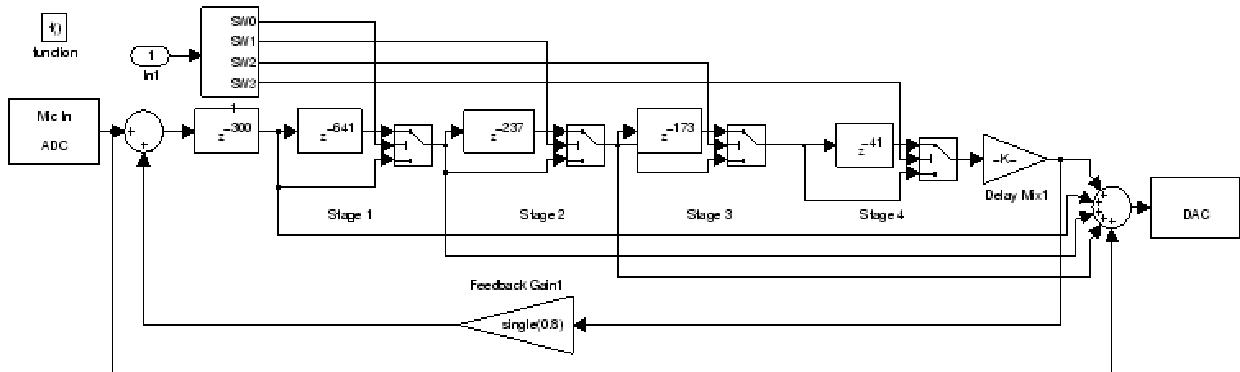
The following sections present common cases for the scheduling blocks described in the previous sections.

Free-Running DSP/BIOS Task. The following model illustrates a case where a reverberation algorithm runs in the context of a free-running DSP/BIOS task.



Normally, the algorithms in this type of task run in free-running mode, that is, they run repetitively and indefinitely. However, in this function-call subsystem (shown in detail in the following figure), ADC and DAC blocks suspend the execution of the task until the ADC and DAC data is available.

Each instance of the reverberation algorithm is triggered only after the data buffer is available (for both ADC and DAC). An asynchronous ADC/DAC device driver layer separate from the task function manages the triggers condition. This device driver layer uses a direct memory access (DMA) interrupt to signal to the DSP/BIOS task when ADC and DAC data become available for the task function.

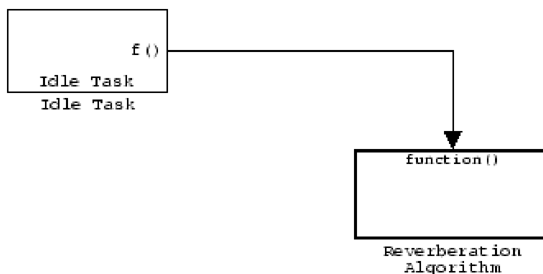


This model also illustrates how synchronous and asynchronous tasks can work together. The code generated for C6416 DSK DIP Switch block runs as a periodic task at the rate of 0.01 s. This is the only periodic task in the model. It runs out of the context of a DSP/BIOS task scheduled via a timer interrupt configured to go off every 0.01 second.

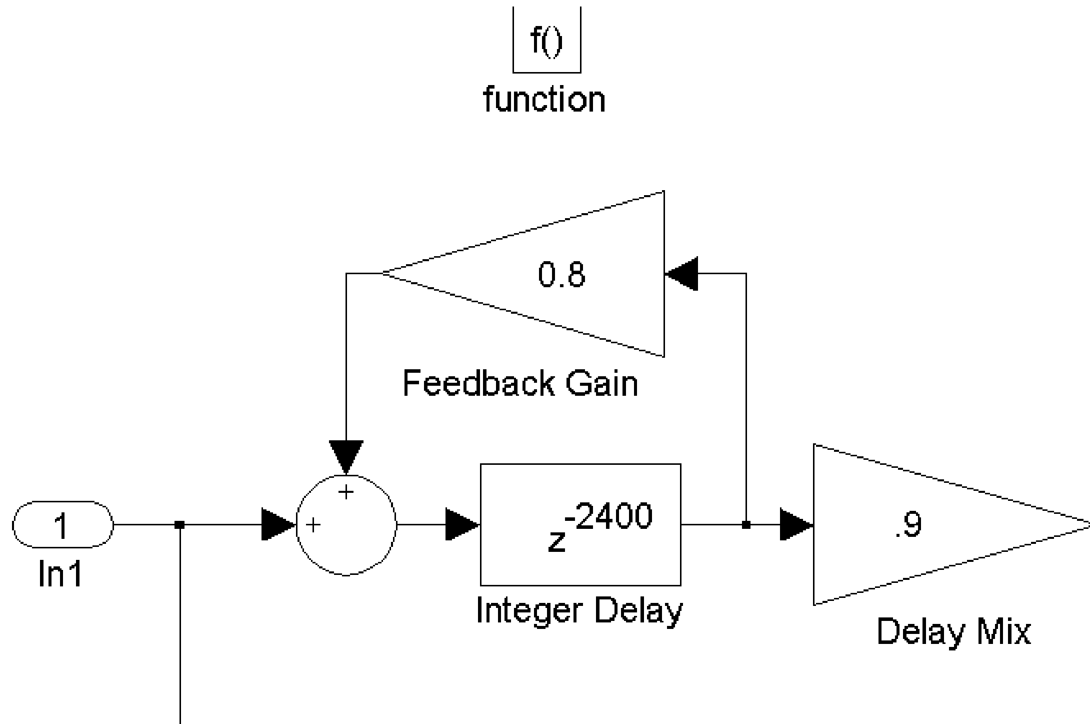
In general, Simulink blocks that specify nonzero sample rates, such as the DIP Switch block, are scheduled by the C6000 synchronous scheduler and executed either from the context of a DSP/BIOS task (if you incorporate DSP/BIOS in your project) or a hardware interrupt (when you do not incorporate DSP/BIOS).

To ensure data integrity, Simulink Rate Transition blocks connect the C6416 DSK DIP Switch block with the reverberation algorithm. This transition is required because the blocks belong to different rate groups. If the synchronous and asynchronous parts of the model do not interact, the Rate Transition blocks are not needed.

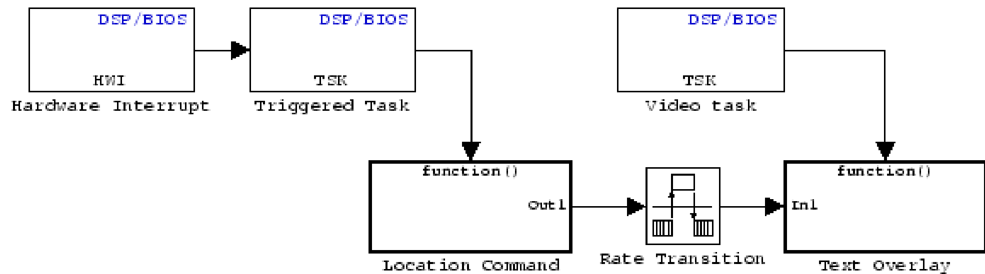
Idle Task. The following model illustrates a case where the reverberation algorithm runs in the context of a background task in bare-board code generation mode.



The function generated for this task normally runs in free-running mode—repetitively and indefinitely. However, the ADC and DAC blocks in this subsystem run in blocking mode. As a result, subsystem execution of the reverberation function is the same as the subsystem described for the Free-Running DSP/BIOS Task. It is data driven via a background DMA interrupt-controlled ISR, shown in the following figure.



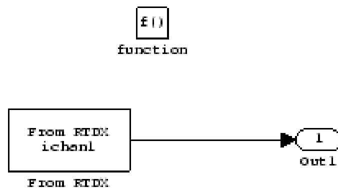
Hardware Interrupt Triggered DSP/BIOS Task. The next model illustrates a case where a function (Location Command) runs in the context of a hardware interrupt-triggered DSP/BIOS task.

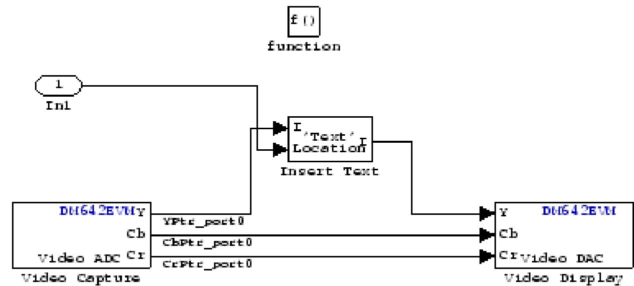


The DSP/BIOS Hardware Interrupt block installs an ISR function that signals a DSP/BIOS task to run when the ISR detects an RTDX interrupt. Signaling between the ISR and DSP/BIOS triggered task occurs via semaphores. This task receives an RTDX message carrying the location command for the downstream Text Insert block in the Text Overlay from the host computer.

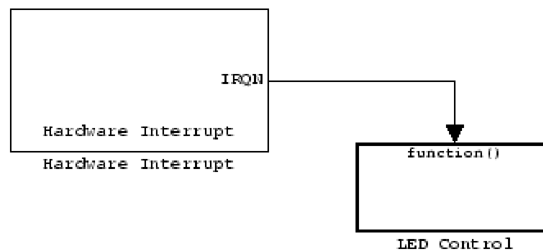
The blocks running inside the Location Command and Text Overlay subsystems are shown in the following figure.

The text overlay subsystem is executed as for the Free-Running DSP/BIOS Task. A Rate Transition block connects the two subsystems that run at two different asynchronous rates to ensure data integrity. The execution of two asynchronous rates is ordered based on the priority settings for the DSP/BIOS Task blocks.

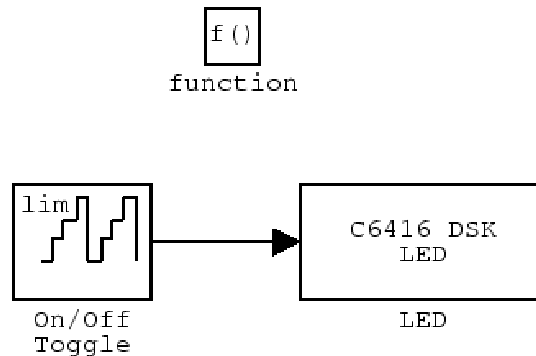




Hardware Interrupt Triggered Task. In the next figure, you see a case where a function (LED Control) runs in the context of a hardware interrupt triggered task.



In this model, the C6000 Hardware Interrupt block installs a task that runs when it detects an external interrupt. This task then toggles an external C6416DSK LED on or off.



Scheduling Considerations

When you use the DSP/BIOS task blocks for scheduling, either the DSP/BIOS Task block or the DSP/BIOS Triggered Task block, you must take care to avoid some common scheduling pitfalls.

First, the DSP/BIOS operating system always executes the task with the highest priority. Contrast this execution scheme with that of some other real-time operating systems (RTOS) where each task gets its fair share of processing time. Therefore, depending on the situation, there may be cases where lower-priority tasks never execute because a higher priority task is never blocked.

A DSP/BIOS task blocks only when a blocking device driver block is included in the function call subsystem the task is executing, such as ADC/DAC blocks and C6000 UDP Receive blocks. If a particular DSP/BIOS task executes a function call subsystem that does not include any device driver blocks, and this particular task has the highest priority, it never releases the CPU, effectively disabling all other lower priority tasks in the application.

For more information about asynchronous schedulers, refer to the “Handling Asynchronous Events” chapter in your Simulink Coder documentation in the online help system.

Model Reference and Embedded Coder Software

- "Overview" on page 56-34
- "How Model Reference Works" on page 56-34
- "Using Model Reference with Embedded Coder Software" on page 56-35
- "Configuring Targets to Use Model Reference" on page 56-37

Overview

Model reference lets your model include other models as modular components. This technique provides useful features because it:

- Simplifies working with large models by letting you build large models from smaller ones, or even large ones.
- Lets you generate code once for all the modules in the entire model and only regenerate code for modules that change.
- Lets you develop the modules independently.
- Lets you reuse modules and models by reference, rather than including the model or module multiple times in your model. Also, multiple models can refer to the same model or module.

Your Simulink Coder documentation provides much more information about model reference.

How Model Reference Works

Model reference behaves differently in simulation and in code generation. For this discussion, you need to know the following terms:

- Top model — The root model block or model. It refers to other blocks or models. In the model hierarchy, this is the topmost model.
- Referenced models — Blocks or models that other models reference, such as models the top model refers to. All models or blocks below the top model in the hierarchy are reference models.

The following sections describe briefly how model reference works. More details are available in your Simulink Coder documentation in the online help system.

Model Reference in Simulation. When you simulate the top model, Simulink Coder software detects that your model contains referenced models. Simulink generates code for the referenced models and uses the generated code to build shared library files for updating the model diagram and simulation. It also creates an executable (a MEX file, `.mex`) for each reference model that is used to simulate the top model.

When you rebuild reference models for simulations or when you run or update a simulation, Simulink rebuilds the model reference files. Whether reference files or models are rebuilt depends on whether and how you change the models and on the **Rebuild options** settings. You can access these setting through the **Model Reference** pane of the Configuration Parameters dialog box.

Model Reference in Code Generation. Simulink Coder software requires executables to generate code from models. If you have not simulated your model at least once, Simulink Coder software creates a `.mex` file for simulation.

Now, for each referenced model, the code generation process calls `make_rtw` and builds each referenced model. This build process creates a library file for each of the referenced models in your model.

After building all the referenced models, Simulink Coder software calls `make_rtw` on the top model, linking to all the library files it created for the associated referenced models.

Using Model Reference with Embedded Coder Software

With few limitations or restrictions, the Embedded Coder provides full support for generating code from models that use model reference.

Build Action Setting. The most important requirement for using model reference with the TI targets is that you must set the **Build action** (go to **Configuration Parameters > IDE Link**) for all models referred to in the simulation to `Archive_library`.

To set the build action

1. Open your model.
2. Select **Simulation > Configuration Parameters** from the model menus.

   The Configuration Parameters dialog box opens.
3. From the **Select** tree, choose `IDE Link`.
4. In the right pane, under **Runtime**, set **Build action** to `Archive_library`.

If your top model uses a reference model that does not have the build action set to `Archive_library`, the build process automatically changes the build action to `Archive_library` and issues a warning about the change.

As a result of selecting the `Archive_library` setting, other options are disabled:

- DSP/BIOS is disabled for all referenced models. Only the top model supports DSP/BIOS operation.
- **Overrun action**, **Overrun notification method**, **Exporting CCS object to the workspace**, and **Stack size** are all disabled for the referenced models.

Target Preferences Blocks in Reference Models. Each referenced model and the top model must include a Target Preferences block for the correct target. You must configure all the Target Preferences blocks for the same target.

To obtain information about which compiler to use and which archiver to use to build the referenced models, the referenced models require Target Preferences blocks. Without them, the compile and archive processes does not work.

By design, model reference does not allow information to pass from the top model to the referenced models. Referenced models must contain all the necessary information, which the Target Preferences block in the model provides.

Other Block Limitations. Model reference with Embedded Coder software does not allow you to use certain blocks or S-functions in reference models:

- No blocks from the C62x DSP Library (`tic64dsplib`) (because these are noninlined S-functions)
- No blocks from the C64x DSP Library (`tic62dsplib`) (because these are noninlined S-functions)
- No noninlined S-functions
- No driver blocks, such as the ADC or DAC blocks from any Embedded Coder library

Configuring Targets to Use Model Reference

Targets that you plan to use in Model Referencing must meet some general requirements.

- A model reference compatible target must be derived from the ERT or GRT targets.
- When you generate code from a model that references another model, you need to configure both the top-level model and the referenced models for the same code generation target.
- The External mode option is not supported in model reference Simulink Coder target builds. Embedded Coder software supports External mode, but not with model reference. If you select this option, it is ignored during code generation. For more information, please see the "Communicating With Code Executing on a Target System Using Simulink External Mode" chapter in the Simulink Coder User's Guide.
- To support model reference builds, your TMF must support use of the shared utilities folder, as described in Supporting Shared Utility folders in the Build Process.

To use an existing target, or a new target, with Model Reference, you set the `ModelReferenceCompliant` flag for the target. For information on how to set this option, refer to `ModelReferenceCompliant` in the online help system.

If you start with a model that was created prior to version 2.4 (R14SP3), to make your model compatible with the model reference target, use the following command to set the `ModelReferenceCompliant` flag to `On`:

```
set_param(bdroot,'ModelReferenceCompliant','on')
```

Models that you target with the Embedded Coder versions 2.4 and later automatically include the model reference capability. You do not need to set the flag.

Targeting Supported Boards

- "Overview" on page 56-38
- "Typical Targeting Process" on page 56-39
- "Targeting the C6713 DSP Starter Kit" on page 56-39
- "Configuring Your C6713DSK" on page 56-41
- "Confirming Your C6713DSK Installation" on page 56-42

Overview

Texas Instruments markets a complete set of tools for you to use with the a range of development boards, such as the C6713 DSK. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications. This section provides a brief example of how to use TI development tools with Simulink Coder software and the C6713 DSK blocks.

Executing code generated from Simulink Coder software on a particular target in real time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine. Other components, such as Embedded Coder software, are required if you need the ability to download parameters on the fly to your target hardware.

Since these components are specific to particular hardware targets (in this case, the C6713 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, Embedded Coder software provides a target makefile specific to the evaluation module. This target makefile invokes the optimizing compiler, provided as part of TI Code Composer Studio software.

Used in combination with Simulink Coder software, TI products provide an integrated development environment that, once installed, needs no additional coding.

Typical Targeting Process

Generally, targeting hardware, or a development environment as some call it, requires that you complete a series of processes that starts with building your model and ends with generating code to suit your target.

1. Build the Simulink model of your algorithm or process to be converted to code for your target.

2. Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters.

3. Add a Target Preferences block to your model.

4. Configure the options on the Target Preferences block to select the target, map memory segments, allocate sections to the memory segments, and configure other target-specific options.

5. Set the configuration parameters for your model. Notice that you do this step after you add the Target Preferences block to your model.

6. Build your model to your target.

Targeting the C6713 DSP Starter Kit

After you install the C6713 DSK development board and supporting TI products on your PC, start the MATLAB software. At the MATLAB command prompt, enter `c6713dsklib`. This opens a Simulink block library, `c6713dsklib`, that includes a set of blocks for C6713 DSK I/O devices, as described in the following table.

Block	Description
C6713 DSK ADC	Configure the analog to digital converter
C6713 DSK DAC	Configure the digital to analog converter

Block	Description
C6713 DSK LED	Control the user status LEDs on the C6713 DSK
C6713 DSK Reset	Reset the processor on the C6713 DSK

These blocks are associated with your C6713 DSK board. As needed, add the blocks to your model.

With your model open, select **Simulation > Configuration Parameters**. From this dialog box, select `Code Generation` from the **Select** tree. You must specify the appropriate versions of the system target file. For the C6713 DSK, in the `Code Generation` pane, specify **System target file** —`idelink_grt.tlc`

With this configuration, you can generate a real-time executable and download it to the TI C6713 evaluation board. You generate the executable by clicking **Build** on the `Code Generation` pane. The Simulink Coder software automatically generates C code and inserts the I/O device drivers as specified in your block diagram. These device drivers are inserted in the generated C code as inlined S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to Target Language Compiler Reference documentation. For a complete discussion of S-functions, refer to your Writing S-Functions documentation.

During the same build operation, the software invokes the TI compiler to build an executable file. If you select the `Build_and_execute` option, Simulink Coder software automatically downloads the executable to the TI evaluation board via the peripheral component interface (PCI) bus. After downloading the executable file to the C6713 DSK, the build process runs the file on the processor.

Starting and Stopping DSP Applications on the C6713 DSK. When you generate code, build the project, and download the code for your Simulink model to your C6713 DSK, you are running actual machine code corresponding to the block diagram you built in Simulink software. To start running your DSP application on the evaluation module, you must open your Simulink model and rebuild the machine executable by clicking **Build**. To start the application on the C6713 DSK, you use Simulink Coder software to rebuild the executable from the Simulink model and download the code to the board.

Your model runs until it encounters one of the following actions:

- You select **Debug > Halt** in CCS IDE.
- You shut down the host PC.
- The process encounters a Stop block in the model code.
- The running application encounters an error condition that stops the process.

If you included a Reset C6713 DSK block in your model, clicking the block stops the running application and restores the digital signal processor to its initial state.

Note When you build and execute a model on the C6713 DSK, the Simulink Coder build process resets the evaluation module automatically. You do not need to reset the board before building models. To stop processes that are running on the evaluation module, or to return the board to a known state for any reason, use the Reset C6713 DSK block.

Configuring Your C6713DSK

When you install the C6713DSK, set the dual inline pin (DIP) switches as shown in the following table. If you have installed the board with different settings, reconfigure the board. Refer to your *TMS320C6201/6713Evaluation Module User's Guide* for details.

DIP Switch	Name	Setting	Effect
SW2-1	BOOTMODE4	On	Boot mode setting
SW2-2	BOOTMODE3	On	Boot mode setting
SW2-3	BOOTMODE2	Off	Sets memory map = 1 when SW2-5 is off
SW2-4	BOOTMODE1	On	Boot mode setting
SW2-5	BOOTMODE0	Off	Sets memory map =1 when SW2-3 is off

DIP Switch	Name	Setting	Effect
SW2-6	CLKMODE	On	Sets multiply-by-4 mode
SW2-7	CLKSEL	On	Selects oscillator A
SW2-8	ENDIAN	On	Selects little endian mode
SW2-9	JTAGSEL	Off	Selects internal Test Bus Controller (TBC)
SW2-10	USER2	On	User-defined option
SW2-11	USER1	On	User-defined option
SW2-12	USER0	On	User-defined option

Confirming Your C6713DSK Installation

Texas Instruments supplies a test utility to verify the operation of the board and its associated software. For complete information about running the test utility and interpreting the results, refer to your *TMS320C6201/6713 DSP Starter Kit User's Guide*.

To run the C6713 DSK verification test, complete the following steps after you install your board:

1. Start CCS IDE.

2. Select **Start > Programs > Code Composer Studio > DSK Confidence Test**. As the test runs, the results appear on your display.

   By default, the test utility does not create a log file to store the test results. To specify the name and location of a log file to contain the results of the confidence test, use the command line options in CCS IDE to run the confidence test utility. For further information about running the verification test from a DOS window and using the command line options, refer to *TMS320C6201/6713 Evaluation Module User's Guide.*

3. Review the test results to verify that everything works. Check that the options settings match the settings listed in the table above.

If your options settings do not match the configuration shown in the preceding table, reconfigure your C6713 DSK. After you change your board configuration, rerun the verification utility to check your new settings.

Simulink Models and Targeting

Creating Your Simulink Model for Targeting

You create real-time digital signal processing models the same way you create other Simulink models—by combining standard DSP blocks and C-MEX S-functions.

You add blocks to your model in several ways:

- Use blocks from the DSP System Toolbox software
- Use blocks from the fixed-point blocks library TI C62x DSPLIB or TI C64x DSPLIB
- Use other Simulink discrete-time blocks
- Use the blocks provided in the C6000 blockset: ADC, DAC, LED and Reset blocks for specific supported target hardware
- Use blocks that provide the functions you need from any blockset installed on your computer
- Create and use custom blocks

Once you have designed and built your model, you generate C code and build the real-time executable by clicking **Build** on the **Code Generation** pane of the Configuration Parameters dialog box. The automatic build process creates the file `modelname.out` containing a real-time model image in COFF file format that can run on your target.

The file `modelname.out` is an executable whose format is target-specific. You can load the file to your target and execute it in real time. Refer to your Simulink Coder documentation for more information about the build process.

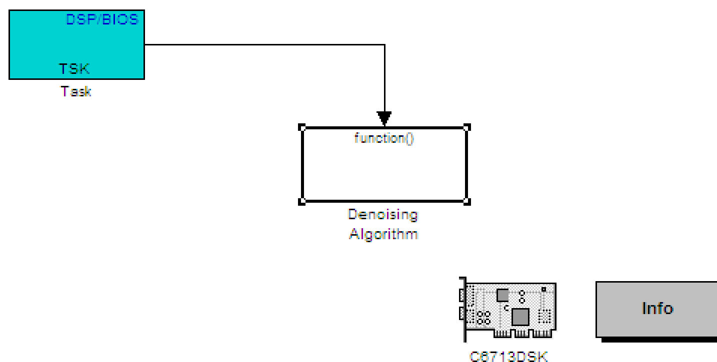
Targeting Tutorial II — A More Complex Application

- "Overview" on page 56-44

- “Working and Build folders” on page 56-45
- “Setting Simulation Program Parameters” on page 56-45
- “Selecting the Target Configuration” on page 56-46
- “Building and Running the Program” on page 56-49
- “Contents of the Build folder” on page 56-50

Overview

For this tutorial, we demonstrate an application that uses multiple stages—using wavelets to remove noise from a noisy signal. Open the demo model, `c6713dskwdnoisf`. As with any model file, you can run this denoising demonstration by typing `c6713dskwdnoisf` at the MATLAB prompt. The model also appears in the MATLAB demos collection in the Help browser—under Simulink demos, in the Embedded Coder category. Here is a picture of the model as it appears in the demonstration library.



Unlike the audio reverberation demo, this model is difficult to build from blocks in Simulink software. It uses complex subsystems for the Delay Alignment block and the Soft Threshold block. For this tutorial, you work with a copy of the demonstration model, rather than creating the model.

This tutorial takes you through generating C code and building an executable program from the demonstration model. The resulting program runs on your C6713 DSK as an executable COFF file.

Working and Build folders

It is convenient to work with a local copy of the `c6713dskwdnoisf` model, stored in its own folder, which you named (something like `c6713dnoisfex`). This discussion assumes that the `c6713dnoisfex` folder resides on drive `d:`. Use a different drive letter if necessary for your machine. Set up your working folder as follows:

1. Create the new model folder from the MATLAB command line by typing

   ```
   !mkdir d:\c6713dnoisfex (on PC)
   ```

2. Make `c6713dnoisfex` your working folder.

   ```
   cd d:/c6713dnoisfex
   ```

3. Open the `c6713dskwdnoisf` model.

   ```
   c6713dskwdnoisf
   ```

   The model appears in the Simulink window.

4. From the **File** menu, choose **Save As**. Save a copy of the `c6713dskwdnoisf` model as `d:/c6713dnoisfex/dnoisfrtw.mdl`.

During code generation, Simulink Coder software creates a build folder within your working folder. The build folder name is `model_target_rtw`, derived from the name of your source model and your chosen target. In the build folder, Simulink Coder software stores generated source code and other files created during the build process. You examine the contents of the build folder at the end of this tutorial.

Setting Simulation Program Parameters

To generate code correctly from the `dnoisfrtw` model, you must change some of the configuration parameters. In particular, Simulink Coder software uses a fixed-step solver. To set the parameters, use the Configuration Parameters dialog box as follows:

1. From the **Simulation** menu, choose **Configuration Parameters**. The Configuration Parameters dialog box opens.

2. Click `Solver` and enter the following parameter values on the **Solver** pane. Note that Embedded Coder software does not honor a stop time if you set one here.

   Start Time: `0.0`

   Stop Time: `inf`

   Solver options: set **Type** to `Fixed-step`. Select the `Discrete` solver algorithm. (Targeting does not work with continuous time solvers.)

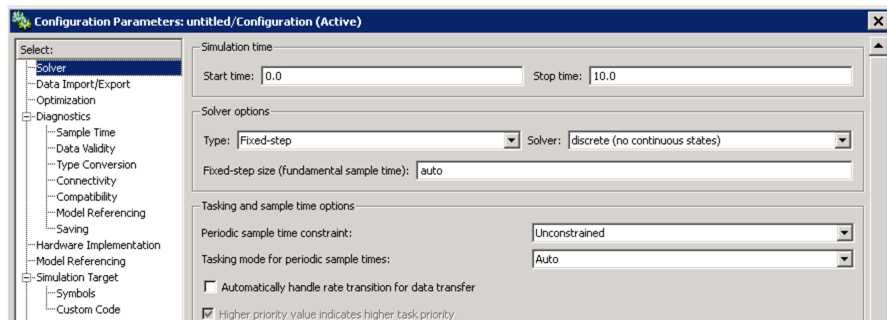
   Fixed step size: `auto`

   Tasking mode for periodic sample times: `Auto`

3. Click **Apply**, and then click **OK** to close the dialog box.

4. Save the model. Configuration parameters persist with the model (as the model configuration set), for you to use in future sessions.

In the next figure you see the `Solver` pane with the correct parameter settings.



Selecting the Target Configuration

To specify the desired target configuration, choose the **System target file**.

In these tutorials, you do not need to specify these parameters individually. Instead, you use the ready-to-run `idelink_grt.tlc` target configuration.

Note The `Code Generation` category has several subcategories that you select using the **Select** tree in the Configuration Parameters dialog box. During this tutorial you change or review options in just a few of the categories in the tree.

To target your C6713 DSK:

1. From the **Simulation** menu, choose **Configuration Parameters**. The Configuration Parameters dialog box opens.

2. Click `Code Generation` on the **Select** tree. The `Code Generation` pane activates.

3. Click **Browse** next to the **System target file** field. This opens the **System Target File Browser**. The browser displays a list of available target configurations. When you select a target configuration, Simulink Coder software automatically chooses the appropriate system target file.

4. From the list of available configurations, select `idelink_grt.tlc`, and click **OK**.

5 To decide whether to export a CCS handle to your MATLAB workspace when you generate code, or run your model, select `IDE Link` from the **Select** tree.

6 Set the **Runtime** and **Project options** as shown in the preceding figure.

7 To export the handle (a variable) that CCS IDE creates when you generate code from your model, select **Export IDE link handle to base workspace**, and enter a name for the handle in **IDE link handle name**.

8 Select `Optimization` from the **Select** tree. A new set of options appears. The options displayed here are common to all target configurations. Make sure that all options are set to their defaults, as shown in the following figure.

9 Click **OK** to close the Configuration Parameters dialog box. Save the model to retain your new build settings.

Building and Running the Program

The Simulink Coder build process generates C code from your model, and then compiles and links the generated program.

To build and run your program:

1. Access the Configuration Parameters dialog box for your model.

2. Click **Build** in the `Code Generation` pane to start the build process.

3. A number of messages concerning code generation and compilation appear in the MATLAB workspace. The initial messages are

   ```
   ### Starting Simulink Coder build procedure for model:
   dnoisfrtw
   ### Generating code into build folder: .\dnoisfrtw_c6000_rtw
   ```

   The content of the succeeding messages depends on your compiler and operating system. The final message is

   ```
   ### Successful completion of Simulink Coder build procedure
   for model: dnoisfrtw
   ```

4. The working folder now contains an executable, `dnoisfrtw.exe`. In addition, Simulink Coder software created a build folder, dnoisfrtw_c6000_rtw.

   To review the contents of the working folder after the build, type the `dir` command at the MATLAB command prompt.

   ```
   dir
   .              dnoisfrtw.exe     dnoisfrtw_c6000_rtw
   ..             dnoisfrtw.mdl
   ```

5. To run the executable from the MATLAB command prompt, type

   ```
   !dnoisfrtw
   ```

   The "`!`" character passes the command that follows it to the operating system, which runs the stand-alone `dnoisfrtw` program.

   The program produces one line of output.

```
**starting the model**
```

6 To see the contents of the build folder, type

```
dir dnoisfrtw_c6713_rtw
```

Contents of the Build folder

The build process creates a build folder and names it `model_target_rtw`, concatenating the name of your source model and your chosen target. In this example, your build folder is named dnoisfrtw_c6713_rtw.

dnoisfrtw_c6713_rtw contains these generated source code files:

- `dnoisfrtw.c` — The stand-alone C code that implements the model.
- `dnoisfrtw.h` — An include header file containing information about the state variables
- `dnoisfrtw_export.h` — An include header file containing information about exported signals and parameters

The build folder also contains other files used in the build process, such as the object (`.obj`) files and the generated makefile (`dnoisfrtw.mk`).

Targeting Your C6713 DSK and Other Hardware

- "Overview" on page 56-50
- "Confirming Your C6713 DSK Installation" on page 56-51
- "Running Models on Your C6713 DSK" on page 56-52

Overview

Embedded Coder software lets you use Simulink Coder software to generate, target, and execute Simulink models on the Texas Instruments (TI) C6713 DSP Starter Kit (C6713 DSK). In combination with the C6713 DSK, your the Embedded Coder is the ideal resource for rapidly prototyping and developing embedded systems applications for the TI C6713 Digital Signal Processor. The Embedded Coder focuses on developing real-time digital signal processing (DSP) applications for the C6713 DSK.

This chapter describes how to use the Embedded Coder to create and execute applications on the C6713 DSK. To use the targeting software, you should be familiar with using Simulink to create models and with the basic concepts of Simulink Coder software automatic code generation. To read more about Simulink Coder software, refer to your Simulink Coder documentation.

Confirming Your C6713 DSK Installation

Texas Instruments supplies a test utility to verify operation of the board and its associated software. For complete information about running the test utility and interpreting the results, refer to your "TMS320CDSK Help" under TMS320C6000 Code Composer Studio Help in the CCS online help system.

To run the C6713 DSK confidence test, complete the following steps after you install and configure your board.

1. Open a DOS command window.

2. Access the folder `\..\ti\c6000\dsk6x11\conftest`

   CCS IDE creates this folder when you install it. It contains the files to run the C6713 confidence test.

3. Start the confidence test by typing `dsk6xtst` at the DOS prompt.

   By default, the test utility creates a log file named `dsk6xtst.log` where it stores the test results. To specify the name and location of a log file to contain the results of the confidence test, use the CCS IDE command line options to run the confidence utility. For further information about running the confidence test from a DOS window and using the command line options, refer to the "DSK Confidence Test" topic in the CCS IDE online help.

4. Review the test results to verify that everything works.

If your confidence test fails, reconfigure your C6713 DSK. After you change your board configuration, rerun the confidence utility to check your new settings.

Running Models on Your C6713 DSK

Texas Instruments markets a complete set of tools for use with the C6713 DSK. These tools are primarily intended for rapid prototyping of control systems and hardware-in-the-loop applications.

This section provides a brief example of how the TI development tools work with Simulink Coder software, the Embedded Coder, and the C6713 DSK block library.

Executing code generated from Simulink Coder software on a particular target in real-time requires target-specific code. Target-specific code includes I/O device drivers and an interrupt service routine.

Other components, such as Embedded Coder software, are required if you need the ability to download parameters on-the-fly to your target hardware.

Since these components are specific to particular hardware targets (in this case, the C6713 DSK), you must ensure that the target-specific components are compatible with the target hardware.

To allow you to build an executable, the Embedded Coder provides a target makefile specific to C6000 hardware targets. This target makefile invokes the optimizing compiler provided as part of CCS IDE.

Used in combination with the Embedded Coder and Simulink Coder software, TI products provide an integrated development environment that, once installed, needs no additional coding.

After you have installed the C6713 DSK development board and supporting TI products on your PC, start the MATLAB software. At the MATLAB command prompt, type `c6713dsklib`. This opens a Simulink block library, `c6713dsklib`, that includes a set of blocks for C6713 DSK I/O devices:

- C6713 DSK ADC — Configure the analog to digital converter
- C6713 DSK DAC — Configure the digital to analog converter
- C6713 DSK LED — Control the user-defined light emitting diodes (LED) on the C6713 DSK

- C6713 DSK DIP Switch — Set the dual inline pin switches on the C6713 DSK
- C6713 DSK Reset — Reset the processor on the C6713 DSK

These devices are associated with your C6713 DSK board.

With your model open, select **Simulation > Configuration Parameters** from the menu bar to open the Configuration Parameters dialog box.

From this dialog box, click **Code Generation** on the select tree. You must specify the appropriate versions of the system target file. For the C6713 DSK, in the **Code Generation** pane of the dialog box, specify **System target file** — `idelink_grt.tlc`

With this configuration, you can generate and download a real-time executable to your TI C6713 DSK. Start the Simulink Coder build process by clicking **Build** on the **Code Generation** pane. Simulink Coder software automatically generates C code and inserts the I/O device drivers as specified by the ADC and DAC blocks in your block model.

These device drivers are inserted in the generated C code as inlined S-functions. Inlined S-functions offer speed advantages and simplify the generated code. For more information about inlining S-functions, refer to your Target Language Compiler documentation. For a complete discussion of S-functions, refer to your documentation about writing S-functions.

During the same build operation, the software invokes the TI compiler to build an executable file.

If you select the `Build_and_execute` option, the executable file is automatically downloaded via the peripheral component interface (PCI) bus to the TI evaluation board. After downloading the executable file to the C6713 DSK, the build process runs the file on the digital signal processor.

Starting and Stopping DSP Applications on the C6713 DSK. When you create, build, and download a Simulink model to the C6713 DSK, you are not running a simulation of your DSP application. You are running the actual machine code corresponding to the block diagram you built in Simulink software. To start running your DSP application on the evaluation module, you must open your Simulink model and rebuild the machine executable by clicking **Build** on the **Code Generation** pane. Each time you want to start the application on the C6713 DSK, you use Simulink Coder software to rebuild the executable from the Simulink model and download the code to the board.

Your model runs until the model encounters one of the following actions:

- Using the **Debug > Halt** option in CCS IDE
- Using halt from the MATLAB command prompt
- Encountering a Stop block in the model.
- Clicking the C6713 DSK Reset block in your model (if you added one) or in the DSK block library

Clicking the Reset block stops the running application and restores the digital signal processor to its initial state.

Creating Code Composer Studio Projects Without Building

- "Introduction" on page 56-54
- "Creating Projects in CCS IDE Without Loading Files to Your Target" on page 56-55

Introduction

Rather than targeting your C6000 board when you build your signal processing application, you can create Texas Instruments Code Composer Studio (CCS) IDE projects. Creating projects for CCS IDE lets you use the tools provided by the CCS IDE software suite to debug your real-time process.

If you build and download your Simulink model to CCS IDE, Embedded Coder software opens Code Composer Studio software, creates a new CCS IDE

project named for your model, and populates the new project with all the files it creates during the build process—the object code files, the assembly language files, the map files, and any other necessary files. As a result, you can immediately use CCS IDE to debug your model using the features provided by the CCS IDE.

Creating a project in CCS IDE is the same as targeting C6000 hardware. You configure your target options, select your build action to create a CCS IDE project, and then build the project in CCS IDE by clicking **Make Project**.

Creating Projects in CCS IDE Without Loading Files to Your Target

From the **Select** tree in the Configuration Parameters dialog box, under `Code Generation`, select `IDE Link`. Select `Create_Project` for the **Build action**, as shown in the next figure. The `Build` and `Build_and_execute` options create CCS IDE projects as well. The `Archive_library` option does not create a CCS IDE project. None of the other options has an effect here. Ignore them when you are creating a project in CCS IDE rather than generating code.

After you select `Create_CCS_Project`, set the options for the **Code Generation** options on the `IDE Link` category on the **Select** tree.

Return to the Simulink Coder category, clear **Generate code only** and click **Build** to build your new CCS IDE project.

Simulink Coder software and Embedded Coder software generate all the files for your project in CCS IDE and create a new project in the IDE. Your new project is named for the model you built, with a custom project build configuration `CustomMW`, not `Release` or `Debug`.

In CCS IDE you see your project with the files in place in the folder tree.

Targeting Custom Hardware

- "Overview" on page 56-56
- "Typical Targeting Process" on page 56-57
- "Targeting a Custom Target" on page 56-59

- "Section Pane" on page 56-64
- "To Create Memory Maps for Targets" on page 56-68

Overview

As long as the processor on your custom board is from the TI C6000 DSP family, you can use Embedded Coder software to generate code for your target.

The blocks for the peripherals in the C6000 DSP Library, such as the C6416 DSK ADC or C6713 DSK DAC blocks, are specific to their hardware and will not work with your custom board. None of the board-specific blocks provided by this toolbox work with custom hardware.

The Target Preferences block provides a way to target boards that are not specifically supported. Due to certain features related to memory maps and other processor-specific attributes, custom hardware targeting only works with the C6000 DSPs.

Several guidelines affect your targeting configuration decisions when you decide to use custom targets and the custom Target Preferences block:

1. Specify the memory allocation (memory mapping) using the **Memory** and **Section** panes on the Target Preferences dialog box. Set the memory mapping for your target that best matches your hardware. For example, if your custom target uses the C6713 processor, be sure your memory configuration is the same as the one on the supported C6713 DSK, such as has the same memory size, the same EMF settings, the same memory sections, and the same cache organization.

2. To use on-chip memory only for your target, choose the `Near_Calls` setting for the **Memory model** in the **TI C6000 compiler** options. To use external memory that is specific to your board, choose the `Far_Calls` setting for the **Memory model**. The other selection in the **Memory model** list offers a combination of near and far allocation for data and aggregate data.

3. Do not use the existing ADC, DAC, DIP Switch, or LED blocks unless you are quite sure that your hardware is identical to the appropriate EVM or DSK in all important respects. Generally, the ADC, DAC, and other target-specific blocks are design specifically for their designated targets and can cause problems when you use them on hardware that is not identical.

4. Set the **Overrun notification method** in the **TI C6000 runtime** category to `Print_message` when you use the overrun notification feature. If you choose to use the LED notification option, verify that on your specialized target you access the LEDs in exactly the same way, and the LEDs respond in the same way, as the LEDs on the corresponding supported DSK or EVM.

To use one of the custom targets, create your model, add and configure the Target Preferences block, and then open the Configuration Parameters dialog box for the model.

Typical Targeting Process

Generally, targeting hardware, or a development environment as it is called by some, requires that you complete a series of processes that starts with building your model and ends with generating code to suit your target.

1. Build the Simulink model of your algorithm or process to be converted to code for your target.

2. Add target-specific blocks to your model, such as ADC and DAC blocks, and configure the block parameters. (Skip this step when you are targeting a processor on a custom board.)

3. Add a Target Preferences block to your model. The top level of the model must contain a Target Preferences block.

4. Configure the options on the Target Preferences block to select the target, map memory segments, allocate code and data sections to the memory segments, and set other target-specific options.

5. Set the Simulink configuration parameters for your model. Notice that you do this after you add the Target Preferences block to your model.

6. Build your model to your target.

Memory Maps. Memory maps are an essential part of targeting any processor or board. Without the map, the code generation process cannot determine where various features of the generated code, such as variables, data, and executable code, reside on the target.

To discuss memory maps and configuring memory, a few terms need to be defined:

- Memory map — Map of the memory space for a target system. The memory space is partitioned into functional blocks.
- memory segment — Memory partition that corresponds to a physical range of memory on the target. The segment is named in some fashion, such as IPRAM or SDRAM.
- Memory section — The smallest unit of an object file. This is a block of data or code that, based on the memory map, resides in an area of contiguous memory on the target and in the memory map. Sections of object files are both distinct and separate. Memory sections come in two flavors:
  - Uninitialized sections that reserve memory space for uninitialized data. One example of an uninitialized section is `.bss`. The `.bss` section reserves space for variables that are not initialized.
  - Initialized sections contain code and data. The `.text` (containing executable code) and `.data` (containing initialized data) sections are initialized.
- Memory management — Process of specifying the memory segments that the various memory sections use for your application. A logical memory map of the hardware memory results from the process of managing memory.

During code generation, the linker and assembler work to allocate your code and data into the memory on your target according to the memory map specifications you provide. For more information about memory utilization and memory management, refer to the CCS IDE online help, using keywords like memory map, memory segment, and section.

The compiler does not interact with the memory map. It makes no assumptions about memory allocation and is not aware of the memory map. As far as the C6000 compiler is concerned, the physical memory on your target is one continuous linear block of memory that is subdivided into smaller blocks containing code, data, or both.

When you configure the block parameters for the Target Preferences block, you are setting up the memory map for your target. You specify the memory segments that are defined and the contents of each segment. You specify the

sections, both named and default, and the segments to which the sections are assigned.

These memory management functions are identical to the ones available in the CCS IDE Configuration Tool.

Targeting a Custom Target

To use a board that has a TI C6000 processor but is not one of the supported boards, configure the Target Preferences block as described in this section.

Configuring the block parameters software about your target processor and how to generate code that will run on the target.

1. Add the Target Preferences block to your model, or edit the current Target Preferences block.
2. Set **Board** to `C6000 Custom`.
3. Select your target processor from the **Processor** list. Most of the C6000 family of DSP processors are on the list. If the one you need is not listed, pick one that closely matches your target.
4. Set the actual CPU clock rate for the CPU on your target in CPU clock speed (MHz). Report the clock speed of the processor on your target. When you enter a value, you are not changing the CPU clock rate, you are reporting the actual rate. If the value you enter does not match the rate on the target, your model real-time results might be wrong, and code profiling results will not be correct. You must enter the actual clock rate the board uses. The rate you enter here does not change the rate on the board. Setting **CPU clock** to the actual board rate allows the code you generate to run correctly according to the actual clock rate of the hardware.
5. If your target is a simulator rather than a hardware target, use the **Get from IDE** button on the **Board** pane of the Target Preferences block. Then set **Board Name** under **IDE Support** to one of the simulators installed with your IDE.
6. To enable the Embedded Coder to connect to CCS IDE, select your target from the **Board Name** list. On this list you see the names of the boards you have configured in the CCS Setup Utility. If your target board does

not appear on the list, start CCS Setup and add your board to the System Configuration dialog box.

7 Select the processor to target from the **Processor Name** list. For the board you selected in **Board Name**, **Processor Name** lists all the processors on the board. The list comes from the processors you added to the board in the CCS Setup Utility.

Now you have completed the process of identifying your target to the Embedded Coder and Simulink Coder software. While this process is necessary, it represents only one small part of enabling you to generate code to run on your custom board.

One very important part of targeting custom hardware is to provide the target memory map configuration to the linker and assembler.

Memory and **Section** panes on the Target Preferences dialog box provide the controls required to specify how the linker and assembler arrange the code, data, and variables on your target.

Memory Pane. The information that follows describes the options on the panes in detail.

The **Memory** pane contains memory options in three areas:

- **Physical Memory** specifies the mapping for processor memory
- **Heap** specifies whether you use a heap and determines the size in words
- **L2 Cache** enables the L2 cache (where available) and sets the size in kB

Be aware that these options can affect the options on the **Section** pane. You can make selections here that change how you configure options on the **Section** pane.

Most of the information about memory segments and memory allocation is available from the Code Composer Studio online help.

Physical Memory Options. This list shows the physical memory segments available on the board and processor. By default, Target Preferences blocks show the memory segments found on the selected processor. In addition, the **Memory** pane on preconfigured Target Preferences blocks shows the memory segments available on the board, but off of the processor. Target preferences blocks set default starting addresses, lengths, and contents of the default memory segments.

The default memory segments for each processor and board are different. For example:

- Custom boards based on C670x processors provide IPRAM and IDRAM memory segments by default.
- C6713 DSK boards provide SDRAM memory segment by default

Name. When you highlight an entry on the **Physical memory** list, the name of the entry appears here. To change the name of the existing memory segment, select it in the Physical memory list and then type the new name here.

Note You cannot change the names of default processor memory segments.

To add a new physical memory segment to the list, click **Add**, replace the temporary label in **Name** with the one to use, and press **Return**. Your new segment appears on the list.

After you add the segment, you can configure the starting address, length, and contents for the new segment. New segments start with code and data as the type of content that can be stored in the segment (refer to the **Contents** option).

Names are case sensitive. `NewSegment` is not the same as `newsegment` or `newSegment`.

Address. **Address** reports the starting address for the memory segment showing in **Name**. Address entries are in hexadecimal format and limited only by the board or processor memory.

When you are using a processor-specific preferences block, the starting address shown is the default value. You can change the starting value by entering the new value directly in **Address** when you select the memory segment to change.

Length. From the starting address, **Length** sets the length of the memory allocated to the segment in **Name**. As in all memory entries, specify the length in hexadecimal format, in minimum addressable data units (MADUs). For the C6000 processor family, the MADU is 8 bytes, one word.

When you are using a processor-specific preferences block, the length shown is the default value. You can change the value by entering the new value directly in this option.

Contents. **Contents** describes the kind of program sections that you can store in the memory segment in **Name**. As the processor type for the Target Preferences block changes, the kinds of information you store in listed memory segments can change. Generally, the **Contents** list contains these strings:

- `Code` — Allow code to be stored in the memory segment in **Name**.
- `Data` — Allow data to be stored in the memory segment in **Name**.
- `Code and Data` — Allow code and data to be stored in the memory segment in **Name**. When you add a new memory segment, this is the default setting for the contents of the new element.

You can add or use as many segments of each type as you need, within the limits of the memory on your processor.

Add. Click **Add** to add a new memory segment to the target memory map. When you click **Add**, a new segment name appears, for example `NEWMEM1`, in **Name** and on the **Physical memory** list. In **Name**, change the temporary name `NEWMEM1` by entering the new segment name. Entering the new name, or clicking **Apply** updates the temporary name on the list to the name you enter.

Remove. This option lets you remove a memory segment from the memory map. Select the segment to remove in the **Physical memory** list and click **Remove** to delete the segment.

Create Heap. If your processor supports using a heap, as does the C6713, for example, selecting this option enables creating the heap and enables the **Heap size** option. **Create heap** is not available on processors that either do not provide a heap or do not allow you to configure the heap.

Using this option you can create a heap in any memory segment on the **Physical memory** list. Select the memory segment on the list and then select **Create heap** to create a heap in the select segment. After you create the heap, use the **Heap size** and **Define label** options to configure the heap.

The location of the heap in the memory segment is not under your control. The only way to control the location of the heap in a segment is to make the segment and the heap the same size. Otherwise, the compiler determines the location of the heap in the segment.

Heap Size. After you select **Create heap**, this option lets you specify the size of the heap in words. Enter the number of words in decimal format. When you enter the heap size in decimal words, the system converts the decimal value to hexadecimal format. You can enter the value directly in hexadecimal format as well. Processors may support different maximum heap sizes.

Define Label. Selecting **Create heap** enables this option that allows you to name the heap. Enter your label for the heap in the **Heap label** option.

Heap Label. Selecting **Define label** enables this option. You use **Heap Label** to provide the label for the heap. Any combination of characters is accepted for the label except reserved characters in C/C++ compilers.

Enable L2 Cache. C621x, C671x, and C641x processors support an L2 cache memory structure that you can configure as SRAM and partial cache. Both the data memory and the program share this second-level memory. C620x DSPs do not support L2 cache memory, and this option is not available when you choose one of the C620x processors as your target.

If your processor supports the two-level memory scheme, this option enables the L2 cache on the processor.

L2 Cache Size. After you enable the L2 cache, select the size of the cache from the list.

Section Pane

Options on this pane let you specify where various program sections should go in memory. Program sections are distinct from memory segments—sections are portions of the executable code stored in contiguous memory locations. Among the sections used generally are `.text`, `.bss`, `.data`, and `.stack`. Some sections relate to the compiler, some to DSP/BIOS, and some can be custom sections as you require.

For more information about program sections and objects, refer to the CCS IDE online help. Most of the definitions and descriptions in this section come from CCS IDE.

In the pane shown in the preceding figure, you configure the allocation of sections for **Compiler**, **DSP/BIOS**, and **Custom** needs.

This table provides brief definitions of the various kinds of sections in the **Compiler**, **DSP/BIOS**, and **Custom** lists. All sections do not appear on both lists. The string appears on the list shown in the table.

String	Section List	Description of the Section Contents
`.args`	DSP/BIOS	Argument buffers
`.bss`	Compiler	Static and global C variables in the code
`.bios`	DSP/BIOS	DSP/BIOS code if you are using DSP/BIOS options in your program
`.cinit`	Compiler	Tables for initializing global and static variables and constants
`.cio`	Compiler	Standard I/O buffer for C programs
`.const`	Compiler	Data defined with the C qualifier and string constants
`.data`	Compiler	Program data for execution
`.far`	Compiler	Variables, both static and global, defined as far variables

String	Section List	Description of the Section Contents
.gblinit	DSP/BIOS	Load allocation of the DSP/BIOS startup initialization tables section
.hwi	DSP/BIOS	Dispatch code for interrupt service routines
.hwi_vec	DSP/BIOS	Interrupt Service Table
.obj	DSP/BIOS	Configuration properties that the target program can read
.pinit	Compiler	Load allocation of the table of global object constructors section.
.stack	Compiler	The global stack
.switch	Compiler	Jump tables for switch statements in the executable code
.sysdata	DSP/BIOS	Data about DSP/BIOS
.sysinit	DSP/BIOS	DSP/BIOS initialization startup code
.sysmem	Compiler	Dynamically allocated object in the code containing the heap
.text	Compiler	Load allocation for the literal strings, executable code, and compiler generated constants
.trcdata	DSP/BIOS	TRC mask variable and its initial value section load allocation

You can learn more about memory sections and objects in your Code Composer Studio online help.

Compiler Sections. During program compilation, the C6000 compiler produces both uninitialized and initialized blocks of data and code. These blocks are allocated into memory as required by the configuration of your system. On the **Compiler Sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections. The initialized sections are

- `.cinit`
- `.const`
- `.switch`
- `.text` (created by the assembler)

These sections are uninitialized:

- `.bss` (created by the assembler)
- `.far`
- `.stack`
- `.sysmem`

Other sections appear on the list as well:

- `.data` (created by the assembler)
- `.cio`
- `.pinit`

Note The C/C++ compiler does not use this section.

When you highlight a section on the list, **Description** shows a brief description of the section. Also, **Placement** shows you where the section is currently allocated in memory.

Description. Provides a brief explanation of the contents of the selected entry in the **Compiler Sections** list.

Placement. Shows you where the selected **Compiler Sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments as defined in the physical memory map on the **Memory** pane. Select one of the listed memory segments to allocate the highlighted compiler section to the segment.

DSP/BIOS Sections. During program compilation, DSP/BIOS produces both uninitialized and initialized blocks of data and code. These blocks get allocated into memory as required by the configuration of your system. On the **DSP/BIOS sections** list you find both initialized (sections that contain data or executable code) and uninitialized (sections that reserve space in memory) sections.

Description. Provides a brief explanation of the contents of the selected **DSP/BIOS Sections** list entry.

Placement. Shows where the selected **DSP/BIOS Sections** list entry is allocated in memory. You change the memory allocation by selecting a different location from the **Placement** list. The list contains the memory segments available on C6000 processors, and changes based on the processor you are using.

DSP/BIOS Object Placement. Distinct from the entries on the **DSP/BIOS Sections** list, DSP/BIOS objects like STS or LOG, if your project uses them, are placed in the memory segment you select from the **DSP/BIOS Object Placement** list. All DSP/BIOS objects use the same memory segment. You cannot select the locations for individual objects.

Custom Sections. When your program uses code or data sections that are not included in either the **Compiler Sections** or **DSP/BIOS Sections** lists, you add the new sections to this list. Initially, the **Custom Sections** list contains no fixed entries, just a placeholder for a section for you to define.

Name. You enter the name for your new section here. To add a new section, click **Add**. Then replace the temporary name with the name to use. Although the temporary name includes a period at the beginning, you do not need to include the period in your new name. Names are case sensitive. `NewSection` is not the same as `newsection`, or `newSection`.

Placement. With your new section added to the **Name** list, select the memory segment to which to add your new section. Within the restrictions imposed by the hardware and compiler, you can select any segment that appears on the list.

Add. Clicking **Add** lets you configure a new entry to the list of custom sections. When you click **Add**, the block provides a new temporary name in **Name**. Enter the new section name to add the section to the **Custom Sections** list. After typing the new name, click **Apply** to add the new section to the list. Or click **OK** to add the section to the list and close the dialog box.

Remove. To remove a section from the **Custom Sections** list, select the section to remove and click **Remove**. The selected section disappears from the list.

To Create Memory Maps for Targets

Although each processor has memory map requirements, the C6000 DSP family of processors share some memory features and not others. Details of the memory sections and segments, as well as memory allocations and limitations for each processor, are provided in your documentation for CCS IDE and from TI.

To manage the memory on your processor, set the options within these panes to specify the memory allocation to use. Recall that the memory map is the result of the settings you provide for the options in the **Memory** and **Section** panes in the Target Preferences dialog box.

Unfortunately, each processor has different needs, and the differences make it impossible to provide details about how you set the options for your target. You determine, from your model and code

- What memory segments you require
- Which sections you need and where
- Whether you need custom memory segments and sections
- Where to begin each memory segment and how much memory to allot to each segment
- Any other information that you need to set the options on the **Memory** and **Section** panes?

After you configure the options in the Target Preferences dialog box, you are ready to set the Simulink configuration parameters for your model and generate code.

Using Embedded Coder Software

- “Introduction” on page 56-69
- “To Use the Embedded Coder Target File” on page 56-69

Introduction

To take advantage of Embedded Coder software features, you must migrate your models to a system target file called `idelink_ert.tlc`. This target is based on the embedded real-time target (ERT) used by Embedded Coder software. Other Embedded Coder target files are based on the generic real-time target (GRT).

To use Embedded Coder software, choose the system target file `idelink_ert.tlc`, available in the **System Target File Browser**.

If you simply choose the system target file `idelink_ert.tlc` in the **System Target File Browser** directly to change the target for the model, all the **IDE Link** options are reset to default values by the switch. The C6000-specific options are the same between the two system target files.

You can set your model to use this system target file the usual way, via the **System Target File Browser**, available from the **Simulink Coder** pane in the Configuration Parameters dialog box. However, when you use the system target browser to switch your model between the ERT- and GRT-based TI C6000 system target files, the TI C6000-specific options (the configuration set) for the model are reset to default values.

To Use the Embedded Coder Target File

For setting up a new model to use the ERT-based target `.tlc` file.

1. From your model menu bar, select **Simulation > Configuration Parameters**.

2. Click `Code Generation` on the **Select** tree to access the Simulink Coder software options.

3. Click **Browse** to open the **System Target File Browser**.

4. On the **System Target File Browser**, find and select the file `idelink_ert.tlc`.

5. Click **OK**.

Targeting with DSP/BIOS Options

In this section...

"Introducing DSP/BIOS" on page 56-71

"DSP/BIOS and Targeting Your C6000 DSP" on page 56-72

"Code Generation with DSP/BIOS" on page 56-75

"Profiling Generated Code" on page 56-79

"Using DSP/BIOS with Your Target Application" on page 56-92

"Generating Code for Any C64x+ Processor or Board" on page 56-93

Introducing DSP/BIOS

Embedded Coder software supports DSP/BIOS features as options when you generate code for your target. In the following sections, read more about DSP/BIOS, how the Embedded Coder incorporates DSP/BIOS features into your generated code, and ways to use the real-time operating system (RTOS) features of DSP/BIOS in your applications. Follow these links for more information on specific areas that interest you, or read on for more details.

- "DSP/BIOS and Targeting Your C6000 DSP" on page 56-72
- "Code Generation with DSP/BIOS" on page 56-75
- "Profiling Generated Code" on page 56-79
- "Using DSP/BIOS with Your Target Application" on page 56-92

As a part of the Texas Instruments eXpressDSP™ technology, TI designed DSP/BIOS to include three components:

- DSP/BIOS Real-Time Analysis Tools — use these tools within Code Composer Studio IDE to view your program as it executes on the target in real time.
- DSP/BIOS Configuration Tool — enables you to add and configure any DSP/BIOS objects that you use to instrument your application. Use this tool to configure interrupt schedules and handlers, set thread priorities, and configure the memory layout on your DSP.

- DSP/BIOS Application Program Interface (API) — use C or assembly language functions to access DSP/BIOS functions by calling over 150 API functions. Embedded Coder software uses the API to access DSP/BIOS.

You link these components into your application, directly or indirectly referencing only functions you need for your application to run efficiently and optimally. Only functions that you specifically reference become part of your code base. To avoid adding unused code to your project, the software excludes functions you do not reference. After you add DSP/BIOS functions, the configuration tool helps you disable features you do not need later, optimizing your program for speed and size.

For details about DSP/BIOS and what it can do for your applications, refer to your CCS IDE and DSP/BIOS documentation from Texas Instruments.

DSP/BIOS and Targeting Your C6000 DSP

- "Introduction" on page 56-72
- "DSP/BIOS Configuration File" on page 56-73
- "Memory Mapping" on page 56-74
- "Hardware Interrupt Vector Table" on page 56-74
- "Linker Command File" on page 56-74

Introduction

When you generate code from your DSP model, you can include DSP/BIOS features provided by Embedded Coder software.

Including DSP/BIOS in your generated project adds the following files to your project:

- `modelname.tcf` — a DSP/BIOS configuration file
- `modelnamecfg.s62` — contains the DSP/BIOS objects required by your application and the vector table for the hardware interrupts.
- `modelnamecfg.h62` — the header file for `modelnamecfg.s62`.
- `modelnamecfg.h` — model configuration header file.

- `modelnamecfg_c.c` — source code for the model.
- `modelnamecfg.cmd` — the linker command file for the project. Adds the required DSP/BIOS libraries and the library `RTS6201.lib`, or the run-time support library for your target.

The executable code and source code you generate when you use the DSP/BIOS option are not the same as the code generated without DSP/BIOS included.

Instead of incorporating the DSP/BIOS files manually, as you would with CCS IDE, the Embedded Coder software starts from your Simulink model and adds the DSP/BIOS files automatically. As it adds the files, the support package:

- Configures the DSP/BIOS configuration file for your model needs
- Sets up the objects you use to analyze your program while it runs on your target
- Handles memory mapping to optimize your code based on the blocks in your model

DSP/BIOS Configuration File

DSP/BIOS projects all have a file with the extension `.tcf`. The file contains the DSP/BIOS configuration information for your project, in the form of objects for instrumenting and scheduling tasks in the program code. A DSP/BIOS project can include the following files:

- Log (LOG) objects for logging events and messages (replace the `*printf` statements, for instance)
- Statistics (STS) objects for tracking the performance of your code
- A clock (CLK) object for configuring the clock on your target, and various memory functions
- Hardware and software interrupt (HWI, SWI) objects that control program execution
- Other objects you use to meet your needs

Your TI DSP/BIOS documentation can provide all the details about the objects and how to use them. In addition, your installed software from TI includes tutorials to introduce you to using DSP/BIOS in projects.

Not all of the DSP/BIOS objects get used by the code you generate from Embedded Coder software. In the next sections, you learn about which objects the targeting software uses and how. You can still add more objects to your code through CCS IDE.

If you add DSP/BIOS objects beyond those provided by the Embedded Coder, you lose your additions when you regenerate code from your Simulink model.

Memory Mapping

Memory mapping that takes place in the linker command file now appears in the `MEM` object in the DSP/BIOS configuration file. Your memory sections, such as the `DATA_MEM` assignments and definitions, move to the MEM object, as do the memory segments. After completing this conversion, the memory assignment portions of your non-DSP/BIOS linker command file are not necessary in the linker command file.

Hardware Interrupt Vector Table

In non-DSP/BIOS project, the assembly language file `vector.asm` in your project defines the hardware interrupt vector table. This file defines which interrupts your project uses and what each one does.

When you use DSP/BIOS, the interrupts defined in the vector table move to the Hardware Interrupt Service Routine Manager in the CCS Configuration Tool. With your interrupts defined as Hardware Interrupts (HWI) in the Configuration Tool, your project does not need `vector.asm`, so the file does not appear in your DSP/BIOS enabled projects.

Linker Command File

After migrating your memory sections, segment, and hardware interrupt vector table to the configuration file, building with the DSP/BIOS option creates a compound linker command file. Because DSP/BIOS allows only one command file per project, and your linker file may comprise command options that did not relocate the DSP/BIOS configuration, Embedded Coder software uses *compound* command files. Compound command files work to let your project use more than one command file.

By starting your original linker command file with the statement

```
"-lmodelnamecfg.cmd"
```

added as the first line in the file, your DSP/BIOS enabled project uses both your original linker command file and the DSP/BIOS command file. You get the features provide by DSP/BIOS as well as the custom command directives you need.

Code Generation with DSP/BIOS

- "Overview" on page 56-75
- "Generated Code Without and With DSP/BIOS" on page 56-75

Overview

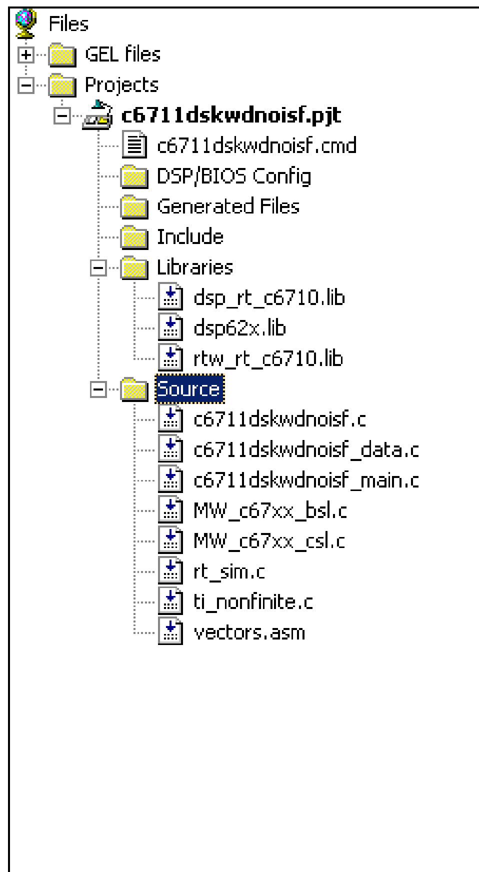
While generating code that includes the DSP/BIOS options is straightforward, changes occur between code that does not include DSP/BIOS and code that does. Two things change when you generate code with DSP/BIOS—files are added and removed from the project in CCS IDE, and DSP/BIOS objects become part of your generated code. With these in place, you can use the DSP/BIOS features in CCS IDE to debug your project, as well as use the profiling option in Embedded Coder software to check the performance of your application running on your target.

To generate code that includes DSP/BIOS options, open the Target Preferences block and select **DSP/BIOS** from the **Operating system** list on the **Board** pane.

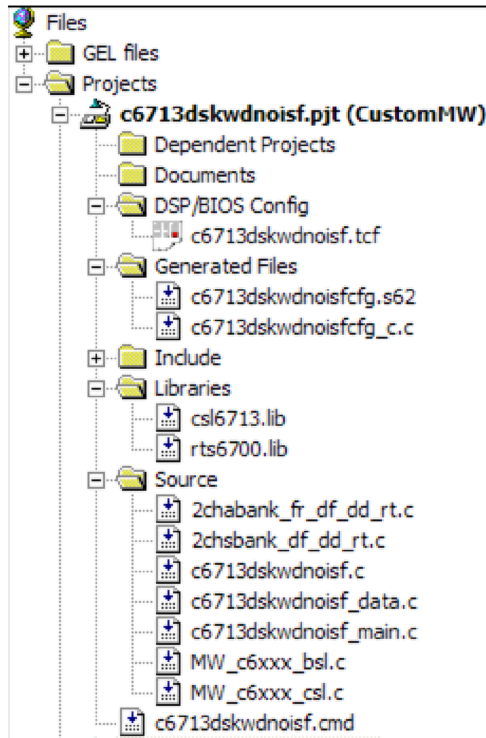
Generated Code Without and With DSP/BIOS

The next two figures show the results of generating code without and with the DSP/BIOS option enabled in the **Simulation Parameters** dialog.

Example — c6713dskwdnoisf.pjt code Generated Without DSP/BIOS. When you create your project in CCS IDE, the folder structure looks like this.



Example — c6713dskwdnoisf.pjt Code Including DSP/BIOS. If you now create a project that includes DSP/BIOS, the folder structure for your project changes to look like the following figure.



Added File	Description
`modelname.tcf`	Contains the DSP/BIOS objects required by your application, and the vector table for the hardware interrupts
`modelnamecfg.s62`	Shows all the included files in your project, the variables, the DSP/BIOS objects, and more in this file generated from the `.tcf` file
`modelnamecfg.h62`	The header file for `modelnamecfg.s62`
`modelnamecfg.h`	Model configuration header file

Added File	Description
modelnamecfg_c.c	Source code for the model
modelnamecfg.cmd	The linker command file for the project. Adds the required DSP/BIOS libraries and the library RTS6201.lib or the run-time support library for your target.

Notice that the new folder includes some new files, shown in the next table.

With DSP/BIOS functions enabled for your project, the following files no longer appear in your project.

Filename	Description
vectors.asm	Defines the hardware interrupts (HWI) used by interrupt service routines on the processor. This file is removed after all of the hardware interrupts appear in the **HWI** section of the Configuration Tool.
Original linker command file—modelname.cmd	Assigns memory sections on the processor. This file is removed if the **SECTION** directive is empty because all of the section assignments moved to the configuration file. Otherwise, include call to the DSP/BIOS command file.
Some *.lib files	Provide access to libraries for the processor, and peripherals. These files are removed if their contents have been incorporated in the new compound linker command file.

When you investigate your generated code, notice that the function main portion of modelname_main.c includes different code when you generate DSP/BIOS-enabled source code, and modelname_main.c incorporates one or more new functions.

Profiling Generated Code

- "Overview" on page 56-79
- "Profiling Subsystems" on page 56-80
- "Details About Timing and Profiling" on page 56-81
- "Profiling Multitasking Systems" on page 56-82
- "The Profiling Report" on page 56-83
- "Interrupts and Profiling" on page 56-85
- "Reading Your Profile Report" on page 56-86
- "Definitions of Report Entries" on page 56-87
- "Profiling Your Generated Code" on page 56-89
- "To Enable Profiling for Your Generated Code" on page 56-89
- "To Create Atomic Subsystems for Profiling" on page 56-90

Overview

When you use Embedded Coder software to generate code that incorporates the DSP/BIOS options, you can easily profile your generated code to gauge performance and find bottlenecks.

By selecting **Profile real-time execution** in the Simulink Coder software options, Simulink Coder software inserts statistics (STS) object instrumentation at the beginning and end of the code for each atomic subsystem in your model. (For more about STS objects, refer to your DSP/BIOS documentation from Texas Instruments.)

After your code has been running for a few seconds on your target, you can retrieve the profiling results from your target and display the information in a custom HTML report.

Code profiling works only on atomic subsystems in your model. To allow the Embedded Coder to profile your model when you build it in Simulink Coder software, you convert segments of your model into atomic subsystems using **Create subsystem**.

By designating subsystems of your model as atomic, you force each subsystem to execute only when all of its inputs are available. Waiting for all the subsystem inputs to be available before running the subsystem allows the subsystem code to be profiled as a contiguous segment.

To enable the profile feature for your Simulink model, select **Simulation** > **Configuration Parameters** from the model menu bar. In the Configuration Parameters dialog box, select **Code Generation** > **IDE Link**. Under **Code Generation**, enable **Profile real-time execution**.

Profiling Subsystems

Nested subsystems are profiled as part of their parent systems—the execution time reported for the parent subsystem includes the time spent in any profiled child subsystems. You cannot profile child subsystems separately.

For models that include multiple sample times, one or more subsystems in your model might not be included in the profiling process. When your model is configured to use single-tasking mode, all atomic subsystems in your model are profiled and appear in the report. When your model uses multitasking (refer to your Simulink Coder documentation for more about multitasking models) profiling applies only to single-rate subsystems that execute at the base rate of your model. This limitation arises because all of the generated code segments must execute contiguously for the profiling timing measurements to be correct. Setting the **Tasking mode for periodic sample times** to `Auto` in the model configuration parameters does not guarantee contiguous execution for all code segments and subsystems.

Notice two things in your code:

- STS objects are added to the generated code
- A generated DSP/BIOS configuration gets added to the project configuration file

Embedded Coder software inserts and configures these objects specifically for profiling your code. You do not have to make changes to the STS objects. To see the statistics objects in use, download your generated application to your board, select **DSP/BIOS > Statistics View** from the menu bar in CCS IDE, and run the board for a few seconds. You see the statistics being accumulated by the STS objects.

Details About Timing and Profiling

The profiling system in Embedded Coder software relies on DSP/BIOS STS objects and the `CLK_gethtime()` function. `CLK_gethtime()` returns a high resolution timing counter that enables profiling to measure the instruction cycles the CPU spends executing code segments. To understand profiling, you need to understand how `CLK_gethtime()` works.

This is how the system determines the value of `CLK_gethtime`:

```
CLK_gethtime() return val = CLK_getltime() *PRDO + CNTO
```

`PRDO` and `CNTO` are timer 0 period and counter registers. In code generation, BIOS allocates timer 0 as a system timer and set the timer to generate a timer interrupt every 1ms. `CLK_getltime()` in turn returns the number of BIOS system timer interrupts. By this logic, `PRDO` is set to the number of CPU clock cycles divided by the number of low resolution clock cycles that is equivalent to 1 millisecond in absolute time (8 low resolution clock cycles for C64x processors, for example).

The key point here is that function `CLK_gethtime()` relies on the `CLK_getltime()` function which in turn relies on a timer 0 interrupt. If your process globally disables interrupts during code execution for more than 1 `PRDO` instruction cycle, one or more timer interrupts can be missed, resulting in a situation where both `CLK_getltime()` and `CLK_gethtime()` can be inaccurate.

`CLK_getltime()` will be inaccurate because it does not report the correct time value. But it is always positive. The situation is worse for `CLK_gethtime()` It may report negative timing around code segments where interrupts are disabled:

```
A = CLK_gethtime();
IRQ_globalDisable();
{
    Code segment;
}
IRQ_globalEnable();
B = CLK_gethtime();
```

In this situation, if interrupts are disabled longer than 1ms around the code segment to be profiled, `B` might be smaller than `A` since `CTNO` might have

rolled over. So the count of the instruction cycles computed as (`B - A`) might be negative.

Correcting Inaccurate Profile Information Due to Timing. One way to correct problems in profiling caused by the disabled interrupts is to set the DSP/BIOS system timer interrupt to occur less frequently. As noted earlier, the timer is set to 1 millisecond by default.

You can change setting manually after you generate code for your project. Here are the steps to use to reset the DSP/BIOS system timer interval.

1 Open the `.tcf` file for the project.

2 Select **Scheduling > CLK Clock Manager**.

3 Right-click `CLK Clock Manager` to set the properties for the clock manager.

4 Change the **Microseconds/Int** value from the default `1000.00` microseconds to something larger, for example, `5000.00` microseconds.

5 Save the project.

This timing change reduces the chances of missing a system timer interrupt. If you do this and profile the code again, the profiling results are usually accurate. You can verify that if you reduce the system timer interrupt interval further, to perhaps 100 microseconds, you get less and less accurate profiling results, possibly reporting negative timing values.

Profiling Multitasking Systems

For a multitasking system, DSP/BIOS STS objects cannot reliably measure the time the processor spends in all tasks. When tasks can be preempted by other tasks (a result of multitasking operation), the profile timing measurements may be incorrect. For this reason, Embedded Coder software includes profiling instrumentation for atomic systems that run at the base sample rate only.

When you run the same model in single tasking mode, you can get the timing measurements for all the systems in your model for one iteration:

1. Select **Simulation** > **Configuration Parameters** from the model menu bar.

2. Under **Tasking** on the **Solver** pane, select `SingleTasking` for **Tasking mode for periodic sample times**.

3. Rebuild and execute your model on your C6000 hardware.

The program will probably overrun immediately since single tasking mode requires that all tasks complete within the base sample time which usually does not happen. However, all systems and subsystems do run once before the program terminates. This allows you to obtain profiling results for all systems.

When the overrun occurs, click **Halt** in CCS IDE to stop DSP/BIOS operation.

Then, enter `CCS_Obj.profile('report')` at the MATLAB prompt to report the statistics measurements.

Now you can view the timing measurements for each subsystem. Keep in mind that the percentages are given relative to the base sample time, so you must do some arithmetic to figure out whether a given system will fit in its available time interval. For instance, if your base sample time is 1 second, subsystem A executes every 3 seconds, the base-rate task takes 0.1 seconds to run, and A takes 2.5 seconds to run, the system should execute without overruns in multitasking mode.

If you change the overrun action option from its default setting of `Notify and halt` to `Notify and continue` or `None`, you can get measurements for multiple iterations of the system. Also, you will be able to request the profile report without first halting the CPU.

The Profiling Report

To help you measure subsystem performance, Embedded Coder software provides a custom report that analyzes and displays the profile statistics. The report shows you the amount of time spent computing each subsystem, including `Outputs` and `Update` code segments, and provides links that open the corresponding subsystem in the Simulink model.

To view the profiling report, enter

```
profile(IDE_Obj,'report')
```

at the MATLAB prompt, where `IDE_Obj` is the handle to your target and CCS IDE, and `report` is one of the input arguments for `profile`.

When you generate the report, the Embedded Coder stores the report in your code generation working folder, something like *`modelname`*`.c6000.rtw`, with the name `profileReport.html`.

If the MATLAB® software cannot find your code generation folder, the profile reports is stored in your temporary folder, `tempdir`. To locate your temporary folder, enter

```
tempdir
```

at the MATLAB command prompt.

Caution Each time you run the profiling process, the software replaces your existing report with a newer version. To save earlier reports, rename and save the report before you generate a new one, or change your destination temporary folder in the MATLAB workspace.

You must invoke `profile` after your Simulink Coder build, without clearing MATLAB memory between operations, so that stored information about the model is still available to the report generator. If you clear your MATLAB memory, information required for the profile report gets deleted and the report does not work properly. When this occurs, and if you have a CCS IDE project that was previously created with Simulink Coder software, you must repeat the Simulink Coder build to see the subsystem-based profile analysis in the report.

Trace each subsystem presented in the profile report back to its corresponding subsystem in your Simulink model by clicking a link in the report. (The mapping from Simulink subsystems to generated system code is complex and thus not detailed here.) Inspect your generated code, particularly `modelname.c`, to determine where and how Simulink and Simulink Coder software implemented particular subsystems.

Within the generated code, you see entries like the following that define STS objects used for profiling.

```
STS_set(&stsSys0_Output, CLK_gethtime());
```

or

```
STS_delta(&stsSys0_Output, CLK_gethtime());
```

This pair of code examples perform the profiling of the code section that lies between them in `modelname.c`.

In CCS IDE, STS objects show up in the Statistics Object Manager section under **Instrumentation** in the `modelname.tcf` file. Double-click the file `modelname.tcf` in the CCS IDE tree view to open the file and see the sections.

In some cases, Simulink Coder software may have pruned unused data paths, causing related performance measurements to become meaningless. Reusable system code, or code reuse, where a single function is called from multiple places in the generated code, can exhibit extra measurements in the profile statistics, while the duplicate subsystem may not show valid measurements.

Interrupts and Profiling

Although there are STS objects that measure the execution time of the entire `mdlOutputs` and `mdlUpdate` functions, those measurements can be misleading because they do not include other segments of code that execute at each interrupt. Statistics for the SWI are used when calculating the headroom (the difference between the number of CPU cycles your process requires to complete and the number available for the process to complete, which does not include the small overhead required for each interrupt. Note that profiling of multitasking systems does not measure the headroom. In addition, multitasking profiling does not use the SWI statistics.

To measure most accurately the overall application CPU usage, consider the DSP/BIOS IDL statistics, which measure time spent *not* doing application work. Your DSP/BIOS documentation from TI provides details about the various DSP/BIOS objects in the `tcf` file.

The interrupt rate for a DSP/BIOS application created by Embedded Coder software is the fastest block execution rate in the model. The interrupt rate

is usually, but not always, the same as the codec frame rate. When there is an upsampling operation or other rate increasing operation in your model, interrupts are triggered by a timer (PRD) object at the faster rate. You can determine the effective interrupt rate of the model by inverting the interrupt interval reported by the profiler.

Profiling subsystems that contain “blocking” device drivers, such as the ADC/DAC blocks and C6000 UDP Receive blocks may produce inaccurate and misleading results, raising values for **Max time spent in this subsystem per interrupt** and **Max percent of base interval** by many orders of magnitude. To avoid this problem, design subsystems to isolate blocking device drivers from algorithmic and other processing functions, and configure profiling appropriately.

Reading Your Profile Report

After you have the report from your generated code, you need to interpret the results. This section provides a link to sample report from a model and explains each entry in the report.

Sample of a Profile Report. When you click Sample Profile Report, the sample report opens in a new Help browser window. This opens the sample report in a new window so you can read the report and the descriptions of the report contents at the same time. Running the model `c6713dskwdnoisf` with DSP/BIOS generates the sample profile report. The next sections explain the headings in the report—what they mean and how they are measured (where that applies).

Report Heading Information. At the beginning of the report, profiling provides the name of the model you profiled, the target you used, and the date of the report. Since the report changes each time you run it, the date can be an important means of tracking model development.

Report Subsections and Contents. Within the body of your profile report, sections report the overall performance of your generated code and the performance of each atomic subsystem.

Report Heading	Description
Timing Constants	Shows you the base sample time in your model (=1/base rate in Hz) and the CPU clock speed used for the analysis.
Profiled Simulink Subsystems	Presents the statistics for each profiled subsystem separately, by subsystem. Each listing includes the STS object name or names that instrument the subsystem.
STS Objects	Lists every STS object in the generated code and the statistics for each. DSP/BIOS uses these objects to determine the CPU load statistics. For more information about STS objects, refer to your DSP/BIOS documentation from TI.

STS objects that are associated with subsystem profiling are configured for host operation at 4*x, reflecting the numerical relationship between CPU clock cycles and high-resolution timer clicks, x. STS Average, Max, and Total measurements return their results in counts of instructions or CPU clock cycles.

Definitions of Report Entries

In the following sections, we provide definitions of the entries in the profile report. These definitions help you decipher the report and better understand how your process is performing.

System name. Provides the name of the profiled model, using the form `targetname`profile. *targetname* is the processor or board assigned as the target, via the Target Preferences block.

Number of Iterations Counted. The number of interrupts that occurred between the start of model execution and the moment the statistics were obtained.

CPU Clock Speed. The instruction cycle speed of your digital signal processor. On the C6713 DSK, you can adjust this speed to one of four values, where 100 MHz is the default—25, 33.25, 100, 133 MHz. If you change the speed to something other than the default setting of 100 MHz, you must specify the new speed in the Simulink Coder software options. Use the **Current C6713DSK CPU clock rate** option on the `TIC6000 runtime` category on the Simulink Coder tab.

Set at a fixed 150 MHz, you cannot change the CPU clock rate on the C6713 DSK. You do not need to report the setting in the Simulink Coder software options.

Maximum Time Spent in This Subsystem per Interrupt. The amount of time spent in the code segment corresponding to the indicated subsystem in the worst case. Over all the iterations measured, the maximum time that occurs is reported here. Since the profiler only supports single-tasking solver mode, no calculation can be preempted by a new interrupt. All calculations for all subsystems must complete within one interrupt cycle, even for subsystems that execute less often than the fastest rate.

Maximum Percent of Base Interval. The worst-case execution time of the indicated subsystem, reported as a percentage of the time between interrupts.

STS Objects. Profiling uses STS objects to measure the execution time of each atomic subsystem. STS objects are a feature of the DSP/BIOS run-time analysis tools, and one STS object can be used to profile exactly one segment of code. Depending on how Simulink Coder software generates code for each subsystem, there may be one or two segments of code for the subsystem; the computation of outputs and the updating of states can be combined or separate. Each subsystem is assigned a unique index, `i`. The name of each STS object helps you determine the correspondence between subsystems and STS objects. Each STS object has a name of the form

```
stsSysi_segment
```

where `i` is the subsystem index and `segment` is `Output`, `Update`, or `OutputUpdate`. For example, in the sample profile report shown in the next section, the STS objects have the names `stsSys1_OutputUpdate`, and `stsSys2_OutputUpdate`.

Profiling Your Generated Code

Before profiling your generated code, you must configure your model and Simulink Coder software to support the profiling features in Embedded Coder software. Your model must use DSP/BIOS features for profiling to work fully.

The following tasks compose the process of profiling the code you generate.

1. Enable DSP/BIOS for your code.
2. Enable profiling in the Simulink Coder software.
3. Create atomic subsystems to profile in your model.
4. Build, download, and run your model.
5. Use `profile` to view the MATLAB profile report.

To demonstrate profiling generated code, this procedure uses the wavelet denoising model `c6713dskwdnoisf.mdl` that is included with Embedded Coder demo programs. If you are using the C6713 DSK as your target, use the model `C6713dskwdnoisf` throughout this procedure. Simulators work as well, just choose the appropriate model for your simulator.

Begin by loading the model, entering

```
c6713dskwdnoisf
```

at the MATLAB prompt. The model opens on your desktop.

To Enable Profiling for Your Generated Code

Recall that you must use DSP/BIOS in your code to use profiling.

To enable the profile feature for your Simulink model, select **Simulation** > **Configuration Parameters** from the model menu bar. In the Configuration Parameters dialog box, select **Code Generation** > **IDE Link**. Under **Code Generation**, enable **Profile real-time execution**.

To Create Atomic Subsystems for Profiling

Profiling your generated code depends on two features—DSP/BIOS being enabled and your model having one or more subsystems defined as atomic subsystems. To learn more about subsystems and atomic subsystems, refer to your Simulink documentation in the Help browser.

In this tutorial, you create two atomic subsystems—one from the Analysis Filter Bank block and a second from the Soft Threshold block:

1. Select the Analysis Filter Bank block. Select **Edit > Create subsystem** from the model menu bar. The name of the block changes to subsystem. Repeat for the Soft Threshold block.

2. To convert your new subsystems to atomic subsystems, right-click on each subsystem and choose **Subsystem parameters...** from the context menu.

3. In the **Block Parameters: Subsystem** dialog for each subsystem, select the **Treat as atomic unit** option. Click **OK** to close the dialog. If you look closely you can see that the subsystems now have heavier borders to distinguish them from the other blocks in your model.

To Build and Profile Your Generated Code. You have enabled profiling in your model and configured two atomic subsystems in the model as well. Now, use the profiling feature to see how your code runs and check the performance for bottlenecks and slowdowns as the code runs on your target.

Caution Do not click on any other open model while you are profiling your model. Clicking on another open model can cause profiling to fail with an error message like "Invalid Simulink object specifier."

---

1. Select **Tools > Code Generation > Build Model**.

   If you did not use the Simulink Coder software options to automate model compiling, linking, downloading, and executing, perform these tasks using the **Project** options in CCS IDE.

   Allow the application to run for a few seconds or as long as necessary to execute the model segments of interest a few times. Then stop the program.

2. Create a link to CCS IDE by entering

```
IDE_Obj = ticcs;
```

   at the MATLAB prompt.

3. Enter

```
profile(IDE_Obj,'report')
```

   at the prompt to generate the profile report of your code executing on your target.

The profile report appears in the Help browser. It should look very much like the following sample report; your results may differ based on your target and the settings in the model.

Profile Report

Simulink model: c6416dskprofile.mdl
Target: C6416DSK

Report of profile data from Code Composer Studio (tm)
XX-XXX-2005 17:27:27

Timing constants

Base sample time	250 ms
CPU Clock speed[1]	720 MHz

Profiled Simulink Subsystems

System name	c6416dskprofile
STS object	stsSys2_OutputUpdate
Max time spent in this subsystem per interrupt	14.93 µs
Max percent of base interval	0.00597%
Number of iterations counted	144

System name	c6416dskprofile/Subsystem1
STS object	stsSys1_OutputUpdate
Max time spent in this subsystem per interrupt	12.8 µs
Max percent of base interval	0.00512%
Number of iterations counted	144

Using DSP/BIOS with Your Target Application

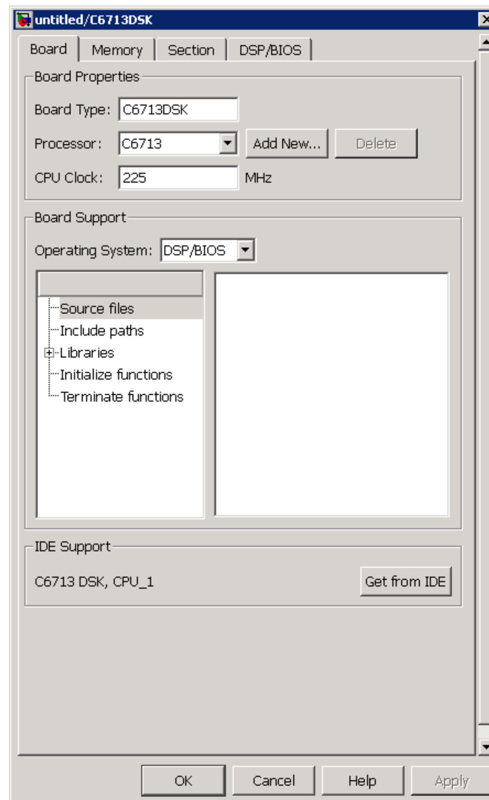
Enabling DSP/BIOS When You Generate Code

For any code you generate using Simulink Coder software and Embedded Coder software, you have the option of including DSP/BIOS features automatically when you generate the code. Incorporating the features requires you to select one option in the Target Preferences block—`DSP/BIOS` for the operating system.

1. Open the model to use to generate code.

2. Open the Target Preferences block in your model.

3 On the **Board** pane, select `DSP/BIOS` for **Operating system** under the Code Generation options.



4 As shown in the figure, select `DSP/BIOS` for **Operating system**.

Generating Code for Any C64x+ Processor or Board

The Target Preferences block imports hardware information directly from DSP/BIOS. This feature enables you to create custom Target Preferences blocks for any C64x+ CPU core-based processor or board. You create and reuse these Target Preferences blocks in your models to generate code for your C64x+ processor or board.

To create a custom Target Preferences block for your C64x+ processor or board:

1. In MATLAB, enter `idelinklib_common`.

2. Drag the Target Preferences block to your Simulink model.

3. Double-click the Target Preferences block. This action opens the block.

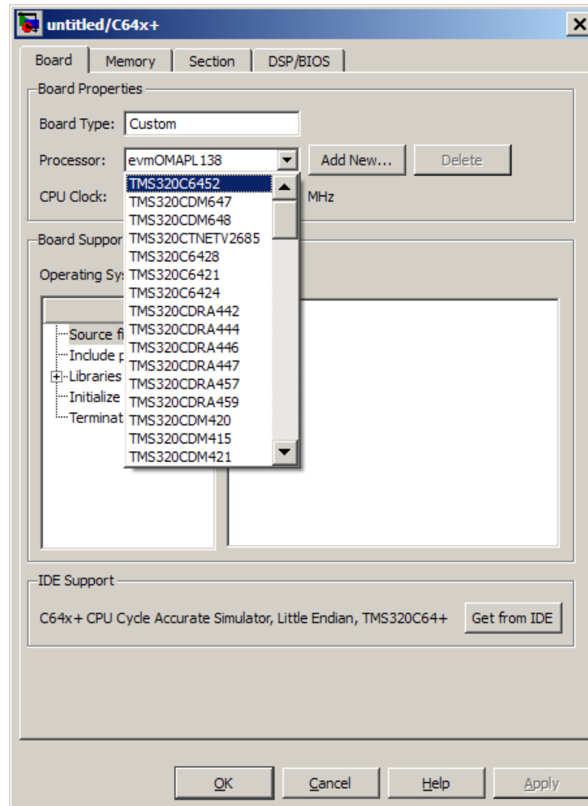
   When you open the Target Preferences block, the Embedded Coder software uses the BIOS_INSTALL_DIR environment variable to locate the DSP/BIOS installation folder. The link software then queries DSP/BIOS for a list of processors and boards with C64x+ cores and displays them in the **Processor** list.

   Note If you change DSP/BIOS versions in the CCS Component Manager, reopen the block to display the updated **Processor** list.

   ---

4. Select your processor from the **Processor** list.

   The Target Preferences block imports settings from DSP/BIOS such as the memory map, cache settings, and CPU clock rate. The block applies the settings to the **Memory**, **Section** and **DSP/BIOS** tabs.



5 Set the CPU Clock rate for your processor.

6 To improve the efficiency of your application, you can adjust the L1 and L2 caches values and the compiler sections. The following section provides an example of how to adjust these settings.

7 Click **OK**.

To create a library of custom Target Preferences blocks:

1 In Simulink, create a library: **File>New>Library**.

2 Copy any custom Target Preferences blocks from your models to the library.

3 Relabel individual blocks for the processors they specify: Click the block labels and edit the text.

4 Save the library to your default current folder. For example, save it as `c64xcustomtgtpreflib.mdl` in `c:\Program Files\matlab\bin`.

Later, you can reopen the library by entering the library name on the MATLAB command line.

Example: Creating a Custom Target Preferences Block for OMAP-L138/C6748 EVM

To create a custom Target Preferences Block for OMAP-L138/C6748 EVM:

1 On the MATLAB command line, enter `idelinklib_common`.

2 Copy the Target Preferences block to your model.

3 Open the Target Preferences block.

4 For **Processor**, select `evmOMAPL138`. The block populates the **Memory**, **Section** and **DSP/BIOS** tabs with default values for that processor.

5 Adjust the cache settings for efficiency. Turn L1 and L2 cache on:

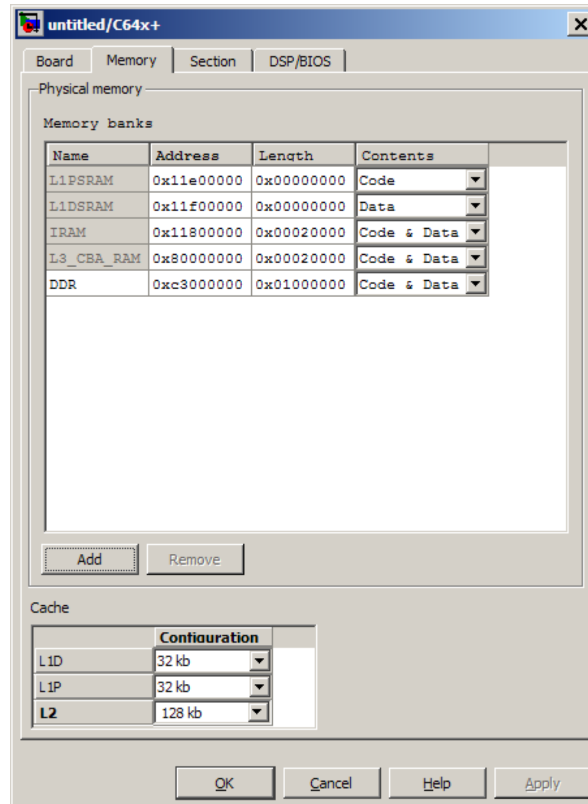
   a Click the **Memory** tab. The L1P, L1D, and L2 cache sizes are zero by default.

   b Set **L1D** and **L1P** cache sizes to `32 kb`.

   c When you increase the cache size, decrease the **L1PSRAM** and **L1DSRAM** to accommodate memory taken from the high address range of the corresponding L1 memory segments. Because L1PSRAM and L1DSRAM are 32 kb in the OMAP-L138/C6748 processor, change the **Length** for **L1PSRAM** and **L1DSRAM** to `0x00000000`. This setting allows you to use the entire L1 memory as level one cache.

   d Set **L2** cache size to 128 kb. Decrease the **IRAM** length to `0x00020000` (128 kb).

   e Confirm the following configuration for the **Memory** tab, and click **Apply**.



6 Reassign compiler sections to optimize the efficiency of the generated code. Put `.stack`, `.bios`, `.hwi`, `.hwi_vec` sections into fast internal memory and assign everything else to external memory. This approach avoids linking errors caused by placing excessive code and data in limited internal memory. It also runs critical sections of the application from internal memory.

a Click the **Section** tab.

b In the **Compiler sections** list, set the **Placement** of every section, except `.stack`, `.bios`, `.hwi`, and `.hwi_vec`, to `DDR`.

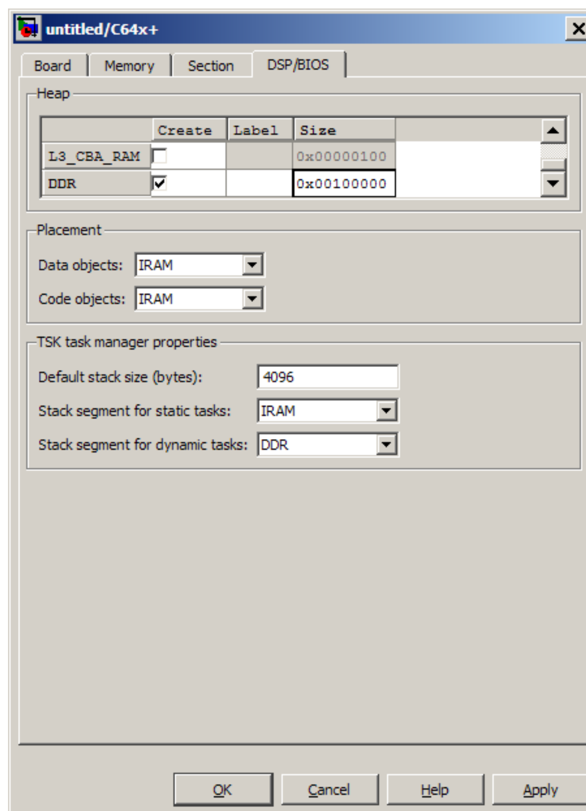
c Set the **Placement** of `.stack`, `.bios`, `.hwi`, and `.hwi_vec` to `IRAM`.

7 The final step in configuring the OMAP-L138/C6748 EVM Target Preferences block is to create a heap in external memory. Device drivers from TI use heap to allocate data structures and device driver buffers. Without a heap, integrating device drivers from TI would not be possible.

a Click the **DSP/BIOS** tab.

b In the **Heap** list, select **DDR** and set the heap size to `0x00100000` (1 MB).

c Confirm the following configuration for the **DSP/BIOS** tab, and click **OK**.



Using the C62x and C64x DSP Libraries

In this section...
“About the C62x and C64x DSP Libraries” on page 56-99
“Fixed-Point Numbers” on page 56-101
“Building Models” on page 56-106

About the C62x and C64x DSP Libraries

- “C62x DSP Library” on page 56-99
- “C64x DSP Library” on page 56-100
- “Supported Platforms” on page 56-101
- “Characteristics Common to C62x and C64x Library Blocks” on page 56-101

C62x DSP Library

Blocks in the C62x DSP library correspond to functions in the Texas Instruments TMS320C62x DSP Library assembly-code library, which target the TI C62x family of digital signal processors. Use these blocks to run simulations by building models in Simulink software before generating code. Once you develop your model, you can invoke Simulink Coder software to generate code that is optimized to run on C6713 DSK development platforms or C62x hardware. (Fixed-point processing on C67x hardware is identical to C62x fixed point hardware and processing so you can develop on the C67x for the C62x.) During code generation, each C62x DSP Library block in your model is mapped to its corresponding TMS320C62x DSP Library assembly-code routine to create target-optimized code.

C62x DSP Library blocks generally input and output fixed-point data types. The block reference topics discuss the data types accepted and produced by each block. “Fixed-Point Numbers” on page 56-101 gives a brief overview of using fixed-point data types in Simulink software. For an in-depth discussion of fixed-point data types, including issues with scaling and precision when you perform fixed-point operations, refer to your “Fixed-Point Toolbox” documentation.

You can use C62x DSP Library blocks with certain blocks from the DSP System Toolbox software and Simulink software. To learn more about creating models that include both C62x DSP Library blocks and blocks from other blocksets, refer to "Building Models" on page 56-106.

C64x DSP Library

Blocks in the C64x DSP library correspond to functions in the Texas Instruments TMS320C64x DSP library assembly-code library, which target the TI C64x family of digital signal processors. Use these blocks to run simulations by building models in Simulink software before generating code. Once you develop your model, you can invoke Simulink Coder software to generate code that is optimized to run on the C6416 DSK development platform or other C64x hardware. During code generation, each C64x DSP Library block in your model is mapped to its corresponding TMS320C64x DSP Library assembly-code routine to create target-optimized code.

C64x DSP Library blocks generally input and output fixed-point data types. "Optimization — C64x DSP Library (tic64dsplib)" discusses the data types accepted and produced by each block in the library. "Fixed-Point Numbers" on page 56-101 gives a brief overview of using fixed-point data types in Simulink software. For an in-depth discussion of fixed-point data types, including issues with scaling and precision when you perform fixed-point operations, refer to your Fixed-Point Toolbox™ documentation.

You can use C64x DSP Library blocks with certain blocks from the DSP System Toolbox software and Simulink software. To learn more about creating models that include both C64x DSP Library blocks and blocks from other blocksets, refer to "Building Models" on page 56-106.

Note While you can use C62x blocks on C64x targets, the generated code is not optimal for the C64x target. Using the appropriate C64x block creates better optimized code. (Embedded Coder software generates a warning message when you try to do this but allows you to use the block.)

Supported Platforms

The C62x and C64x DSP libraries can be used with the platforms listed in the following table:

Library	Supported platforms
C62x	C62x, C67x, C67x+, C64x, C64x+
C64x	C64x, C64x+

Characteristics Common to C62x and C64x Library Blocks

The following characteristics are common to all C62x and C64x DSP Library blocks:

- All blocks inherit sample times from driving blocks.
- The blocks are single rate.
- Block filter weights and coefficients are tunable, but not in real time. Other block parameters are not tunable.
- All blocks support discrete sample times. Individual block reference pages indicate blocks that also support continuous sample times.

To learn more about characteristics particular to each block in the library, refer to "Optimization — C62x DSP Library (tic62dsplib)" and "Optimization — C64x DSP Library (tic64dsplib)"

Fixed-Point Numbers

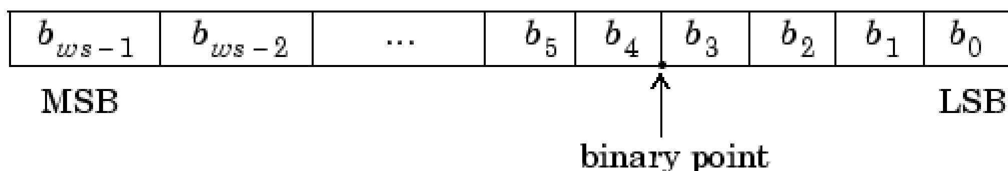
- "Notation" on page 56-101
- "Signed Fixed-Point Numbers" on page 56-102
- "Q Format Notation" on page 56-103

Notation

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a fractional fixed-point number (either signed or unsigned) is shown below.



where

- b_i is the ith binary digit.
- ws is the word size in bits.
- b_{ws-1} is the location of the most significant (highest) bit (MSB).
- b_0 is the location of the least significant (lowest) bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example the number is said to have four fractional bits, or a fraction length of four.

Note For Embedded Coder, the results of fixed-point and integer operations in MATLAB/Simulink match the results on the hardware target down to the least significant bit (bit-trueness). The results of floating-point operations in MATLAB/Simulink do not match those on the hardware target, because the libraries used by the third-party compiler may be different from those used by MATLAB/Simulink.

Signed Fixed-Point Numbers

Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude

- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and the one TI digital signal processors use.

Negation using signed two's complement representation consists of a bit inversion (translation into one's complement) followed by the binary addition of a one. For example, the two's complement of 000101 is 111011:

000101 ->111010 (bit inversion) ->111011 (binary addition of 1 to the LSB)

results in the negative of 000101 being 111011.

Q Format Notation

The position of the binary point in a fixed-point number determines how you interpret the scaling of the number. When performing arithmetic such as addition or subtraction, hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a binary point. They perform signed or unsigned integer arithmetic—as if the binary point is to the right of the LSB (b_0). Therefore, you determine the binary point in your code.

In the C62x DSP Library, the position of the binary point in signed, fixed-point data types is expressed in and designated by Q format notation. This fixed-point notation takes the form

$Qm.n$

where

- Q designates that the number is in Q format notation—the Texas Instruments notation for signed fixed-point numbers.
- m is the number of bits used to designate the two's complement integer portion of the number.
- n is the number of bits used to designate the two's complement fractional portion of the number, or the number of bits to the right of the binary point. Sometimes n is called the scale factor.

Q format always designates the most significant bit of a binary number as the sign bit. Representing a signed fixed-point data type in Q format requires m+n+1 bits to account for the sign.

Example — Q.15. For example, a signed 16-bit number with n = 15 bits to the right of the binary point is expressed as

```
Q0.15
```

in this notation. This is (1 sign bit) + (0 = m integer bits) + (15 = n fractional bits) = 16 bits total in the data type. In Q format notation the m = 0 is often implied, as in

```
Q.15
```

In Fixed-Point Toolbox software, this data type is expressed as

```
sfrac16
```

or

```
sfix16_En15
```

DSP System Toolbox software expresses this data type as the vector

```
[16 15]
```

meaning the word length is 16 bits and the fraction length is 15 bits.

Example — Q1.30. Multiplying two Q.15 numbers yields a product that is a signed 32-bit data type with 30 bits to the right of the binary point. One bit is the designated sign bit, forcing m to be 1:

m+n+1 = 1+30+1 = 32 bits total

Therefore this number is expressed as

```
Q1.30
```

In Fixed-Point Toolbox software, this data type is expressed as

```
sfix32_En30
```

In DSP System Toolbox software, this data type is expressed as

```
[32 30]
```

Example — Q-2.17. Consider a signed 16-bit number with a scaling of $2^{(-17)}$. This requires n = 17 bits to the right of the binary point, meaning the most significant bit is a *sign-extended* bit.

Sign extension adds bits to the high end (MSB end) of the word and fills the added bits with the value of the MSB. For example, consider a 4-bit two's complement number 1011. Extending the number to 7 bits with sign extension changes the number to 1111011—the value of the number remains the same.

One bit is the designated sign bit, forcing m to be -2.

m+n+1 = -2+17+1 = 16 bits total

Therefore this number is expressed as

```
Q-2.17
```

In Fixed-Point Toolbox software, this data type is expressed as

```
sfix16_En17
```

To express this data type in DSP System Toolbox software, use

```
[16 17]
```

Example — Q17.-2. Consider a signed 16-bit number with a scaling of 2^(2) or 4. The binary point is implied to be 2 bits to the right of the 16 bits, or that there are n = -2 bits to the right of the binary point. One bit must be the sign bit, forcing m to be 17.

m+n+1 = 17+(-2)+1 = 16

Therefore this number is expressed as

```
Q17.-2
```

In Fixed-Point Toolbox software, this data type is expressed as

```
sfix16_E2
```

In DSP System Toolbox software, this data type is expressed as

```
[16 -2]
```

Building Models

- "Overview" on page 56-106
- "Converting Data Types" on page 56-106
- "Using Sources and Sinks" on page 56-107
- "Choosing Blocks to Optimize Code" on page 56-107

Overview

You can use C62x or C64x DSP Library blocks in models along with certain core Simulink and DSP System Toolbox software. This section discusses issues you should consider when you build models with blocks from these libraries.

Converting Data Types

Any blocks you connect in a model have compatible input and output data types. In most cases, C62x or C64x DSP Library blocks handle only a limited number of specific data types. Refer to any block reference page in "Optimization — C62x DSP Library (tic62dsplib)" and "Optimization — C64x DSP Library (tic64dsplib)" for a discussion of the data types that each block accept sand produces.

When you connect C62x or C64x DSP Library blocks and Simulink blocks, you often need to set the data type and scaling in the block parameters of the Simulink block to match the data type of the C62x DSP Library block. Many Simulink blocks allow you to set their data type and scaling by inheriting from the driving block, or by back propagating from the next block. This can be a good way to set the data type of a Simulink block to match a connected C62x DSP Library block.

Some DSP System Toolbox software blocks and Simulink blocks also accept fixed-point data types. Make the appropriate settings in the block parameters when you connect them to a C62x DSP Library block.

To use DSP System Toolbox software or core Simulink blocks that do not handle fixed-point data types with C62x DSP Library blocks in your model, you must use an appropriate data type conversion block:

- To connect fixed-point and nonfixed-point blocks, use the Data Type Conversion block from the Simulink Data Type library.
- To provide an interface to nonfixed-point blocks, use the C62x Convert Floating-Point to Q.15 and C62x Convert Q.15 to Floating-Point blocks from the C62x DSP Library.
- To connect blocks of varying nonfixed-point data types in your model, use the Data Type Conversion block from the Signals and Systems Simulink library
- To connect blocks of varying fixed-point data types in your model, use the Data Type Conversion Inherited block from the Simulink Data Type library.

Refer to the reference pages for these blocks or invoke the Help system from their block dialogs for more information.

Using Sources and Sinks

The C62x DSP Library does not include source or sink blocks. Use source or sink blocks from the core Simulink library or DSP System Toolbox software in your models with C62x DSP Library blocks. See "Converting Data Types" on page 56-106 for more information on incorporating blocks from other libraries into your models.

Choosing Blocks to Optimize Code

In some cases, blocks that perform similar functions appear in more than one blockset. For example, the C62x DSP Library, the C64x DSP Library, and the DSP System Toolbox software all have Autocorrelation blocks. How do you choose which to include in your model? If you are building a model to run on the C6713 DSK or on C62x hardware, choosing the block from the C62x DSP Library always yields better optimized code. You can use a similar block from another library if it provides functionality that

the C62x DSP Library block does not support, but you generate less well optimized code.

In the same manner, if you are building a model to run on the C6416 DSK or on C64x hardware, choosing the block from the C64x DSP Library always yields better optimized code. You can use a similar block from another library if it provides functionality that the C64x DSP Library block does not support, but you generate less well optimized code.

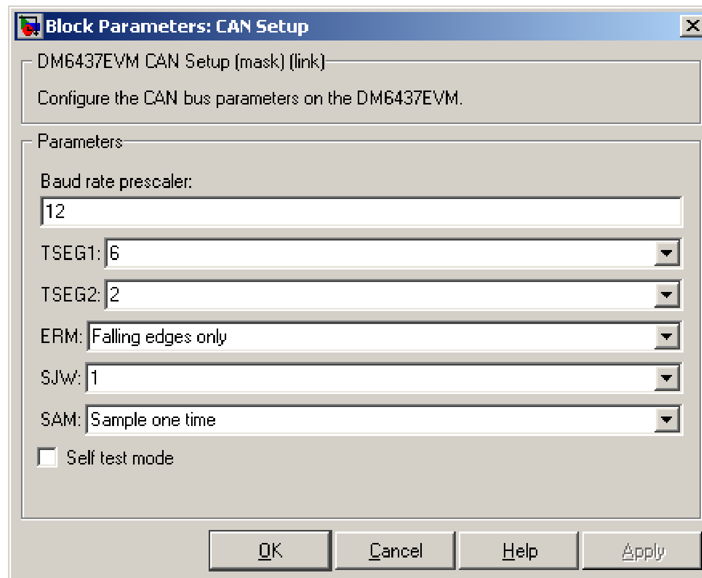
Configuring Timing Parameters for CAN Blocks

Setting Timing Parameters

- "Accessing the Timing Parameters" on page 56-109
- "Determining Timing Parameter Values" on page 56-110
- "CAN Bit Timing Example" on page 56-111

Accessing the Timing Parameters

The timing parameters that control the bit rate for DM643x CAN Receive and DM643x CAN Transmit blocks are **Baud rate prescaler**, **TSEG1**, and **TSEG2** in the DM643x CAN Setup block.



The following sections describe how to set these parameters.

Determining Timing Parameter Values

The following steps show you how to determine the appropriate values to use for the timing parameters.

1. Gather these two values:
   - Bit rate of the CAN network
   - SYSCLKOUT — This is equivalent to the CAN module system clock frequency. The CAN peripheral in the DM6437 is in the CLKIN clock domain, which operates at the same frequency as the primary reference clock to the DSP. In the DM6437EVM board, the primary reference clock operates at 27 MHz.

2. Estimate the value of the **Baud rate prescaler (BRP)** and then solve this equation for BitTime:

   BitTime = SYSCLKOUT/(BRP * Bit rate)

3. Estimate values for **TSEG1** and **TSEG2** that satisfy the following equation:

   BitTime = TSEG1 + TSEG2 + 1

   The estimated values must also satisfy the following constraints:

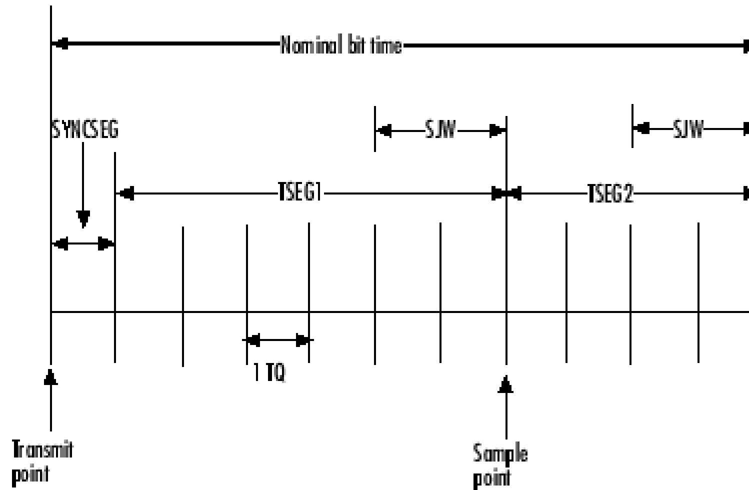
   TSEG1 >= **TSEG2**
   IPT (Information Processing Time) = 3/**BRP**
   IPT <= **TSEG1** <= 16 TQ
   IPT <= **TSEG2** <= 8 TQ
   1 TQ <= **SJW** <= min(4 TQ, **TSEG2**)

   where:

   IPT is Information Processing Time, TQ is Time Quanta, and **SJW** is Synchronization Jump Width, which can be set in the **CAN Setup** block.

4. Iterate steps two and three until the values selected for TSEG1, TSEG2, and BRP meet all of the criteria.

The following illustration shows the relationship between the parameters:



CAN Bit Timing Example

This example shows how to determine appropriate CAN timing parameters.

Assume that SYSCLKOUT = 27 MHz, and a Bit rate of 1 Mbits/s is required.

1. With the **Baud rate prescaler (BRP)** set to 12, substitute the values of Bit rate, *BRP*, and *SYSCLKOUT* into the following equation, solving for *BitTime*:

   BitTime = SYSCLKOUT/(BRP * Bit rate)

   BitTime = 27MHz/(12 * 0.25 MBits/sec) = 9TQ

2. Set the values of **TSEG1** and **TSEG2** to 6TQ and 2TQ, respectively. Substitute the values of *BitTime* from the previous equation, and the chosen values for *TSEG1* and **TSEG2** into the following equation:

   BitTime = TSEG1 + TSEG2 + 1

   9TQ = 6TQ + 2TQ + 1

3 Finally, check the selected values against the rules:

IPT = 3/**BRP** = 3/12 = .25
IPT <= **TSEG1** <= 16 TQ True! .25 <= 6TQ <= 16TQ
IPT <= **TSEG2** <= 8TQ True! .25 <= 2TQ <= 8TQ
1TQ <= **SJW** <= min(4TQ, **TSEG2**), as a result of which **SJW** can be set to either 1 or 2.

4 All chosen values satisfy the criteria, so no further iteration is necessary.

The following table provides common timing parameter settings for typical values of Bit rate and SYSCLKOUT = 27 MHz. This clock frequency is the maximum for the DM6437 EVM blocks.

Bit rate	TSEG1	TSEG2	Bit Time	BRP	SJW
250 Kbits/sec	6	2	3	12	1 or 2
500 Kbits/sec	3	1	6	9	1
1 Mbits/sec*	6	2	9	3	1 or 2
2 Mbits/sec*	1	1	4.5	3	ERROR

* 3-time sampling in the DM643x CAN module is not possible at this Bit rate. In the DM643x CAN Setup block, the **SAM** parameter cannot be set to `Sample three times`.

References. For detailed information on the CAN module, see *TMS320DM643x DMP High-End CAN Controller User's Guide (Rev. A)*, Literature Number SPRU981A, available at the Texas Instruments Web site.

See Also. DM643x CAN Setup, DM643x CAN Transmit

Hardware Issues

In this section...
"Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP" on page 56-113
"Requirements for the DM642 EVM" on page 56-113
"Installing and Configuring the Avnet Board Support Library" on page 56-116
"Continuing Issues with Embedded Coder Software" on page 56-118

Configuring the D.signT DSK-91C111 to Use TCP/IP and UDP

Specific evaluation boards that don't have a build-in Ethernet ports accept the D.signT DSK-91C111 daughter card with the required Texas Instruments TMS320C6000 TCP/IP Stack. To use the D.signT DSK-91C111, change the position of solder point jumper JPINTPOL. Set the jumper to the "b" position from the default "a" position. Refer to your TI TCP/IP Stack User's Guide documentation for additional information about configuring the daughter card.

Requirements for the DM642 EVM

- "Identifying Your DM642 EVM Board Version" on page 56-113
- "Installing Third-party Software" on page 56-114
- "Configuring the Target Preferences Block for Your DM642 EVM" on page 56-114
- "Configuring the DM642 EVM Video ADC Block" on page 56-115

This section provides details about using both the DM642 EVM hardware target and the simulator.

Identifying Your DM642 EVM Board Version

Spectrum Digital has released three versions of the DM642 EVM board:

- **Version 1** — Original board with 600 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 50 MHz oscillator.
- **Version 2** — Original board revised to use 720 MHz DM642, Philips SAA7115 video decoders. ASSY 506840 Rev. D on back of board, 60 MHz oscillator.
- **Version 3** — Revised board with 720 MHz DM642, TI TVP5146/5150 video decoders and HD filters. ASSY 507340 Rev. B on back of board, 60 MHz oscillator.

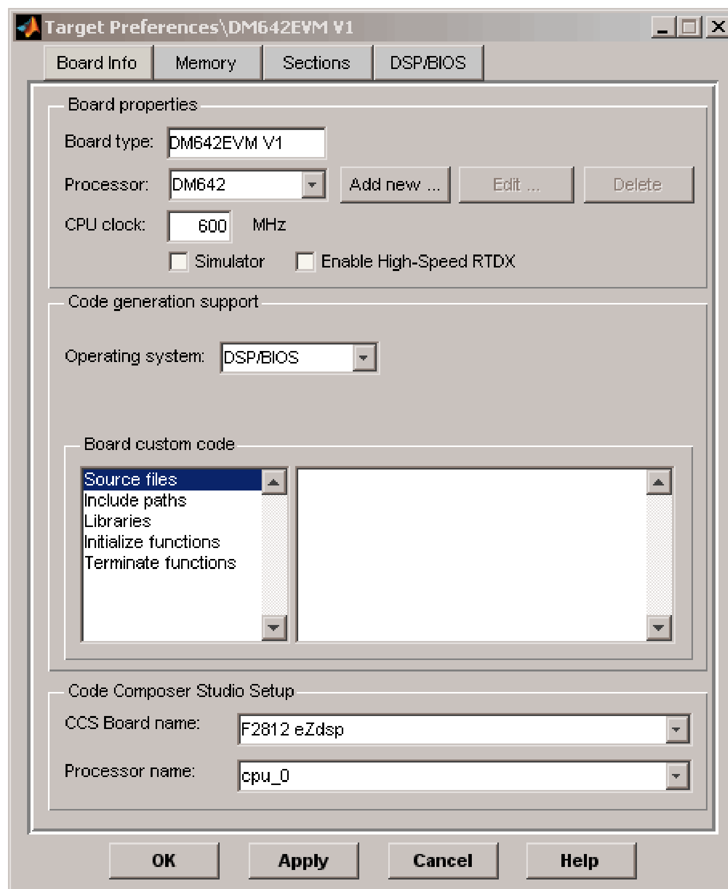
To determine the board version, consult the documentation provided with your board, or refer to the ASSY number located on the bottom surface of the board.

Installing Third-party Software

After determining the board version, install the *supported* versions of the third-party software for that board version. See the "System Requirements" on page 56-4 for the Embedded Coder software.

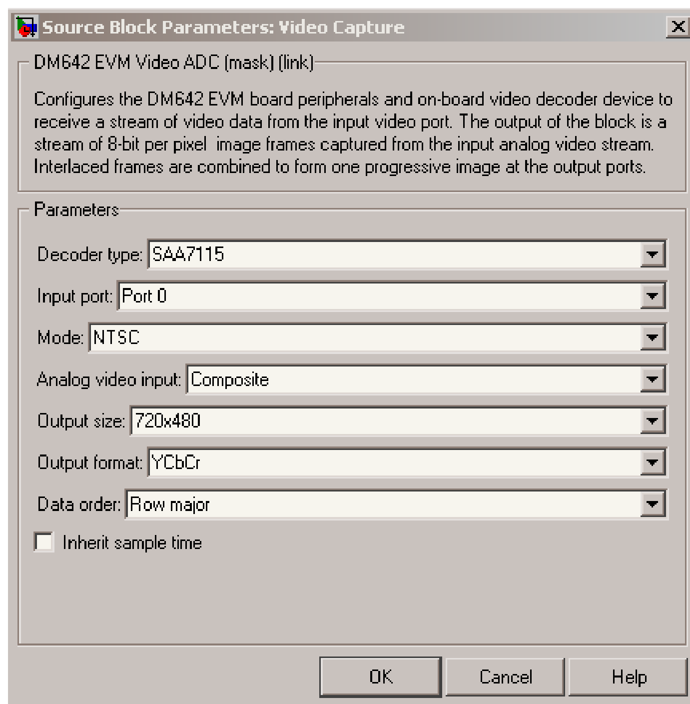
Configuring the Target Preferences Block for Your DM642 EVM

When you use the DM642EVM V1, V2, and V3 Target Preferences block, make sure that you enter the CPU clock speed that matches the CPU clock on your board. The figure below shows the correct setting of 600 for Version 1 boards in **CPU clock speed (MHz)**. For Version 2 and 3 boards, change the clock speed to 720.



Configuring the DM642 EVM Video ADC Block

If you have a DM642 EVM Version 2 or 3 board, make sure that you have the updated video drivers in your CCS IDE installation folder and that you select the correct decoder type `TVP5146` when you use DM642 EVM Video ADC blocks as shown in the following figure.



Installing and Configuring the Avnet Board Support Library

- "Preface" on page 56-116
- "Installing the Avnet Board Support Library" on page 56-117
- "Setting the MATLAB Environment" on page 56-117
- "For Spectrum Digital DM6437EVM Users" on page 56-118
- "Verifying Your Installation" on page 56-118

Preface

The Avnet S3ADSP DaVinci evaluation platform is designed for joint software and hardware design. It brings the Texas Instruments TMS320DM6437 DSP and Xilinx Sparta-3A FPGA together. This chapter provides an overview of

the board, and instructions for installing, configuring, and using the Avnet S3ADSP DM6437.

Installing the Avnet Board Support Library

Download and install the current Avnet Board Support Package for Simulink (Avnet BSL), available from the Avnet Web site, www.avnet.com. Doing so creates environment variables that the Embedded Coder software uses to locate files in the Avnet BSP.

Make a note of the installation folder for the Avnet BSL.

Setting the MATLAB Environment

The Embedded Coder software uses environment variables to locate files in the Avnet BSP.

The MathWorks utility, setTgtEnv.m, automatically maps the following environment variables (where <Avnet BSL> is the Avnet BSL installation folder):

- PSP_EVMDM6437_INSTALLDIR: must be mapped to "<Avnet BSL>\psp"
- CSLR_DM6437_INSTALLDIR: must be mapped to "<Avnet BSL>\psp\pspdrivers\soc\dm6437\dsp\inc"
- NDK_INSTALL_DIR: must be mapped to "<Avnet BSL>\ndk"

Run setTgtEnv by entering the following command at the MATLAB command prompt: `setTgtEnv('avnet_s3adsp_dm6437')`

If you installed the Avnet BSL prior to installing the MathWorks BSL, the utility detects the AVNET_S3ADSP_DM6437_INSTALLDIR environment variable created by the Avnet BSL installer. It will automatically set the environment variables above based on the path stored in the AVNET_S3ADSP_DM6437_INSTALLDIR environment variable. On a successful run, you should see the following messages printed on the MATLAB command window:

```
Setting environment variable "PSP_EVMDM6437_INSTALLDIR" to
C:\avnet_s3adsp_dm6437_1_06\psp"
```

```
Setting environment variable "CSLR_DM6437_INSTALLDIR" to
C:\avnet_s3adsp_dm6437_1_06\psp\pspdrivers\soc\dm6437\dsp\inc"

Setting environment variable "NDK_INSTALL_DIR" to
C:\avnet_s3adsp_dm6437_1_06\ndk"
```

If automatic mapping fails for any reason, the script will prompt you to browse for the “avnet_s3adsp_dm6437_version.txt” file stored in the top-level Avnet BSL installation folder. If so, browse for the file and click the **Open** button. This will set the required environment variables.

For Spectrum Digital DM6437EVM Users

If you have a Spectrum Digital DM6437EVM board together with an Avnet S3ADSP DM6437 board, setting environment for the Avnet board as explained in section 2.3 will override DM6437EVM environment setup. To revert back to DM6437EVM environment after using Avnet board, execute the following at the MATLAB command prompt: `setTgtEnv('dm6437evm')`

Follow the instructions printed on the MATLAB command window to complete environment configuration. To go back and forth between DM6437EVM environment and Avnet S3ADSP DM6437 environment, use the setTgtEnv script with the appropriate platform name specified as the argument.

Verifying Your Installation

Open the Avnet S3ADSP Board Support Library by entering the following command at the MATLAB command prompt: `avnet_s3adsp_dm6437` This opens the **Avnet Spartan–3A DSP DaVinci Evaluation Platform Board Support Library**. You have completed installing and configuring the MathWorks and Avnet Board Support Libraries. You are ready to start using the Avnet S3ADSP DaVinci evaluation platform.

Continuing Issues with Embedded Coder Software

This section details some target operations that you should know about as you use Embedded Coder software.

- “Setting the Clock Speed on the C6713 DSK” on page 56-119

- "Simulink Stop Block Works Differently When Not Using DSP/BIOS Features" on page 56-120
- "Installing Third-Party Embedded Coders" on page 56-120

Setting the Clock Speed on the C6713 DSK

The C6713DSK PLL is not automatically set to the correct CPU Clock frequency when you try to target the board. When you power-up your DSK, it runs at a clock speed of 50 MHz. However, the C6713 is capable of running at 225 MHz.

If you generate code incorporating the DSP/BIOS real-time operating system, the PLL is automatically configured for you at run-time to use the correct clock speed. If you are not using DSP/BIOS in your project, you must manually configure the PLL to the correct clock rate before running your code.

Setting the PLL to Drive the CPU at 225 MHz. To set the C6713 DSK PLL to drive the CPU at 225 MHz, perform the following steps. Be sure you have defined your GEL file for your DSK in the Setup Utility for CCS IDE.

1. Launch Code Composer Studio.
2. Open your C6713 DSK project with the GEL file.
3. Select **GEL > Resets > InitPLL** from the menu bar in CCS IDE.

To make this happen whenever you open Code Composer Studio to use your C6713 DSK, edit the file `\ti\IDE_Obj\gel\dsk6713.gel`. Add the following command to the `StartUp()` function:

```
init_pll();
```

This tells the GEL file to initialize the PLL to operate at 225 MHz.

On the DM642 EVM, ADC-DAC Loopback Does Not Display An RGB Image Correctly After Power-Up. When you set up the DM642 EVM to use loopback from the ADC to the DAC, the DAC block does not reproduce the captured image correctly immediately after you power up the board. Colors in the image are not shown correctly.

To get a clean image, reload the program to the target and run the program again. This also happens with the examples Texas Instruments ships with the DM642 EVM product.

Simulink Stop Block Works Differently When Not Using DSP/BIOS Features

If you are using the Simulink Stop block in your model, but you are not using DSP/BIOS features, your model might take longer to stop when it is running on the target than if you are using DSP/BIOS.

The condition the model uses to detect the stop processing flag is different when you do not use DSP/BIOS. The result is that the model may not detect and respond to the flag as promptly, taking longer to stop the running model on the target.

Installing Third-Party Embedded Coders

For a list of required third-party target packages, with version numbers, see the Embedded Coder System Requirements page at http://www.mathworks.com/products/target-package/requirements.html.

When you install any of the third-party target packages listed below, perform a default installation using the installation path provided for that package and perform any additional steps given.

This documentation uses placeholders for portions of the installation path that may vary by software version or environment. Please replace the placeholders with the correct path information for your software environment. For example, if the CCS IDE installation path is `C:\CCStudio_v3.3`, then enter `C:\CCStudio_v3.3\boards\evmdm642` instead of `<CCStudio_vn.n>\boards\evmdm642`.

Placeholders:

- `<CCStudio_vn.n>` — The installation path for Code Composer Studio
- `<n.n>` — Version-specific path information

Note If you do not use the installation paths provided, update the **Libraries** and **Include paths** parameters in the Target Preferences and C6000 IP Config blocks of the Embedded Coder™ software with the correct paths. Otherwise, the software produces error messages when you attempt to generate code.

DM642EVM Version 3 Board.

- Spectrum Digital EVMDM642 Board Support Package — <CCSvn.n>\boards\evmdm642
- TI's Network Developer's Kit (NDK) — <CCSvn.n>\C6000\NDK

DM642EVM Version 1 & 2 Boards.

- Spectrum Digital EVMDM642 Board Support Package — <CCSvn.n>\boards\evmdm642
- Device Driver Developer's Kit (DDK) — <CCSvn.n>\ddk
- TI's Network Developer's Kit (NDK) — <CCSvn.n>\C6000\NDK

DM6437EVM.

- Spectrum Digital DM6437EVM DVSDK RTM — Install anywhere. TI recommends using the root path of your main drive. For example, C:\dvsdk_<n.n>

Also, set the following environment variables, replacing DVSK with the DVSDK installation path (e.g., C:\dvsdk_<n.n>).

The first time you generate code, the Embedded Coder™ software prompts you to locate specific files in the DVDSK folders and creates environment variables mapped to the location of required folders. For example, the application creates an environment variable called CSLR_DM6437_INSTALLDIR for the path of the Register Layer Chip Support Library.

C6455DSK. Spectrum Digital DSK6455/EVM6455 Target Content Package — <CCSvn.n>\boards\dsk6455_v<n.n>

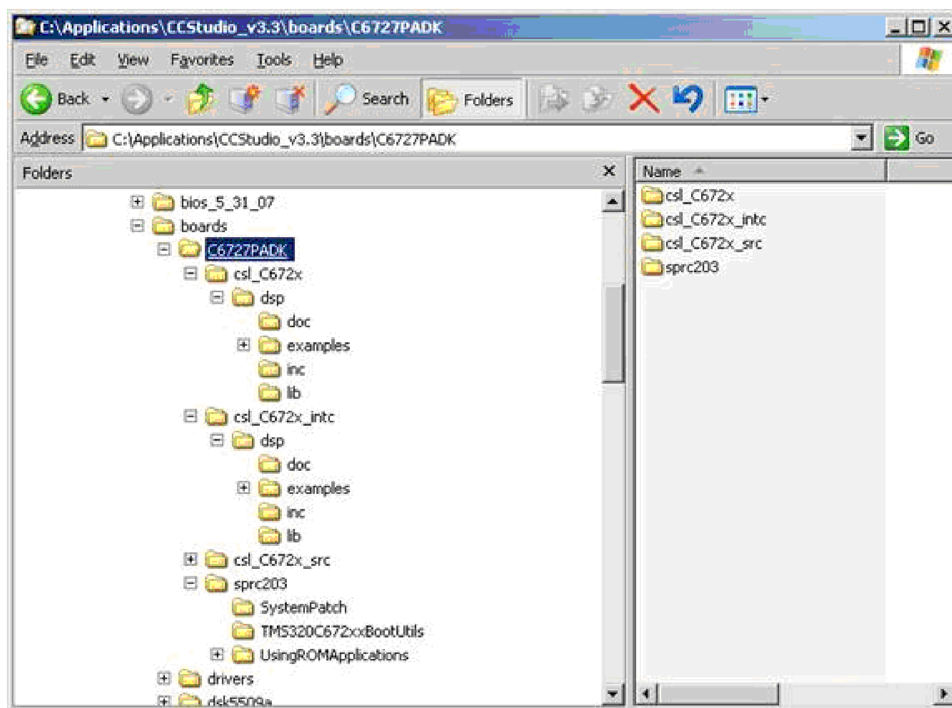
Network Developer's Kit NDK — `<CCSvn.n>\C6000\NDK`

C6727PADK. Lyrtech's PADK Software — Install anywhere.

TI's C672x Chip Support Libraries (CSL) — Extract all three C672x CSL components from **sprc223.zip** to `<CCSvn.n>\boards\C6727PADK`.

TI's System Patch Code, FastRts(V<n.n>)/DSPLIB (V<n.n>) — `<CCSvn.n>\boards\C6727PADK\sprc203`

After installation, the path structure for the C672x CSL libraries should resemble the following figure:



The PADK Software installer automatically sets the PADK_DIR environment variable with the correct installation path.

The first time you generate code, the Embedded Coder™ software prompts you to locate the following files under <CCSvn.n>\boards\C6727PADK\ and sets the environment variables accordingly:

- `$(CSL_C672x_INSTALLDIR)\lib\csl_C6727.lib`
- `$(CSL_C672x_INTC_INSTALLDIR)\lib\csl_C672x_intc.lib`
- `$(SYSPATCH_C672x_INSTALLDIR)\applySystemPatch.obj`

You have completed installation of the third-party softwares.

Working with Wind River VxWorks Target

- “Overview of Support for Wind River VxWorks” on page 57-2
- “Tutorial: Building and Running Embedded Software on VxWorks” on page 57-4
- “Generating Code for VxWorks Running on Other Targets” on page 57-9
- “Schedulers” on page 57-10

Overview of Support for Wind River VxWorks

Using Embedded Coder software, you can generate software for the VxWorks RTOS.

- Real-time scheduler
- Non real-time (free-running) scheduler
- Single-tasking and multitasking modes

You can use the following MathWorks software features to verify that the target software you create for VxWorks:

- External Mode simulation
- Processor-in-the-loop simulation over TCP/IP (TCP/IP PIL)

Additionally, the Simulink Library Browser contains a new VxWorks block library, `vxworkslib`, with the following blocks:

- VxWorks Task
- UDP Receive
- UDP Send

You can find `vxworkslib` in the Simulink Library Browser under **Embedded Targets** > **Operating Systems** > **VxWorks**.

In this release, Embedded Coder cannot generate IDE projects for the Wind River Diab or GNU tool chains. Instead, use the XMakefile feature to generate makefiles, source code, and related files from your model. You can then use those files with to build, load, and run your embedded software on VxWorks. For more information, see "IDE Projects" on page 43-18 and "Makefiles" on page 43-24.

The XMakefile feature includes the following new configuration files:

- `wrsdiab_arm9_vxworks_rtp` — Diab tool chain, ARM9 processor, VxWorks real-time process (RTP) applications

- `wrsdiab_arm9_vxworks_rtp_so` — Diab tool chain, ARM9 processor, VxWorks real-time process (RTP) applications with shared object
- `wrsdiab_hostsim_vxworks_rtp` — Diab tool chain, host-simulator, VxWorks real-time process (RTP) applications
- `wrsdiab_hostsim_vxworks_rtp_so` — Diab tool chain, host-simulator, VxWorks real-time process (RTP) applications with shared object
- `wrsgnu_arm9_vxworks_rtp` — GNU tool chain, ARM9, VxWorks real-time process (RTP) applications
- `wrsgnu_hostsim_vxworks_rtp` — GNU tool chain, host-simulator, VxWorks real-time process (RTP) applications

For a demonstration of how to develop embedded software for VxWorks, see the Code Generation and Verification demo.

Tutorial: Building and Running Embedded Software on VxWorks

In this section...

"Install and Set Up the Wind River Development Environment" on page 57-4

"Setting VxWorks Environment Variables and Starting MATLAB" on page 57-5

"Setting Up XMakefile for VxWorks" on page 57-6

"Customizing XMakefile to Automatically Download and Build Your Software" on page 57-7

"Prepare Your Model for VxWorks and Makefiles" on page 57-8

"Build Your Embedded Software" on page 57-8

This tutorial shows you how to use the XMakefile feature in your MathWorks software to build and run embedded software for VxWorks. For more information about XMakefile, see "Makefiles" on page 43-24.

Install and Set Up the Wind River Development Environment

Set up VxWorks software and hardware on your host and target:

1. Install Wind River Workshop on the host.

2. Set up host-to-target communications.

3. Create a VxWorks operating system kernel image on the host.

4. Boot the embedded target with the kernel image.

For detailed instructions, consult your Wind River Workshop and VxWorks documentation.

Setting VxWorks Environment Variables and Starting MATLAB

Each time you use VxWorks with your MathWorks software, start by:

- Setting the VxWorks environment variables
- Starting MATLAB

The XMakefiles feature in your MathWorks software uses standard input/output (command line) to start and communicate with the tool chain in Wind River Workbench. Wind River recommends that you set environment variables using the wrenv utility each time you start those tools from the command line. For more information, read the `readme.txt` file in the Wind River installation folder, and search the Wind River Workbench help for “Setting Environment Variables With wrenv”.

Set the VxWorks environment variables, and then start MATLAB:

1. Open a command-line session in Windows or Linux.
2. Change folders to the Wind River installation folder. For example, at the Windows command line, enter:

   ```
   cd C:\WindRiver
   ```

3. Run the `wrenv` utility, including `-p` followed by the relative path to the VxWorks platform you are using.

   For example, in Windows, enter:

   ```
   wrenv.exe -p vxworks-6.7
   ```

   For example, in Linux, enter:

   ```
   ./wrenv.sh -p vxworks-6.7
   ```

4. Start MATLAB. For example, enter:

   ```
   C:\Program Files\MATLAB\R2010a\bin\matlab.exe
   ```

Setting Up XMakefile for VxWorks

The XMakefile feature tells your MathWorks software how to create makefiles for a “configuration”, which is a specific combination of tool chain and embedded target. Some configurations require additional information before you can use them.

Select and complete a configuration for VxWorks:

1. Enter `xmakefilesetup` at the MATLAB command prompt. This action opens the XMakefile User Configuration dialog box.
2. Deselect **Display operational configurations only**.
3. Set **Configurations** to a choice that starts with `wrs` and contains `vxworks`, and click **Apply**.
4. If the configuration is incomplete, the software displays a series of **Browse For Folder** dialog boxes that include instructions to provide missing information.
5. Examine the **Tool Directories** tab to see if the paths are correct.
6. When you have supplied the missing information and the configuration is complete, click **OK** to close the XMakefile User Configuration dialog box.

For example, to generate code for VxWorks and an ARM9 processor:

1. Enter `xmakefilesetup` on the command line.
2. In the XMakefile dialog box, deselect **Display operational configurations only**, set **Configurations** to `wrsdiab_arm9_vxworks_rtp`, and click **Apply**.
3. When the **Browse For Folder** dialog box appears, stating “Select the Wind River Diab compiler binary directory...”, browse and select a path.

   For example, you can select a path such as `C:\WindRiver\diab\5.7.0.0\WIN32\bin`.
4. When another **Browse For Folder** dialog box appears, stating “Select the Wind River root installation directory...”, browse and select a path.

For example, you can select a path such as `C:\WindRiver`.

5 Examine the **Tool Directories** tab to see if the paths are correct.

6 After you complete the update of configuration information, click **OK** to save the updated configuration and close the dialog box. The `wrsdiab_arm9_vxworks_rtp` configuration is now operational.

Customizing XMakefile to Automatically Download and Build Your Software

Create a customized XMakefile configuration that automatically builds and downloads your embedded software to the VxWorks target.

1 Enter `xmakefilesetup` at the MATLAB command prompt. This action opens the XMakefile User Configuration dialog box.

2 Verify that **Configurations** displays the `wrsdiab_arm9_vxworks_rtp` configuration.

3 Click **New**, and name the new configuration.

4 On the **Post-build** tab:

- For **Post-build tool**, enter `echo`
- For **Arguments**, enter the following text *as a single line*:

```
rtpSp \"host:[|||MW_XMK_GENERATED_TARGET_REF[E]|||]\">"
[|||MW_XMK_OUTPUT_PATH_REF[E]|||]vxscript.txt"
```

5 On the **Execute** tab:

- For **Execute tool**, enter the complete path to the VxWorks simulator executable. For example:

```
C:\WindRiver\vxworks-6.7\host\x86-win32\bin\vxsim.exe
```

- For **Arguments**, enter the following string. Substitute *`vxworksimage`* with the complete path to the VxWorks operating system kernel image you created earlier:

```
-f vxworksimage -s
"[|||MW_XMK_OUTPUT_PATH_REF[E]|||]vxscript.txt" &
```

Prepare Your Model for VxWorks and Makefiles

The Target Preferences block contains information that your MathWorks software needs to generate code for a specific combination of tool chain and embedded target.

Configure your model to generate code VxWorks by updating the Target Preferences block.

If your model does not contain a Target Preferences block:

1. Open the Common block library by entering `idelinklib_common` on the command line.
2. Copy the Target Preferences block to your model.
3. Complete the **Target Preferences: Initialize Configuration Parameters** dialog box:
   - Set **IDE/Tool Chain** to: `Wind River Diab/GCC (makefile generation only)`.
   - Set **Processor** to `ARM9`.
4. Click **Yes**, to update the appropriate values in your Target Preferences and model Configuration Parameters.

If your model already contains a Target Preferences block, open the block and update the parameters described in step 3 and 4.

Build Your Embedded Software

In your model, build your embedded software by entering **CTRL+B**. This action causes your MathWorks software to generate code and makefiles. Then the Wind River tool chain builds and loads the embedded software on the VxWorks target.

Generating Code for VxWorks Running on Other Targets

The XMakefile feature includes configurations for the ARM9 processor. To generate makefiles for VxWorks running on other targets, such as the ARM9E, ARM10, ARM11, or generic/custom processors:

1. Start the XMakefile dialog box by typing `xmakefilesetup` at the command line.

2. Click **New**, and name the new configuration.

3. Update the new configuration with any missing information and new arguments. Consult the Wind River documentation for information on which arguments to provide.

Schedulers

Running Target Applications on Multicore Processors

If you are generating code for a processor running Linux or VxWorks, you can elect to partition the code such that each rate is placed in its own thread. With code generated from a multi-rate model, the multi-threaded application will be enabled for concurrent multicore execution, as scheduled by the target operating system.

1. Create a multi-rate Simulink model.

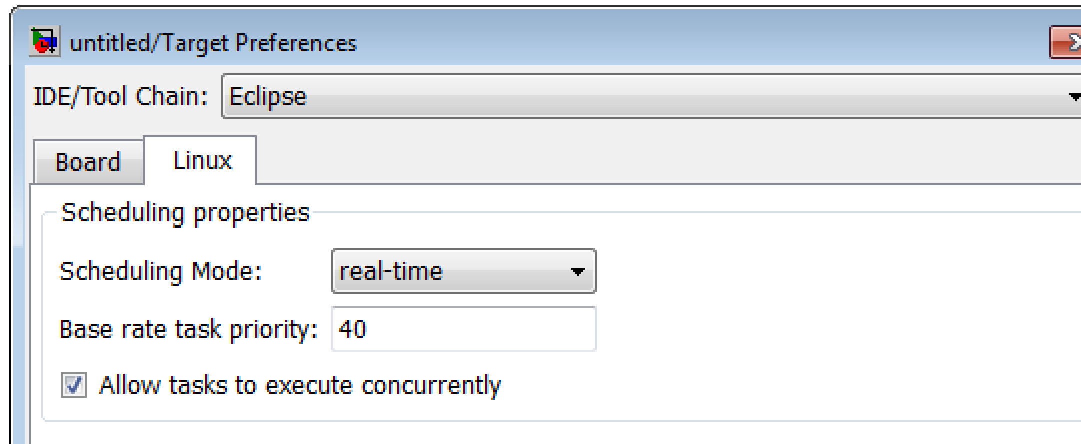
2. Add a Target Preferences block to your model as described in the "Target Preferences" on page 43-4 section.

3. Verify that your model uses a Rate Transition block to transition between rates.

4. Clear the **Ensure deterministic data transfer** checkbox of the Rate Transition block. This action forces the Rate Transition block to use the most recent data available.

5. In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

6. In the Linux or VxWorks tab of the Target Preferences block, select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables the generated multi-threaded code to run concurrently on multicore processors.



7. In the Configuration Parameters dialog box, on the Solver pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

8. Open the Rate Transition block and clear the **Ensure deterministic data transfer** checkbox. This action forces the Rate Transition block to use the most recent data.

9. In the Target Preferences block, set **Operating System** to `Linux` or `VxWorks`.

10. Select the **Allow tasks to execute concurrently** checkbox. Selecting this option enables generated multi-threading code to run concurrently on multicore processors.

11. For the best performance, in the Configuration Parameters dialog box, on the **Solver** pane, set **Tasking mode for periodic sample times** to `Auto` or `Multitasking`.

12. In your model, click the build button or enter **Ctrl+B**. The software performs the actions you selected for **Build action** in the model Configuration Parameters, under Code Generation > IDE Link.

Examples

Use this list to find examples in the documentation.

Code Generation

"Generating a Shared Library Version of Your Model Code" on page 4-18
"Creating Application Code to Load and Use Your Shared Library File" on page 4-19
"Specifying Code Generation Objectives Using the GUI" on page 15-4
"Specifying Code Generation Objectives at the Command Line" on page 15-6
"Reviewing the Model Without Generating Code" on page 15-7
"Reviewing the Model During Code Generation" on page 15-9
"How to Create Custom Objectives" on page 15-14
"Using Virtualized Output Ports Optimization" on page 21-24
"Importing an AUTOSAR Software Component" on page 24-28
"Creating a SIL Block" on page 26-3
"Techniques for Exporting Function-Call Subsystems" on page 27-7
"Function-Call Subsystem Export Example" on page 27-12
"Examples of Modular Function Code for Nonvirtual Subsystems" on page 28-9
"Model Function Prototypes Example" on page 29-12
"Sample Script for Configuring Model Function Prototypes" on page 29-22
"C++ Encapsulation Quick-Start Example" on page 30-4
"Sample Script for Configuring the Step Method for a Model Class" on page 30-24

Custom Storage Classes

"Creating Packages that Support CSC Definitions" on page 8-8
"Generating Code with Custom Storage Classes" on page 8-58
"Defining Advanced Custom Storage Class Types" on page 8-62
"GetSet Custom Storage Class Example" on page 8-68

Memory Sections

"Requirements for Defining Memory Sections" on page 9-4
"Defining Memory Sections" on page 9-7

"Applying Memory Sections" on page 9-15
"Examples of Generated Code with Memory Sections" on page 9-23

Advanced Code Generation

"Specifying Type Definition Location for User-Defined Data Types" on page 11-5
"Generating Source and Header Files with a Custom File Processing (CFP) Template" on page 17-35
"Creating a Custom File and Function Banner Template" on page 17-53
"Customizing a Code Generation Template (CGT) File for File and Function Banner Generation" on page 17-54
"Adding a Configuration Wizard Block to Your Model" on page 21-9
"Creating a Custom Configuration Wizard Block" on page 21-11

Target Function Libraries

"Target Function Libraries Quick-Start Example" on page 31-9
"Example: Mapping Math Functions to Target-Specific Implementations" on page 31-27
"Example: Mapping the memcpy Function to a Target-Specific Implementation" on page 31-34
"Example: Mapping Nonfinite Support Utility Functions to Target-Specific Implementations" on page 31-38
"Example: Mapping Scalar Operators to Target-Specific Implementations" on page 31-43
"Adding Target Function Library Reserved Identifiers" on page 31-137
"Examining and Validating Function Replacement Tables" on page 31-139
"Using the sl_customization API to Register a TFL with Simulink Software" on page 31-149
"Registering Multiple TFLs" on page 31-154

Data Structures, Code Modules, and Program Execution

“Rate Grouping and the Static Main Program” on page 34-15
“Making Your S-Functions Rate Grouping Compliant” on page 34-19

Verifying Generated Code

“Demos of the Target Connectivity API” on page 39-59

Makefiles

“Example: Creating a New XMakefile Configuration” on page 43-31

Verification

“By Tasks” on page 44-15
“By Subsystems” on page 44-17
“Profiling System Stack Use” on page 44-22

Tutorials

“Tutorial: Using Option Sets” on page 46-75
“Tutorial: Creating New Template Projects” on page 46-76
“Creating a New Configuration” on page 46-80
“Tutorial: Configuring an Existing Model for Embedded Coder Software” on page 46-81

Automation Interface

“Getting Started with Automation Interface” on page 47-7

“Getting Started with Automation Interface” on page 50-10
“Getting Started with Automation Interface” on page 54-10

Working with adivdsp Objects

“Example — Constructor for adivdsp Objects” on page 47-23
“Example — Setting Object Property Values at Construction” on page 47-24
“Example — Setting Object Property Values Using set” on page 47-25
“Example — Retrieving Object Property Values Using get” on page 47-25
“Example — Direct Property Referencing in Links” on page 47-26

Project Generator

“Project Generator Tutorial” on page 47-31
“Project Generator Tutorial” on page 50-34
“Project Generator Tutorial” on page 54-60

Real-Time Target

“Tutorial: Creating a New Application” on page 49-26
“Using External Mode” on page 49-54
“ASAP2 File Generation Procedure” on page 49-65
“Data Acquisition (DAQ) List Configuration” on page 49-66
“Using External Mode” on page 55-35

Processor-in-the-Loop Target

“Tutorial 1: Building and Running a PIL Simulation” on page 49-84
“Using the Demo Model In a PIL Simulation” on page 49-94

Algorithm Export Target

"Algorithm Export Target" on page 49-103

Working with ghsmulti Objects

"Example — Constructor for ghsmulti Objects" on page 50-26
"Example — Setting Link Property Values at Construction" on page 50-28
"Example — Setting Link Property Values Using set" on page 50-29
"Example — Retrieving Link Property Values Using get" on page 50-29
"Example — Direct Property Referencing in Links" on page 50-30

Simple Example Applications

"Example Model 1: c166_serial_transmit" on page 51-24
"Example 2: c166_serial_io" on page 51-28
"Debugging and Using The Code Profile Report" on page 51-30
"RAM / ROM Code Profile Report" on page 51-31
"Parameter Tuning and Signal Logging" on page 51-34
"Parameter Tuning and Signal Logging" on page 55-34

Integrating Manually Coded Device Drivers

"Tutorial: Using the Example Driver Functions" on page 51-43

Bit-Addressable Memory

"Using the Bitfield Example Model" on page 51-50

Execution Profiling

"Multitasking Demo Model" on page 51-62

Working with ticcs Objects

"Getting Started with RTDX" on page 54-27
"Example — Constructor for ticcs Objects" on page 54-49

Exporting Filter Coefficients from FDATool

"Exporting Filter Coefficients from FDATool to the CCS IDE Editor" on page 54-75
"Example: Changing Filter Coefficients Stored on Your Processor" on page 54-82

Q Format Examples

"Example — Q.15" on page 55-57
"Example — Q1.30" on page 55-58
"Example — Q-2.17" on page 55-58
"Example — Q17.-2" on page 55-59
"Example — Q.15" on page 56-104
"Example — Q1.30" on page 56-104
"Example — Q-2.17" on page 56-105
"Example — Q17.-2" on page 56-105

Targeting Tutorials

"Targeting Tutorial — Single Rate Application" on page 56-11
"Targeting Tutorial II — A More Complex Application" on page 56-43

Asynchronous Scheduler

"Asynchronous Scheduler Examples" on page 56-24
"Free-Running DSP/BIOS Task" on page 56-28
"Idle Task" on page 56-30
"Hardware Interrupt Triggered DSP/BIOS Task" on page 56-31
"Hardware Interrupt Triggered Task" on page 56-32

Profiling Code

"Profiling Multitasking Systems" on page 56-82
"Profiling Your Generated Code" on page 56-89

Target Preferences

"Example: Creating a Custom Target Preferences Block for OMAP-L138/C6748 EVM" on page 56-96

Index

A

absolute time 32-3
access properties 47-24 50-28
acquisition window
  ADC blocks
    ACQ_PS 55-47
add build subdirectory suffix 46-15
Add IDE Link Configuration to Model 46-23
additional options
  adding custom comments 17-4
  delimiter for all #includes 17-69
adivdsp 47-22
adivdsp object properties 47-29
  procnum 47-28
  sessionname 47-29
algorithm export 49-103
algorithms
  verifying in context of complete real-time target environment 41-1
Alias 7-9
Analog Devices model reference 47-35
apiversion 54-53
Archive_library 47-37 50-41 54-66 56-35
ASAP2 files
  generating for C166 51-31
ASAP2 files, generating 49-64 51-35 55-45
asynchronous interrupt processing 55-7
asynchronous scheduling 56-21
attributes 12-3
AUTOSAR 22-2
  standards, complying with 22-5

B

bit-addressable memory 51-49
Bitfield (Custom) 7-6
block limitations using model reference 47-38 50-42 54-67 56-36
blocks
  adding to model 55-26
boardnum 54-54
boards, selecting 43-21 54-58
Browse button
  on Code Generation pane 16-2
build action 46-13
  setting 46-18
build configuration 46-13
build folder
  contents of 56-50
  naming convention 56-45
build format 43-12
build process
  command line information 46-35
  folder structure 46-33
  overview 46-27
  shared libraries 46-31
  template projects 46-29
build subdirectory suffix 46-15
building models
  use C62x DSP Library blocks 56-106

C

c2000lib startup 55-24
C6000 model reference 54-65 56-34
C6000 Target
  targeting Code Composer Studio 56-54
C62x DSP Library blocks
  building models 56-106
  choosing blocks to optimize code 56-107
  common characteristics 56-101
  Q format notation 56-103
  using source and sink blocks 56-107
C6713 DSK
  confirming proper configuration 56-42
  start/stop models 56-40 56-54
  tutorial about multirate applications 56-43
C6713 DSK blocks
  tutorial 56-43
C6713 DSK folders

build 56-45
working 56-45
CAN
timing parameters
Bitrate 56-109
CAN/eCAN
timing parameters
bit rate 55-29
CCS 54-2
See also Code Composer Studio™
CCS IDE
create projects for the IDE 56-54
CCS IDE objects
tutorial about using 54-10
`ccsappexe` 54-54
changing identifier names 12-30
classes 12-3 46-38
clock speed 55-7
code analysis report 49-104
Code Composer Studio 56-54
MATLAB API 54-5
Code Composer Studio™ 54-2
code generation
overview 55-28
code generation options
Application lifespan (days) 21-3
Bitfield declarator type specifier 21-4
code style pane 17-22
Configure Step Function 32-6
Custom comments 17-2
External mode 21-26
Fixed-point exception protection 21-3
Generate reusable code 32-5
Generate scalar inlined parameters 17-13
GRT compatible call interface 32-5
Identifier format control 17-12
MAT-file logging
clearing 21-23
Maximum identifier length 17-12
Minimum mangle length 17-12
Pack Boolean data into bitfields 21-4
Parameter structure 21-4
Pass reusable subsystem outputs as 21-4
Pass root-level I/O as 32-7
Requirements in block comments 17-3
Reusable code error diagnostic 32-7
Simplify array indexing 21-4
Simulink block descriptions 17-2
Simulink data object descriptions 17-2
Single output/update function 32-5
clearing 34-17
Stateflow object descriptions 17-3
Support absolute time 32-3
Support complex numbers 32-2
Support continuous time 32-3
for using continuous time blocks 5-2
limitations 26-4
Support floating-point numbers 32-2
Support non-finite numbers 32-2
Support noninlined S-functions 32-4
Suppress error status in real-time model data structure 32-6
Terminate function required 32-5
Code generation options 7-4
Code Generation pane
target configuration options
Browse button 16-2
system target file field 16-2
Code Generation Report 12-25
code modules, generated 19-2
code optimization 55-60
code style
controlling 17-22
code templates
example of use 17-35
generating code with 17-36
structure of 17-31
summary of API 17-47
code tracing
by using HTML reports 37-2

code, generated
verifying in target environment 40-1
code, user-written 19-5
codegen
generating reusable, reentrant code 23-1
comments
adding custom 17-4
adding global 17-5
`Complexity` 7-3
component models
verifying in context of complete real-time target environment 41-1
components
, project generator 46-37
project generator 46-27
Configuration Class blocks 49-23 51-19
configuration default 54-2
configuration options 46-13
configuration parameters
Code Generation Pane: ET MPC5xx (Algorithm Export) Options Tab 49-109
Code Generation Pane: ET MPC5xx (Processor-in-the-Loop) Options Tab 49-111
Code Generation Pane: ET MPC5xx Real-Time Options (1) Tab 49-116
Code Generation Pane: ET MPC5xx Real-Time Options (2) Tab 49-119
pane 46-84
Add build directory suffix 46-88
Build action 49-115
Build directory suffix: 46-89
Compiler optimization switches 49-113
Configure model to build PIL algorithm object code 46-94
CrossView Pro handle name: 46-92
EDE handle name: 46-90
Execution profiling 49-122
Export CrossView Pro handle to MATLAB base workspace 46-92
Export EDE handle to MATLAB base workspace 46-90
Maximum number of concurrent base-rate overruns: 49-120
Maximum number of concurrent sub-rate overruns: 49-121
Number of data points: 49-122
Optimize compiler for 49-113
Target Memory Model 49-117
TaskingBuildAction 46-85
TaskingConfiguration 46-87
Use prebuilt (static) libraries 49-111
Configuration Parameters
Code Generation Pane: C166 Options Tab 51-72
pane
Execution profiling 51-77
Include input/output driver function hooks 51-73
Maximum number of concurrent base-rate overruns: 51-74
Maximum number of concurrent sub-rate overruns: 51-75
Number of data points: 51-78
Configuration Parameters dialog box 18-3
configuration sets 46-13
Configuration Wizard buttons 21-7
Configure model to build PIL algorithm object code 46-15
confirm your C6713 DSK configuration 56-42
`Const (Custom)` 7-7
`ConstVolatile (Custom)` 7-7
controllers
verifying in context of complete real-time target environment 41-1
controlling signal storage 21-25
convert data types 56-106
CPU clock speed 55-7
Create a New Model (configured for use with TASKING) 46-22

create custom target function library 45-8
Create New Template Projects 46-22
creating a data dictionary 12-4
CrossView Pro handle name 46-15
custom C6000 target
  about 56-59
  preferences block 56-59
  setup 56-59
custom code generation
  of file banners 17-50
  with code templates 17-31
custom comments 17-4
custom file processing (CFP) template 17-23
custom hardware guidelines 56-55
custom hardware, target 56-55
custom storage class 51-49

D

`daexplr` command 12-8
`Data access` 7-8
data dictionary 12-2
  introduction 12-2
  See also data objects
data initialization
  of floats and doubles 21-2
  of internal states 21-2
  of root-level I/O ports 21-2
data object wizard 12-5
data objects
  adding missing 12-5
  naming rules
    changing all `#defines` 12-36
    changing all parameter names 12-35
    changing all signal names 12-35
  properties 7-2
  setting property values 12-8
  wizard 12-5
data placement
  introduction 13-2
  rules for 13-25
  settings 13-2
data templates 17-23
`Data type` property 7-3
data type support 55-5
data types
  conversion 55-60
  creating 12-38
`dataobjectwizard` 12-6
`Default (Custom)` storage class 7-5
default build configuration 54-2
`Define (Custom)` 7-7
`#defines`
  changing all 12-36
defining all objects in separate file 12-26
defining one object in its own file 12-28
`Definition file` 7-6
Definition File priority 13-8
Demos 46-23
`Description` 7-9
design specification
  developing 3-1 4-1 5-1
designs, model
  optimizing for specific hardware with
    on-target rapid prototyping 38-1
device driver blocks
  input data types 49-20
  input scaling 49-20
  output data types 49-20
  output scaling 49-20
dialog boxes
  AUTOSAR Options 24-31
  Model Interface 29-4
Dialog boxes
  Model Explorer 12-8
`Dimensions` 7-3
`Direct` 7-8
DO-178B 22-2
  standards, complying with 22-10
DO-178B Standard 22-10

DO178-B 22-2
DocBlock 17-6
domain
  installing products for 3-1
downloading code 51-26
downloading code to target 49-39
  application code 49-42
    to flash memory 49-44
    to RAM 49-43
DSP/BIOS
  added files 56-78
  files removed from project 56-78
  to enable 56-92
DSP/BIOS, enabling 56-92

E

Eclipse™ IDE for C/C++ Developers 48-4
EDE handle name 46-15
elapsed time 32-3
Embedded Coder™
  about 56-2
  create Simulink® model for targeting 56-43
  expected background for use 56-2
  information for new users 56-3
  listing link functions 54-48
Embedded Coder™ for use with Altium® TASKING®
  build process 46-27
  objects 46-37
Embedded Coder™ product
  PIL simulation 46-50
Embedded Coder™ software
  IDE Link Utilities for Use with TASKING dialog 46-21
  introduction 46-2
  limitations and tips 46-95
  supported toolsets 46-6
  target preferences 46-8
  Tools menu 46-23
  tutorials 46-75
  user guide 46-7
enabling DSP/BIOS 56-92
entry points, model 33-1
`ert_main.c` 34-14
`ert_main.cpp` 34-14
example model
  c166_bitfields 51-49
  c166_fuelsys 51-31
  c166_multitasking 51-56
  c166_serial_io 51-28
  c166_serial_transmit 51-24
  c166_user_io 51-38
execution in timer-based models 56-22
execution profiling 51-56
  subsystem 44-17
Export CrossView Pro handle to MATLAB base workspace 46-15
Export EDE handle to MATLAB base workspace 46-15
export filters to CCS IDE from FDATool 54-69
  select the export data type 54-72
  set the Export mode option 54-71
Export handles 46-15
`ExportToFile (Custom)` 7-8
external data dictionary
  importing data objects from 12-16
External mode support 21-26

F

FDATool. *See* export filters to CCS IDE from FDATool
file banners, generation of 17-50
file packaging 19-2
files added to DSP/BIOS project 56-78
files removed from DSP/BIOS projects 56-78
fixed-point example 51-31
fixed-point numbers 55-54 56-101
  signed 56-102

Frame based 7-4
functions
    overloading 47-26 50-30 54-50

G

generate optimized code 43-11
generated code
    modules 19-2
    verifying in target environment 40-1
generating code 51-26
generating reusable, reentrant code
    codegen 23-1
`Get function` 7-9
`GetSet (Custom)` 7-8
GetSet custom storage class 8-66
getting properties 47-25 50-29
`ghsmulti` 50-26
`ghsmulti object properties` 50-32
    `portnum` 50-32
    `procnum` 50-31
`Global (Custom)` storage class 7-5
global comments
    using DocBlock 17-6
    using Simulink annotation 17-9
    using sorted notes 17-10
    using Stateflow note 17-9
Global priority 13-7
GNU Tool Chain on Linux® 48-5
GNU Tool Chain on Windows 48-7
Green Hills MULTI® IDE objects
    tutorial about using 50-10
Green Hills Software model reference 50-39
guidelines
    MISRA C® 22-5

H

hardware 56-4
Hardware Implementation parameters
    configuration of 21-20
hardware, custom 56-55
`Header file` 7-6
Header File priority 13-8
heap size, set heap size 43-15
high-speed peripheral clock 55-7

I

identifier format control parameters 17-15
identifier format control tokens 17-14
IEC 61508 22-2
    standards, complying with 22-6
IEC 61508 Standard 22-6
import filter coefficients from FDATool.. *See* FDATool
`ImportFromFile (Custom)` 7-8
inaccurate profile information 56-82
`#include`
    specifying delimiter 17-69
industry standards and guidelines
    modeling and coding 22-2
initialized memory 56-58
inserting custom comments 17-4
inserting global comments 17-5
installing software 55-2 56-5
integer-only code 21-23
integer-only code generation 21-23
integrating manually coded device drivers 51-38
interrupts, servicing 34-4
intrinsics. *See* target function library
IQ Math library 55-53
    building models 55-59
    code optimization 55-60
    common characteristics 55-54
    Q format notation 55-56
ISO 26262
    standards, complying with 22-8
ISO 26262 Standard 22-8
issues, using PIL 44-13

L

Launch and Test Communication with TASKING EDE 46-21
link filters properties
  getting 47-26 50-30
link properties
  about 47-28 50-30 to 50-31 54-51 to 54-52
  `apiversion` 54-53
  `boardnum` 54-54
  `ccsappexe` 54-54
  `numchannels` 54-54
  `page` 54-55
  `procnum` 54-55
  quick reference table 54-51
  `rtdx` 54-55
  `rtdxchannel` 54-56
  setting 47-26 50-30
  `timeout` 54-57
  `version` 54-57
link properties, details about 47-28 50-31 54-52
linking objects
  quick reference 47-27
links
  closing CCS IDE 54-26
  closing Green Hills MULTI® 50-25
  closing RTDX 54-45
  closing VisualDSP++® 47-21
  communications for RTDX 54-36
  creating links for RTDX 54-33
  details 47-28 50-31 54-52
  introducing the tutorial for using links for RTDX 54-29
  loading files into CCS IDE 54-18
  loading files into Green Hills MULTI® IDE 50-17
  loading files into VisualDSP++® IDE 47-14
  quick reference 50-30 54-51
  running applications using RTDX 54-38
  tutorial about using links for RTDX 54-27
  working with your processor 47-15 50-19 54-20

M

main program (`ert_main`)
  modifying 34-5
  operation of 34-5
  static module 34-14
  VxWorks example 35-1
management, memory 56-58
map memory 56-57
map, memory 56-58
math blocks. *See* IQ Math library
MathWorks Automotive Advisory Board (MAAB)
  guidelines, complying with 22-4
MathWorks® Automotive Advisory Board (MAAB) 22-2
MATLAB functions
  `#define` naming 12-36
  parameter naming 12-35
  signal naming 12-35
`Maximum` property 7-4
`MemConst` 7-5
`MemConstVolatile` 7-5
memory
  initialized 56-58
  management 56-58
  map 56-58
  section 56-58
  segment 56-58
  uninitialized 56-58
memory maps 56-57
`Memory section` 7-5
`MemVolatile` 7-5
methods
  tasking.edeapi 46-46
  tasking.edeproject 46-48
  tasking.edeprojectspace 46-48
  tasking.xviewapi 46-48

Minimum property 7-4
MISRA C 22-2
MISRA C®
guidelines, complying with 22-5
MISRA C® guidelines 22-5
model
add blocks 55-26
building overview 55-21
IQmath library 55-59
model design specification
developing 3-1 4-1 5-1
model designs
optimizing for specific hardware with on-target rapid prototyping 38-1
model entry points 33-1
model execution 56-21
Model Explorer
parameter and signal properties 7-2
model reference 47-35 50-39 54-65 56-34
about 47-35 50-39 54-65 56-34
Archive__library 56-35
Archive_library 47-37 50-41 54-66
block limitations 47-38 50-42 54-67 56-36
modelreferencecompliant flag 47-38 50-42 54-68 56-37
setting build action 47-37 50-41 54-66 56-35
Target Preferences block 56-36
Target Preferences blocks 47-38 50-42 54-67
using 47-37 50-41 54-66 56-35
model schedulers 56-21
modelreferencecompliant flag 47-38 50-42 54-68 56-37
modifying rt_OneStep 34-12
MPC555 Target 49-2
mpt (module packaging tool) data object 12-4
MULTI
starting from MATLAB 50-12
stopping from MATLAB 50-12
multitasking 51-56

N

name mangling 17-16
naming rules
applying globally 12-30
changing all #defines 12-36
changing all parameter names 12-35
changing all signal names 12-35
new configuration
creating 46-80
numchannels 54-54

O

object
adivdsp 47-22
ghsmulti 50-26
ticcs 54-48
object properties
about 47-27
quick reference table 47-27 50-31
objects
accessing properties 46-45
calling methods 46-45
creating 46-43
creating objects for CCS IDE 54-16
creating objects for Green Hills MULTI® IDE 50-16
creating objects for VisualDSP++® IDE 47-12
demo 46-46
finding methods 46-44
finding properties 46-45
introducing the objects for CCS IDE tutorial 54-10
introducing the objects for Green Hills MULTI® IDE tutorial 50-10
introducing the objects for VisualDSP++® IDE tutorial 47-7
list of methods 46-46
obtaining method help 46-44
selecting processors for CCS IDE 54-14

selecting processors for VisualDSP++® IDE 47-11
terms 46-37
tutorial about using Automation Interface for CCS IDE 54-10
tutorial about using Automation Interface for Green Hills MULTI® IDE 50-10
tutorial about using Automation Interface for VisualDSP++® IDE 47-7
using 46-38
on-target rapid prototyping
optimizing model designs for specific hardware with 38-1
Open Existing Template Projects 46-22
optimization code 55-60
optimization, processor specific 43-11
optimize code 56-107
option sets 46-24
tutorial 46-75
Options 46-23
overloading 47-26 50-30 54-50
`Owner` 7-6
ownership
effects of settings 13-10
explanation 13-10
Ownership priority 13-8

P

package 12-3
`page` 54-55
Parameter class 12-3
parameter names
changing all 12-35
`Persistence level` 7-6
PIL (processor-in-the-loop) simulation 49-82
benefits of 49-82
getting started tutorial 49-85
hardware connections for 49-85
in plant/controller simulation 49-83
preparation for 49-85
technical overview of 49-83
PIL (processor-in-the-loop) target 49-82
files and folders created by 49-100
in SIL simulation 49-97
in simulation 49-94
using in closed-loop simulation 49-97
PIL block
creating 46-53
PIL issues 44-13
PIL simulation
building and downloading 46-54
coverage and profiling reports 46-58
debugging 46-56
overview 46-50
profiling reports 46-60
`Pointer` 7-8
portnum 50-32
priority and usage 13-3
Definition File priority 13-8
Global priority 13-7
Header File priority 13-8
introduction 13-3
Ownership priority 13-8
Read-Write priority 13-4
See also interdependent settings
processor configuration options
build action 43-12
overrun action 43-13
processor function library. *See* target function library
processor specific optimization 43-11
processor-in-the-loop (PIL)
communications API 39-56
connectivity API 39-53
connectivity API demos 39-59
connectivity configuration 39-54
custom target 39-54
limitations 39-60
rtiostream API 39-56

target connectivity API 39-54
Processor-in-the-Loop (PIL) Verification 46-15
procnum 47-28 50-31 54-55
profile generated code 56-79
profile report
about 56-79
correcting inaccurate profile information 56-82
CPU clock speed 56-88
maximum percent of interrupt interval (Max %) 56-88
maximum time spent in this subsystem per interrupt (Max time) 56-88
number of interrupts counted 56-87
profiling subsystems 56-80
reading 56-86
sample 56-86
STS objects 56-88
timing details 56-81
to generate 56-89
profiling execution
by subsystem 44-17
program execution
main program 34-5
rt_OneStep 34-7
multirate multitasking operation 34-9
multirate single-tasking operation 34-11
reentrancy 34-11
single-rate single-tasking operation 34-8
project generation
selecting the board 43-21 54-58
project-based build process 46-29
projects, create for CCS IDE 56-54
properties
link properties 50-30 54-51
object properties 47-27
referencing directly 47-26 50-30
retrieving 47-24 50-28
function for 47-25 50-29
retrieving by direct property referencing 47-26 50-30
setting 47-24 50-28
property values
definition 12-2
descriptions 7-2
setting 12-8
pure integer code 21-23
and external mode 21-27

Q

Q format 55-56
Q format notation 56-103

R

rapid prototyping
optimizing model designs for specific hardware with 38-1
rate grouping 34-10
Read-Write priority 13-4
real-time target
C166 tutorial 51-22
introduction 49-24
tutorial 49-26
code generation 49-31
example model for 49-28
prerequisites for 49-27
reentrant code 32-5
codegen 23-1
Remove IDE Link Configuration from Model 46-23
requirements
validating with traceability 36-1
reset 55-22
reusable code 32-5
codegen 23-1
rtdx 54-55
RTDX links

tutorial about using 54-27
rtdxchannel 54-56
RTW.copyFileToBuildDir 31-135
rtwdemo_mpf.mdl 12-19
run the DSK confidence test 56-42

S

S-function wrapper generation 26-1
Sample based 7-4
Sample mode 7-4
Sample time 7-3
scheduling 55-6
section,memory 56-58
segment, memory 56-58
Select Preconfigured Target Preference Settings 46-21
selecting boards 43-21 54-58
sessionname 47-29
Set function 7-9
set heap size 43-15
set properties 47-24 50-28
set stack size 43-14
Signal class 12-3
signal names
  changing all 12-35
signed fixed-point numbers 55-56 56-102
simulation 49-82
simulation parameters
  automatic 55-24
simulator
  device cycle accurate 56-9
  use simulators for development 56-9
  use with DSP/BIOS 56-9
simulators, about 56-9
Simulink annotation 17-9
software-in-the-loop (SIL) simulation 49-97
solver modes, permitted 34-7
sorted notes 17-10
source and sink blocks 56-107
source code files, generated 19-2
specification
  developing 3-1 4-1 5-1
stack size, set stack size 43-14
standards
  DO-178B 22-10
  IEC 61508 22-6
  ISO 26262 22-8
standards and guidelines, modeling and coding 22-2
start MULTI from MATLAB 50-12
startup c2000lib 55-24
Stateflow note 17-9
stop MULTI from MATLAB 50-12
Storage class 7-4
Struct (Custom) 7-8
Struct name 7-6
structure-like referencing 47-26 50-30
Sun™ Java™ Runtime Environment 48-4
supported hardware 56-5
symbols for templates
  alphabetical list 17-62
synchronous scheduling 56-22
system requirements 56-4
System Target File Browser 16-4
system target files
  selecting programmatically 16-4

T

target Code Composer Studio 56-54
target connectivity API 39-54
target custom hardware 56-55
target environment
  verifying generated code in 40-1
target function library
  assessing execution time after selecting a library 45-5
  create a custom library 45-8
  optimization 45-2

seeing the library changes in your generated code 45-5
selecting the library to use 45-4
use in the build process 45-3
using with link software 45-2
viewing library tables 45-8
when to use 45-4
. *See* TFL
target hardware setup
communications ports 49-133
jumper settings 49-134
target preferences
fields 46-10
setting 46-8
Target Preferences 46-21
Tools menu 46-23
Target Preferences block in referenced models 56-36
Target Preferences blocks in referenced models 47-38 50-42 54-67
Target Preferences Configuration 46-14
targets
selecting programmatically 16-4
task identifier (`tid`) 34-9
TASKING® CrossView Pro (Debugger)
MATLAB® API 46-4
TASKING® EDE
MATLAB® API 46-4
template projects 46-29
creating 46-76
templates
example with generated file 17-44
rules for creating or modifying 17-66
symbols 17-62
TFL 31-1
build information for function replacements 31-134
cache hits and misses 31-144
conceptual view of function or operator 31-3
defining 31-16
examining 31-139
registering 31-148
reserved identifiers 31-137
RTW.copyFileToBuildDir 31-135
rtwTargetInfo.m file 31-148
selecting in MATLAB Coder 31-8
selecting in Simulink 31-8
sl_customization.m file 31-148
table definition file 31-16
table entry 31-3
target-specific implementation of function or operator 31-3
TargetFcnLibHandle 31-144
tracing generated code 31-143
validating 31-139
Viewer 31-140
workflow 31-7
. *See* target function library
`ticcs` 54-48
`tid` 34-9
`timeout` 54-57
`timeout` 47-29 50-32
timer interrupts 34-4
timer-based models, execution 56-22
timer-based scheduler 56-22
timing 56-21
interrupts 55-6
traceability
for validating requirements 36-1
tutorial
changing identifier names 12-31
changing organization of generated file 17-33
configuring existing models 46-81
creating a data dictionary 12-19
defining all objects in separate file 12-26
defining one object in its own file 12-28
new configuration 46-80
new template projects 46-76
option sets 46-75
tutorial for C6713 DSK blocks 56-43

tutorials
  links for RTDX 54-27
  objects for CCS 54-10
  objects for Green Hills MULTI® 50-10
  objects for VisualDSP++® 47-7

U

uninitialized memory 56-58
`Units` 7-3
use blocks for the C6713 DSK 56-43
use C62x and C64x DSP Library blocks 56-99
use C6713 DSK blocks 56-43
User data type 12-38
`User object type` 7-2

V

`Value` 7-3
`version` 54-57
View, Modify, and Copy Configuration Sets via
  Model Explorer 46-22
viewing target function libraries 45-8
virtualized output port optimization 21-24
VisualDSP++® IDE objects
  tutorial about using 47-7
`Volatile (Custom)` 7-7
VxWorks deployment example 35-1

W

wizard
  data object 12-5
working folder 56-45